VPML: An Approach to Detect Design Patterns of MOF-Based Modeling Languages

MAGED ELAASAR[1], LIONEL C. BRIAND[2], AND YVAN LABICHE[3]

[1]*IBM Canada Ltd, Rational Software, Ottawa Lab*
770 Palladium Dr., Kanata, ON. K2V 1C8, Canada
melaasar@ca.ibm.com
[2]*SnT Centre, University of Luxembourg*
6, rue Richard Coudenhove-Kalergi, L-1359 Luxembourg, Luxembourg
lionel.briand@uni.lu
[3]*Carleton 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 a model-driven engineering methodology, detecting occurrences of design patterns supports the activities of model comprehension and maintenance. With the recent explosion of domain-specific modeling languages, each with its own syntax and semantics, there has been a corresponding explosion in approaches to detecting design patterns that are so much tailored to those many languages that they are difficult to reuse. This makes developing generic analysis tools extremely hard. Such a generic tool is however desirable to reduce the learning curve for pattern designers as they specify patterns for different languages used to model different aspects of a system. In this paper, we propose a unified approach to detecting design patterns of MOF-based modeling languages. MOF is increasingly used to define modeling languages, including UML and BPMN. In our approach, a pattern is modeled with a Visual Pattern Modeling Language (VPML) and mapped to a corresponding QVT-Relations (QVTR) transformation. Such a transformation runs over an input model where pattern occurrences are to be detected and reports those occurrences in a result model. The approach is prototyped on Eclipse and validated in two large case studies that involve detecting design patterns—specifically a subset of GoF patterns in a UML model and a subset of Control Flow patterns in a BPMN model. Results show that the approach is adequate for modeling complex design patterns for MOF-based modeling languages and detecting their occurrences with high accuracy and performance.

KEYWORDS: *Design Pattern, Modeling, MOF, UML, BPMN, QVT, GoF, VPML*

1 Introduction

Model-driven engineering (MDE) [1] is a software development methodology that focuses on the use of models for specifying software systems. Over the years, growing interest in MDE has led to the definition of many modeling languages. It has also led to the development of technologies and tools that generically support the definition, analysis and use of models. Model analysis techniques, in particular, have become popular as they support the comprehension, validation, verification and maintenance of models. One such technique is the detection of occurrences of known design patterns.

In this context, a design pattern is a structure of constrained and interrelated model elements playing complementary roles in a best practice solution to a standard design problem. A pattern occurrence is an instance of a pattern where specific model elements play the pattern roles. Each modeling language may have its own set of design patterns [2]. For example, the Gang of Four (GoF) [3] family of design patterns is commonly used when modeling software with UML [4]. Also, the Control Flow (CF) [5] family of design patterns is popular for modeling business processes with BPMN [6].

As models become larger and more complex, they become more difficult to comprehend. Since design patterns are well-understood fragments of design, it follows that analyzing models in terms of design patterns helps raise their level of abstraction, and thus aids their comprehension [7]. For example, knowing which classes play the GoF Composite pattern roles (i.e., Component, Composite, Leaf) allows a user to easily infer some of their structural and behavioral characteristics. Furthermore, models are often developed by several people, under tight schedules and with changing requirements [8]. This can cause their existing design pattern occurrences to be invalidated over time. Without prior knowledge of these occurrences, it becomes extremely difficult to preserve them when models change. In this context, pattern detection can be thought of as a re-documentation technique at the design level [9]. For example, if a UML model has occurrences of the GoF Observer pattern, where observer classes listen and react to changes in subject classes, it would be much easier to preserve these occurrences if they were known before a change is considered. Moreover, pattern detection can be used for model verification, i.e., detection of anti-patterns or design flaws [10], but this is out of the scope of this paper.

1.1 Problem Description: Automating Design Pattern Detection in Models

In order to perform pattern detection in models, it is extremely important to be able to rely on an adequate (i.e., flexible, accurate and scalable) model-based pattern detection technology. Pattern detection needs to be automated since manually detecting patterns in large models cannot realistically be done and is potentially error-prone. We believe that an automated pattern detection approach must meet three prerequisites. The first prerequisite is to have design patterns specified in a precise machine-readable formalism. Without formal specification, patterns would be ambiguous. The second prerequisite is to be able to automatically drive detection from those specifications. It would be unrealistic to design a detection algorithm for each pattern. The third prerequisite is to be able to report pattern occurrences in a structured way. Without an efficient way to represent a possibly large number of pattern occurrences, analyzing them would not scale in practice.

Other desirable requirements of a pattern detection approach include: 1) being applicable to a family of modeling languages to allow building a generic and reusable analysis tool, 2) having an intuitive and concise syntax to ease the expression and maintenance of pattern specifications, 3) requiring technical skills that already exist or are easy to acquire by pattern designers to reduce the learning curve, 4) having facilities (e.g., inheritance and composition) to cope with the variability and the complexity of pattern specifications, 5) being scalable to

detect pattern occurrences in large and complex models, as this would fit the needs of industry today, and 6) leveraging existing technologies to facilitate tool support (specification and detection) and integration with modeling tools, since users would appreciate value-added features to the tools they already use rather than having to use new ones. Although many approaches to pattern detection have been proposed in the literature, our review (Section 2) indicates that they do not satisfy one or more of these prerequisites or requirements.

1.2 Proposed Solution: A New Approach to Pattern Detection in Models

In this paper, we propose a new approach to design pattern specification and detection that satisfies the prerequisites and requirements described in Section 1.1. Specifically, the approach allows specifying patterns for the MOF [11] family of modeling languages (requirement 1). MOF is a popular standard that is widely used to define the abstract syntax (i.e., the concepts and their relationships and constraints) of modeling languages, including general purpose ones, like UML, and domain specific ones (DSML), like BPMN (for business process modeling). MOF defines the abstract syntax of a modeling language with a metamodel, which describes how to organize information in a model. MOF metamodels often include constraints expressed in OCL [12], a MOF-based constraint language that supports first-order predicate logic and set theory. MOF-based transformation languages (e.g., QVT [13], ATL [14]) also exist and use OCL as their query language. In the remainder of the paper, we always mean MOF-based (modeling) languages when we discuss (modeling) languages and we mean MOF-based model when we discuss model, unless otherwise specified.

Our approach defines a new pattern specification DSML (prerequisite 1) called the Visual Pattern Modeling Language (VPML). The concise visual notation of VPML makes pattern specifications easy to create and maintain (requirement 2). The abstract syntax of VPML is defined with a small metamodel that defines concepts and relationships, and that should be familiar to design pattern practitioners (requirement 3). The language also has features, like pattern inheritance/composition and contextual properties/operations, to handle the complexity and variability of pattern specifications (requirement 4).

Furthermore, the semantics of VPML is defined by mapping it to that of QVT-Relations (QVTR) [13], a model-to-model transformation language. We choose QVTR as it is declarative, it has built-in pattern detection semantics (hence it is easy to map from VPML), and it is a standard (hence it is interoperable between tools). In fact, one can think of VPML as a simpler (due to being specific to the domain of pattern specification), more concise (due to having highly expressive features), and more visual (due to its visual notation) alternative to specifying patterns in QVTR directly. We also automatically derive a scalable QVTR transformation from a VPML model (prerequisite 2), using a model-to-text mapping. The transformation runs on an input model, looking for pattern occurrences and adding them to a result model (requirement 5). This model conforms to a newly-defined language called pattern results (PResults), which allows pattern occurrences to be represented using a tree-based data structure (prerequisite 3).

Furthermore, we prototype our approach on the Eclipse platform (requirement 6). Specifically, we use the Eclipse Modeling Framework (EMF) [15], which implements a subset of MOF called Essential MOF (EMOF), as our modeling infrastructure. We also use many EMF-based projects in our implementation. For example, we use the Graphical Modeling Framework (GMF) [16] to implement our editors for VPML and PResults. We also use the open-source tool Medini QVT [17] to execute QVTR transformations on EMF-based models.

We validate our approach with two large case studies that involve the detection of occurrences of design patterns in models conforming to different MOF-based modeling languages. The first case study involves the specification of eleven GoF design patterns for UML (a general purpose modeling language) and detecting their occurrences in a large UML design model of an open-source software project. The second case study involves the specification of ten CF design patterns [5] for BPMN (a DSML) and detecting their occurrences in a large BPMN model designed for the financial services industry. In both case studies, we evaluate the adequacy of the approach using three criteria: 1) the ability to express a complex family of design patterns (with multiple roles, conditions and variants), 2) the accuracy of the detection, and 3) the performance of the detection. Results show that our approach adequately expresses complex patterns in the chosen MOF-based modeling language. They also show that the approach detects occurrences of such patterns with high accuracy and performance.

The rest of this paper is structured as follows. Section 2 reviews the literature on design pattern detection approaches. The definition of VPML, our Visual Pattern Modeling Language, and how it addresses common pattern specification requirements is discussed in Section 3. Section 4 illustrates the detection of patterns through a mapping from VPML to QVTR. A case study on specifying GoF patterns and detecting their occurrences in a UML model is described in Section 5. Section 6 describes a case study on specifying CF patterns and detecting their occurrences in a BPMN model. A discussion of insights gained from carrying out the case studies is provided in Section 7. Section 8 highlights the limitations of our approach and proposes possible future mitigations. Finally, Section 9 summarizes the contributions and provides the conclusions.

2 Related Work

In our search for a suitable solution to the problem of pattern detection in MOF-based models, we reviewed many approaches in the literature. In this section, we categorize these approaches, discuss representative examples in each category (but do not report on an exhaustive list of papers as in a systematic literature review), highlight why they were not considered fully satisfactory based on the prerequisites and requirements discussed in Section 1.1, and compare them to our approach. Other relevant reviews exist ([18] and [19]).

2.1 Logic Approaches

Some approaches to pattern detection use logic-based formalisms to encode pattern constraints and inference engines to detect them. Most of these formalisms (e.g., SQL [20] and Prolog [21]) are based on first-order

predicate logic, while some are based on higher-order logic (e.g., temporal logic [22], [23] and monadic logic [24]). Despite their scientific contributions, we believe that these approaches have one or more of the following drawbacks. First, they require skills in mathematics and logic that might not be available or hard to learn by personnel in charge of specifying patterns (especially for non software-related MOF languages), making them difficult to use. Second, they require a first phase of detection where a model to be analyzed is converted from its original representation into another one (e.g., Prolog terms [21], a binary decision diagram [25], a fact base [20], or a Petri net [26]) more suitable for the proposed formalism, which would negatively impact scalability. Third, conversion rules are usually specified using a different formalism (e.g., a programming language) than the one used to specify patterns, making the specification more complex. In comparison, our approach specifies patterns using VPML (a dedicated DSML) directly on metamodels and using technologies (e.g., OCL) a pattern practitioner is most probably familiar with (or can easily learn). For detection, the VPML models are automatically converted to QVTR transformations that run on instance models without conversion.

2.2 Graph Approaches

Graph approaches also first require the conversion of the model to be analyzed into a graph representation before detection. For example, one approach [27] initially translates source code into a graph representation then applies a series of graph transformations that gradually add higher-level relationships between nodes until pattern level relationships are added. Other approaches [28], [29] represent both source code and patterns as graphs and use a graph isomorphism algorithm to find matches. Another approach [30] reformulates the graph matching problem as a constraint satisfaction (CSP) problem and solves it with CSP solvers. In comparison, our approach works on models in their native MOF-based graph representation directly, hence avoids a costly upfront conversion.

2.3 Quantitative Approaches

Other approaches [31], [32], [33] use quantitative analysis methods to detect patterns. They devise mathematical formulae that assign scores to both model elements and pattern roles. For example, one of the approaches [32] assigns a unique prime number to each kind of UML class feature (e.g., Property or Operation) and raises it to a power that equals the number of features of that kind in a class. The score for each class is the product of these numbers. Checking a pattern condition (e.g., a class playing a certain role must have two operations) is then reduced to an arithmetic operation over these scores (e.g., whether the score for a class is a multiple of the score for the role). However, quantitative approaches suffer from practical drawbacks. First, they are mainly suitable for encoding quantitative conditions. Other conditions (e.g., the name of an operation must contain string "X") are harder to encode. Second, it is not trivial for a pattern designer who is not trained in mathematics to devise good mathematical formulae to capture the required pattern conditions. Third, based on model size, some formulae may go outside acceptable numerical ranges. In comparison, our VPML language can specify any pattern condition

directly or through OCL. It also has facilities (e.g., pattern composition) to cope with pattern variability and complexity, in addition to a scalable detection algorithm encoded as a QVTR transformation.

2.4 Modeling Approaches

There exist approaches that specify patterns at the instance model level of each modeling languages. For example, some approaches [34], [35] specify GoF patterns using UML collaborations, detect them with a specific algorithm, and report matches as collaboration occurrences. The problem here is that not all languages have, or can be extended to have, pattern specification semantics. Also, what works for one modeling language may not easily be applicable to others. Another approach [36] assumes a metamodel is represented as a UML model and specifies a pattern with a UML object diagram. The diagram is considered an implementation of a query operation that detects occurrences of that pattern. Pattern composition is supported through query invocations although the UML syntax for that is not clear. In comparison, our approach specifies patterns visually using VPML directly on a target MOF-based metamodel. VPML provides several complexity handling features (like pattern composition and derived properties) and its detection semantics is well defined with a mapping to QVTR.

2.5 Metamodeling Approaches

Some modeling approaches try to be more generic by specifying patterns at the metamodel level. For example, one approach [37] augments MOF with collaboration semantics and specifies patterns with a metamodel-level composite structure diagram. Unfortunately, only a simple pattern is shown and no detection semantics is discussed, thus making it difficult to assess the genericity of the approach. Another approach [38] specifies patterns by extending their target metamodels, adding pattern-related constraints. For example, the structural aspect of GoF patterns is defined by extending the classifier-related metaclasses, while the behavioral aspect is defined by extending the interaction-related metaclasses of the UML metamodel. Pattern detection is then performed using algorithms specific to each kind of extension, an approach that is not trivial to extend to new DSMLs. In comparison, our approach allows patterns to be specified using a single formalism (VPML) and their occurrences to be detected using a single algorithm (encoded as a mapping to QVTR).

2.6 DSML Approaches

Other approaches define new DSMLs for pattern specification. For example, one approach [39] defines a new DSML that focuses on structural constraints only. The approach lacks complexity management mechanisms for patterns (like composition). Another approach [40] proposes the specifications of idioms with metamodels and the specification of patterns with a DSML. Detection occurs in two phases: (i) idiom models are populated by specific algorithms (written in a programming language) that query the input model, and (ii) pattern specifications are translated into a system of constraints that gets solved to find occurrences. Another approach [41] proposes a new

metamodeling formalism based on an extension to BNF for graph definition. The formalism is used to define the abstract syntax of modeling languages. The approach also proposes a first-order logic language that works on that formalism and a domain-specific language for pattern specification. The approach has several drawbacks. First, it is not MOF-based, making it hard to integrate with MOF-based tools. Second, the metamodeling formalism has fewer features compared to MOF (e.g., no operations). Third, there is no complexity handling mechanism (like pattern composition). In comparison, our approach allows both idioms and patterns to be specified using VPML, directly on MOF-based metamodels. It also has pattern variability and complexity handling mechanisms (e.g., pattern composition and derived properties).

2.7 Transformation Approaches

Some approaches use transformation-based methods. One approach [42] uses a profiled UML object diagram to specify transformations with support for hierarchal and polymorphic substructure definitions. Although it was not proposed for pattern detection per se, one might foresee the use of such an approach to visually specify a pattern detecting transformation. Unfortunately, the approach focuses on modeling the target (the occurrence) rather than the source (the pattern) of a transformation rule. Another approach [43] specifies patterns adequately with transformation rules and generates search plans for them, optimized by statistics from typical instance models. However, the approach does not discuss ways to deal with pattern complexity/variability. Another approach [44] specifies patterns visually with graph mapping rules. The rules, which are run by an inference engine, detect pattern occurrences and add them as annotations to the graph, which could support pattern composition. However, the approach works on its own graph definition and would need MOF-based models to be translated first into it. In comparison, our VPML language specifies patterns only and corresponding occurrence structures are automatically derived. VPML works on the MOF-based representation of models directly and provides features to deal with the complexity and variability of patterns. Pattern detection is defined by a mapping to QVTR. The resulting QVTR transformations run directly on MOF-based input models.

2.8 Summary

In summary, previous approaches differ from our approach in one or more of the following ways: (i) they use specification formalisms that do not leverage the MOF-based representation of models; (ii) they use specification formalisms that are complex (require sometimes hard to find skills) or restricted (to certain constraints or languages); (iii) they use a second formalism to pre-process models or to specify certain aspects of patterns; (iv) they cannot cope efficiently with pattern variability or complexity; (v) they use an un-scalable detection algorithm or require the design of a custom algorithm for each pattern or modeling language.

Furthermore, some reviewed approaches have not been validated in significant case studies or in terms of both their accuracy and performance. In this paper, we report on the expressiveness, accuracy and performance of our

approach using two large case studies (the second case study comes from industry). Also, many approaches have focused on analyzing source code, sometimes by transforming it first into a user-defined model representation. In contrast, our case studies analyzed non-trivial realistic models conforming to standard metamodels. We publish all the details of our case studies online (as pointed out for each case study) to help replication and comparisons.

3 Design Pattern Specification with VPML

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

We note that we could have also defined VPML as a profile of UML collaboration diagrams, whose syntax, semantics and notation are close to VPML's. However, a profile may inherit extra baggage from UML that is not relevant to or needed for the problem. For example, a pattern that is modeled as a stereotyped UML collaboration inherits the ability to have its behavior specified with a behavioral diagram, a feature that would not be useful when pattern roles are played by MOF metaclasses that do not specify behavior. Moreover, there is no standard way to extend the UML notation to accommodate the needs of a profile (aside from adding icons to stereotyped elements). Also, not all UML tools offer mechanisms to extend the notation in proprietary ways.

3.1 Running Example

The running example consists of a simple DSML we defined for circuit logic design (CLD) and two of its design patterns (*HalfAdder* and *FullAdder*) [45]. The metamodel of CLD is shown in Figure 3-1 (left). Basically, a logic component can have a number of input pins and output pins. A gate is a component with a specific kind of logic function (AND, OR, XOR ...). A circuit is a container component for other components and wires. A wire connects exactly two pins together. The notation of the various logic elements is shown in Figure 3-1 (right).

The *HalfAdder* pattern is a circuit that adds two one-bit numbers. It has two input pins A and B, each representing a number, and two output pins S and C, representing the sum and carry, respectively, of adding these two numbers. The simplest *HalfAdder* variant (Figure 3-2 left) connects the input pins to S, through an XOR gate, and to C, through an AND gate. Similarly, the *FullAdder* pattern is a circuit that adds two numbers but accounts for values carried in as well as out. It has three input pins A, B, and C_{in} (representing a bit carried in from a past addition) and two output pins S and C_{out} (representing a bit carried out to the next addition). A *FullAdder* can be designed in a variety of ways but a common variant (Figure 3-2 right) is defined in terms of two *HalfAdders* by

connecting pins A and B to the input of one *HalfAdder*, connecting the sum from that to an input of the second adder, connecting C_{in} to the other input and finally ORing the two carry outputs.

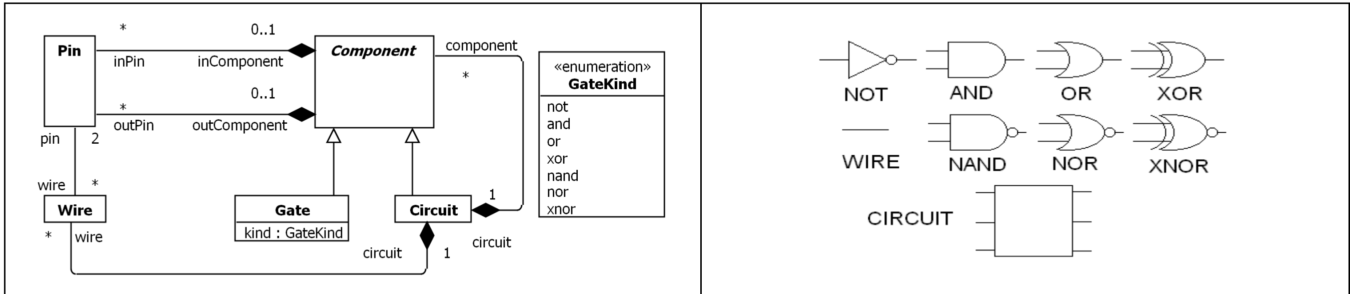

Figure 3-1 CLD metamodel (left) and CLD notation (right)

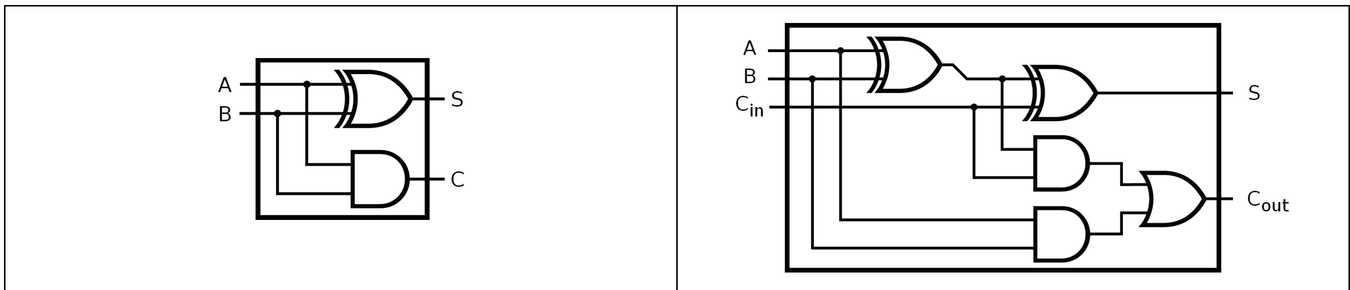

Figure 3-2 HalfAdder design pattern (left) and the FullAdder design pattern (right)

3.2 Specifying a Pattern Catalog

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

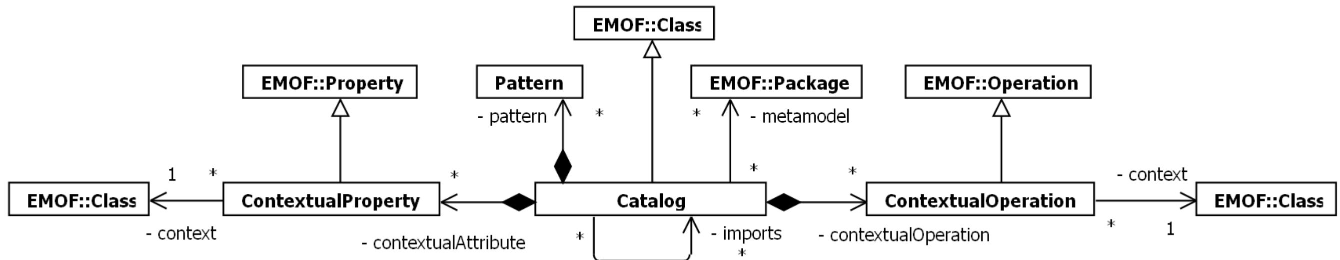

Figure 3-3 Excerpt from the VPML metamodel for catalog definition

Furthermore, Catalog is a subclass of *EMOF::Class* and therefore a catalog can have a name. It can also define a set of attributes (of type *EMOF::Property*) and a set of operations (of type *EMOF::Operation*), whose derivations and bodies are expressed in OCL and used within the catalog (as will be seen later) to simplify pattern specifications. A catalog can also define a set of contextual attributes and operations (of types *ContextualProperty* and *ContextualOperation*, respectively) that are not originally part of, but are dynamically woven in, a context

metaclass, without modifying its possibly standardized metamodel. One example is catalog *Adders* defining contextual attribute *wiresTo:Set(Pin)*, in the context of metaclass *Pin*, with the OCL derivation expression [*self->closure(wire.pin)*] that calculates the transitive closure of other pins connected to a context pin through wires. Another example is catalog *Adders* defining operation *isWiredTo(p:Pin):Boolean*, in the context of metaclass *Pin*, with an OCL body [*self.wiresTo->includes(p)*] that checks whether a given pin is wired to the context pin. Finally, *Catalog* being a subclass of *EMOF::Class*, a catalog can reference other catalogs as super classes and inherits their attributes, operations and patterns. A catalog may also reference other catalogs as imports and access (in order to use) their attributes, operations and patterns.

3.3 Specifying a Pattern

VPML allows patterns to be specified declaratively as a set of interrelated and constrained roles. Figure 3-4 is an excerpt from the VPML metamodel related to pattern definition. Class *Pattern* is a subclass of *EMOF::NamedElement* and therefore a pattern can have a name. It can also define a set of one or more roles. Class *Role* is a subclass of *EMOF::TypedElement* and therefore a role can have a name and a type (a metaclass playing this role in the pattern). It can also have a set of conditions expressed with *RoleAttribute* (specifying a value for an attribute), *RoleInterrelationships* (specifying a value for an association end), or an OCL *condition* directly. A pattern may also have a set of OCL *conditions* that typically restrict multiple related roles.

Furthermore, a role can be flagged as exposed, which makes it visible to other patterns using the role's defining pattern (more on this in Section 3.5). Conversely, a non-exposed role is not visible to using patterns. Finally, a pattern may specify that a subset of its roles is reported on. The subset is represented by property *Pattern::reportedRole* in the metamodel. It is ordered, allowing role bindings, which are reported for a pattern occurrence to be ordered based on the relative importance of these roles (more on result reporting in Section 4).

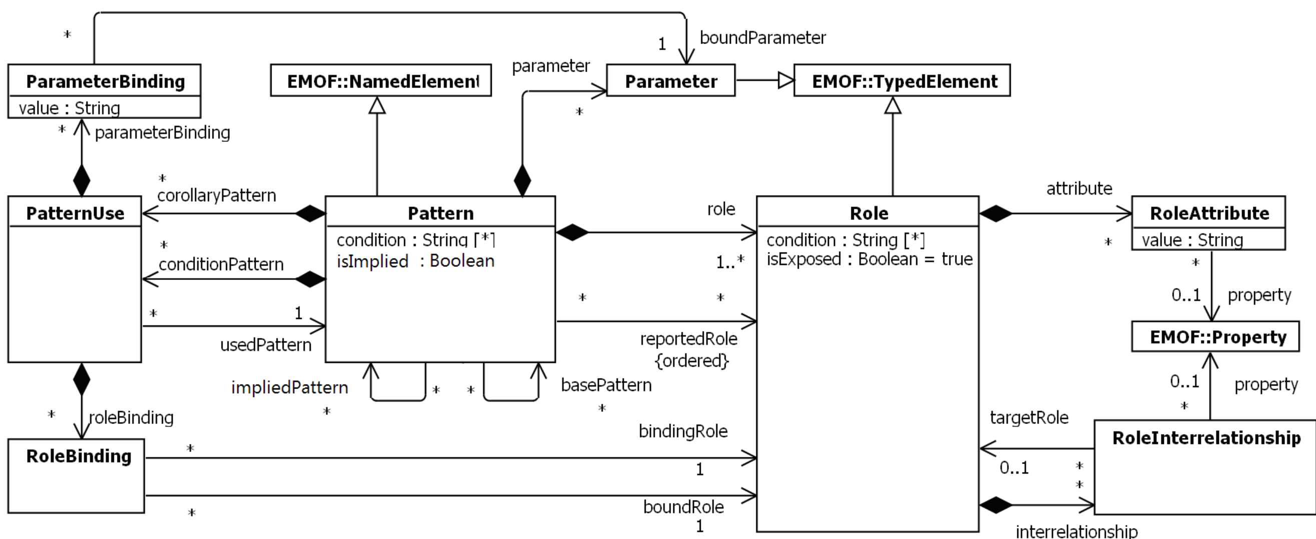

Figure 3-4 Excerpt from the VPML metamodel for pattern definition

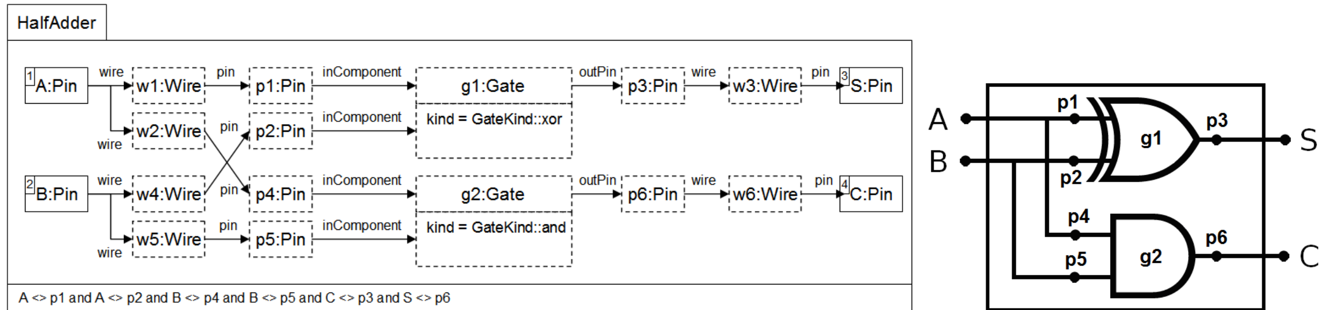

Figure 3-5 Basic VPML specification (left) of the HalfAdder pattern (right)

We now know enough of the metamodel to start specifying the *HalfAdder* pattern (Figure 3-5 right) while introducing the VPML notation. The VPML specification of *HalfAdder* (Figure 3-5 left) is depicted with a solid-line box having three compartments: a name compartment (top), a details compartment (middle) and a conditions compartment (bottom). The details compartment shows the roles involved in the pattern and how they are interrelated. Roles are represented by smaller boxes, while interrelationships are represented by arrows. Each role has a name compartment showing its name and type (e.g., A : Pin). Exposed roles are depicted with solid-line boxes, while non-exposed roles are depicted with dashed-line boxes. In this particular pattern, the circuit's pins (A, B, S and C) are exposed roles, while the intermediate details of how they are wired (i.e., wires w1-w6, gates g1-g2, and pins p1-p6) are non-exposed roles. Furthermore, a pattern shows its reported roles annotated with small number boxes in their top-left corners, representing their reporting order. For example, pin A is the first reported role and role B is the second. Some roles (e.g., g1) have a second compartment for their required attribute value (e.g., *kind = GateKind::and*). While not shown here, some roles may also have a third compartment for their extra OCL conditions, if any. The arrow label on a role interrelationship specifies an attribute in the source role's type that relates the two roles together (e.g., role A is related to role w1 through attribute *Pin::wire*). Traversing the arrows in this pattern from input pins (A and B) to output pins (S and C) shows exactly how they are interrelated according to the *HalfAdder* pattern. The third compartment of a pattern shows OCL conditions that typically involve multiple roles. For *HalfAdder*, conditions are added to guard against a pin simultaneously matching pin roles on the two ends of a wire (e.g., S and p6), which is possible according to the CLD metamodel (both pins are in the same collection *Wire::pin*). Since there are six wire roles (w1-w6) in the specification, we specify six (pinX <> pinY) conjunctive expressions in the condition.

3.4 Improving Specification Expressiveness

We define the expressiveness of a pattern specification as a measure of its ability to specify different conditions or alternative specifications of a pattern, in an easy, clear, and concise manner. For example, one issue with the current specification of *HalfAdder* is that it specifies only one way of wiring the circuit's external pins to the intermediate gates, which is direct wiring. In circuit design, it is possible that wires go through several intermediate pins before/after being wired to the gates without affecting the function of the circuit. For example,

w1 may be connected to another free pin p7, which in turn may be wired to p1. The current specification will fail to detect this occurrence, which is an obvious limitation.

In order to address this limitation, we need to change the *HalfAdder*'s VPML specification to account for intermediate pins from the circuit's input pins (A and B) to the gates, and similarly from the gates to the output pins (S and C). We can do that by first defining new contextual attributes (with OCL derivation expressions) in the catalog, as shown in Figure 3-6. Notice that attribute *wiresTo* (line 1) uses the transitive closure operation in OCL to find all other pins that are wired to a context pin directly or indirectly. The other attributes (lines 2-4) are convenient ones to allow bypassing the gate pins. Then, we can change the *HalfAdder* specification, as shown in Figure 3-7 (left), to connect the input and output pins directly to the gates with these new properties. For example, compare pin A that is connected to gate g1 directly with property *inputTo*, with its counterpart in Figure 3-5 (left) that is connected to gate g1 through intermediate wire w1 and pin p1. A similar comparison can be made for output pin S, whose connection uses property *outputTo*. The resulting *HalfAdder* specification expresses the connection conditions more accurately, covering possible intermediate wire-pin steps (Figure 3-7 right), and at the same time it is more concise (it has fewer roles and role interrelationships).

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

Figure 3-6 Contextual properties used to generalize the HalfAdder pattern specification

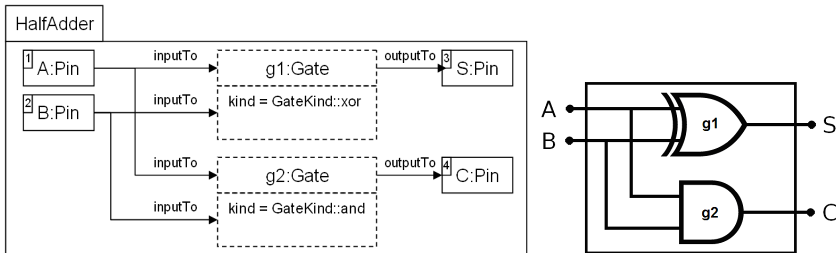

Figure 3-7 More accurate and concise VPML specification (left) of the HalfAdder pattern (right)

3.5 Improving Specification Abstraction

When patterns become complex, their specifications tend to be large and cluttered, making them hard to read and maintain. In Section 3.4, we showed how using contextual attributes helped make the *HalfAdder* specification more accurate and concise. However, consider the *FullAdder* specification in Figure 3-8 (left), which corresponds to the pattern's design in Figure 3-8 (right). Notice how the input pins are connected with contextual properties through gates g1-g5 to the output pins, in a way that parallels the circuit layout. This pattern specification is still cluttered. Fortunately, VPML provides another way to abstract (i.e., make pattern specifications at higher levels of abstraction by hiding details) a large pattern by using smaller patterns as conditions (pattern composition). The VPML metamodel (Figure 3-4) defines class *PatternUse*, which references a pattern and owns a set of *RoleBindings*. Each role binding binds an exposed role (*boundRole*) of the used pattern to some role

(*bindingRole*) of the using pattern. A pattern can have a set of *PatternUses*, referenced as *conditionPatterns*, as pre-conditions (i.e., the used patterns need to hold for the using pattern to hold).

Fortunately, in the case of *FullAdder*, the circuit can be designed using two *HalfAdder* circuits (Figure 3-9 right). This allows us to refactor the *FullAdder* pattern to use the *HalfAdder* pattern twice as a condition (Figure 3-9 left). A condition pattern-use is depicted with an ellipse showing the name of the used pattern. Role bindings are depicted with non-arrow lines connecting the pattern-use to the binding roles, while showing the name of the bound roles as labels. For example, the A and B pins of the first *HalfAdder* are bound to the A and B pins of the *FullAdder*. The S pin of that *HalfAdder* is bound (through pin p2) to the A pin of the other *HalfAdder*, while its C pin is bound (through pin p1) to gate g5. Notice that the refactored *FullAdder* specification is more abstract (higher level and fewer detailed), hence easier to read and maintain, than the original specification.

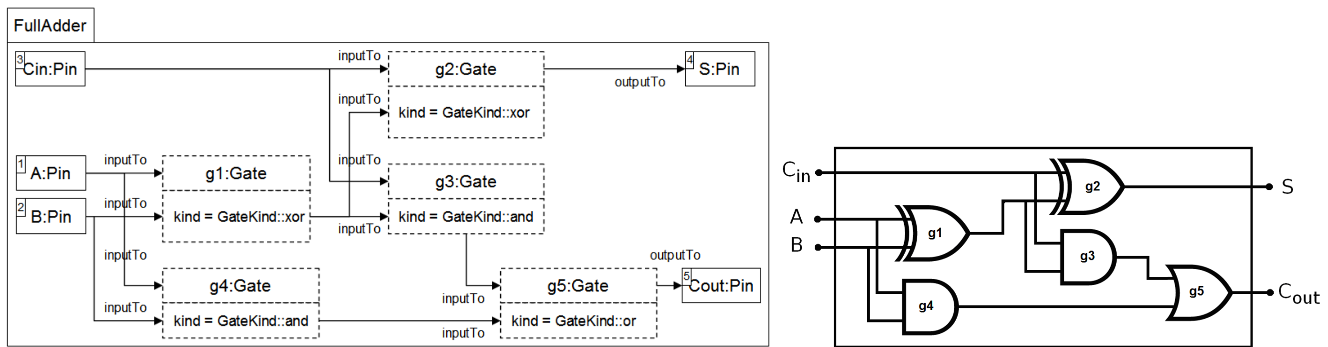

Figure 3-8 Basic VPML specification (left) of the FullAdder pattern (right)

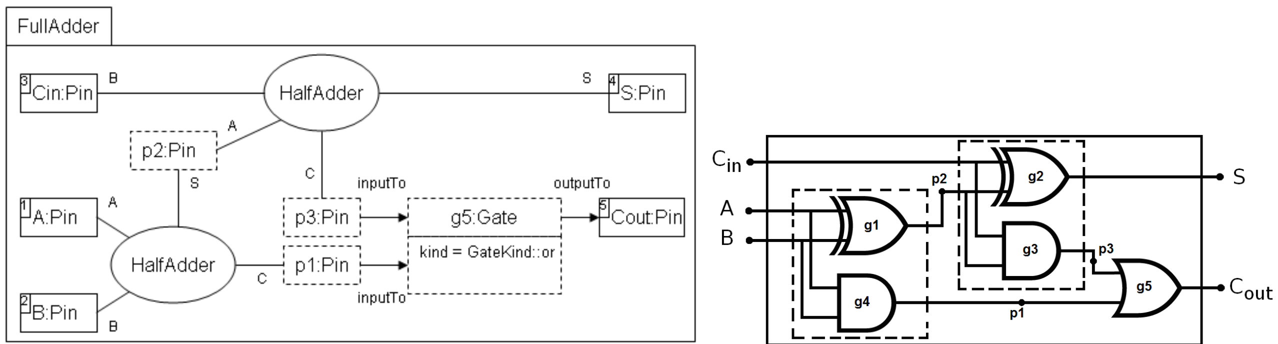

Figure 3-9 More abstract VPML specification (left) of the FullAdder pattern (right)

3.6 Dealing with Pattern Variability

3.6.1 Base pattern

Design patterns may have a number of variants. In Section 3.4, we showed how to deal with one kind of variability (intermediate relations) through the use of contextual properties specifying transitive closures. Variability can also impact other kinds of pattern details (like roles, interrelationships and/or conditions), resulting in sufficiently different variants. Instead of specifying each variant separately with all its details, it makes more

sense to specify common details in a separate pattern that gets used as a condition by each variant. A variant then adds its own unique details on top. Since a condition pattern must hold for its using pattern to also hold (Section 3.5), the common details must hold for the specifics to hold.

For example, consider another variant of *FullAdder* where the AND and OR gates are replaced by NAND gates, without affecting the function of the circuit. Instead of specifying this new variant from scratch, it would be desirable to specify it in a way that common details are shared with the original variant. With VPML, one can specify those common details in another pattern, named *FullAdderBase* (Figure 3-10 left) that gets used as a condition pattern by both variants. Notice that *FullAdderBase* is quite similar to the original variant (Figure 3-8 left) except for two differences: (i) gates g3-g5 are exposed (solid line on rectangle) and do not have their *kind* attribute conditions (allowing such conditions to be added by the extending variants), and (ii) the exposed roles (A, B, S, C_{in} and C_{out}) are missing their little number boxes, which means they are not reported on (since *FullAdderBase* represents only a subset of the details of a pattern variant, it should not itself report any pattern occurrence). Having specified *FullAdderBase*, we now rename the original variant to *FullAdder1* and refactor it to use *FullAdderBase as a condition* (Figure 3-10 right). On top of that, *FullAdder1* adds its own specifics: it un-exposes gates g3-g5 (dashed line on rectangle), adds its specific *kind* attribute conditions (to gates g3-g5) and specifies reported roles using the little number boxes (since *FullAdder1* represents all the details of a pattern variant, it can report on occurrences of that variant).

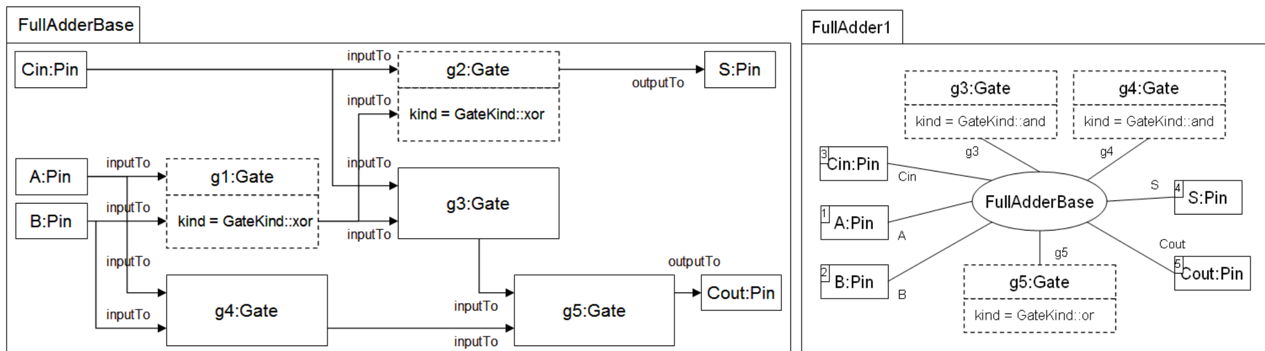

Figure 3-10 VPML specification of FullAdderBase pattern (left) and FullAdder1 variant (right)

However, with a large number of roles in a pattern, like in *FullAdderBase*, using it as a condition is cumbersome syntactically and notationally, since an equivalent large number of role binding objects need to be created and depicted in the using patterns (Section 3.5). Fortunately, the VPML metamodel (Figure 3-4) provides a short-cut syntax to this situation by allowing a pattern to directly reference (i.e., without using a condition pattern with its role bindings) another pattern as a base. Referencing a pattern as a base (syntactically through the *Pattern::basePattern* property and notationally by following the name of the pattern by "->" then the name of the base pattern) is semantically equivalent to, although syntactically and notationally more concise than, using it as a condition pattern with role bindings to similarly-named roles in the used pattern. For example, notice how the

FullAdder1 pattern in Figure 3-11 (left), which references the *FullAdderBase* pattern as a base (with the "-> FullAdderBase" notation), is specified more concisely than its version in Figure 3-10 (right), which uses *FullAdderBase* as a condition pattern-use and binds the roles in the pattern to similarly-named roles in the condition pattern (e.g., role A in *FullAdder1* is bound to role A in *FullAdderBase*) with the ellipse/connector notation. The new variant *FullAdder2* is similarly specified (Figure 3-11 right), but requires all its gate roles to be of *kind* NAND instead of AND/OR as in *FullAdder1* (Figure 3-11 left).

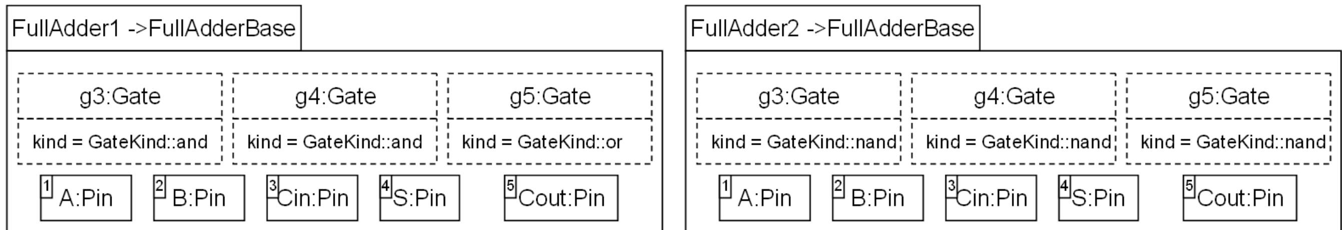

Figure 3-11 More concise VPML specifications of the FullAdder1 and FullAdder2 variants using base pattern

3.6.2 Corollary Pattern

When multiple variants (e.g., P1, P2 and P3) have been specified for a pattern P, any one of them may explicitly be used as a condition of another pattern Q. This is handled by the mechanisms we have discussed so far. However, we may need to make the specification more generic and state that pattern Q requires pattern P as a condition without explicitly specifying which variant of P to use (i.e., any of P's variant can be used). In this case, the VPML variability feature described thus far (i.e., condition patterns) does not help: using all variants as conditions gives the unintended semantics of requiring them all to hold at the same time. In this section, we introduce another feature in the VPML metamodel (Figure 3-4) which can help in this situation. We first introduce the feature, and then we explain how it can help in this situation.

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

Interestingly, the corollary feature can help express the situation of generically using a multi-variant pattern as a condition without explicitly using individual variants. The idea is to define a pattern P as a corollary of each of its variant patterns (e.g., P1, P2, P3). This means that P holds once any of the variants holds (i.e., P1 *implies* P, P2 *implies* P, …). When pattern P is also used as a condition of pattern Q (i.e., P *implies* Q), it means that Q transitively holds when any variant holds (i.e., P1 *implies* Q, P2 *implies* Q, …), which is the desired semantics.

Furthermore, a corollary is syntactically represented as a pattern-use referenced by a pattern through its *corollaryPattern* property, while notationally, it is depicted as a double-line ellipse. The used pattern in the case of a corollary must be flagged as *isImplied* and depicted as a double-line box.

For example, consider a CLD pattern *2BitAdder* that calculates the sum of two 2-bit numbers. The *2BitAdder* pattern (Q) is specified in VPML (Figure 3-12 right) to use the *FullAdder* pattern (P) twice as a condition (i.e., *FullAdder implies 2BitAdder)*. Since the *FullAdder* pattern has two variants (*FullAdder1* and *FullAdder2*, Section 3.6.1), it needs to be flagged as *isImplied* pattern (Figure 3-12 left) and used as a corollary by each variant (e.g., *FullAdder1* in Figure 3-13 left). This makes pattern *FullAdder* to hold once any of the two variants holds (e.g., *FullAdder1 implies FullAdder*), which causes *2BitAdder* to transitively hold (e.g., *FullAdder1 implies 2BitAdder*).

Furthermore, the VPML metamodel (Figure 3-4) provides a semantically equivalent but notationally more concise short-cut syntax for a corollary pattern that maps the roles of the used pattern to similarly named roles of the using pattern. The concise syntax is to directly reference (i.e., without using a corollary pattern with its role bindings) the used pattern in this case through the *Pattern::impliedPattern* property. The concise notation is to follow the name of the pattern by "->>" then the name of the implied pattern. For example, the *FullAdder1* specification in Figure 3-13 (left) has been re-expressed using the more concise notation in Figure 3-13 (right).

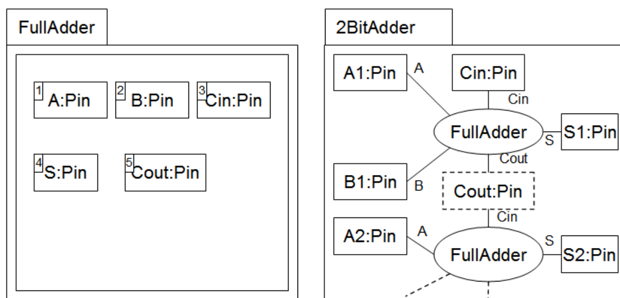

Figure 3-12 VMPL specification of FullAdder corollary (left) and its use by 2BitAdder pattern (right)

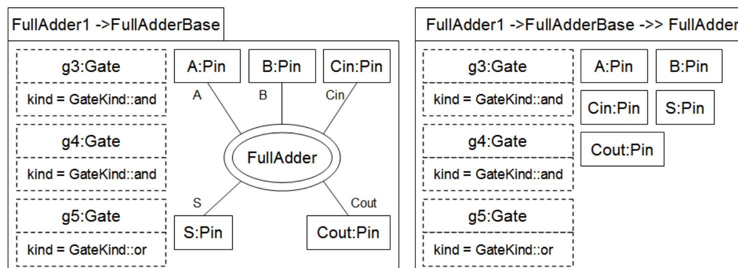

Figure 3-13 VMPL specification of FullAdder1 using a corollary (left) or using an implied pattern (right)

3.6.3 Parameterized Pattern

Sometimes a pattern has small variability or one that only affects unexposed roles (hence cannot be handled using a base pattern since this requires specialized roles to be exposed). The VPML metamodel (Figure 3-4) allows a pattern to capture this variability using pattern parameters. Class *Parameter* is a subclass of *EMOF::TypedElement*, and can therefore have a name and a type (only primitive types like Integer, Boolean, and

enumeration are allowed). A parameter can be used in a pattern specification by referencing it in OCL expressions. When a parameterized pattern is used by another pattern, as a condition or a corollary, the pattern-use must bind values to those parameters. Those values can themselves be expressed in OCL.

For example, consider the *HalfAdder* specification in Figure 3-7 left. Two input pins are wired to two gates, each of which is wired in turn to an output pin. While this circuit design with two gates works for the *HalfAdder*, it might also work (albeit with different gate kinds) for other circuits. In order to make such a design more reusable, it can be specified as a parameterized *TwoGates* pattern (Figure 3-14 left). The pattern defines two parameters k1 and k2 of type *GateKind* in their own compartment below the pattern's name. The *HalfAdder* specification can then be refactored to use the *TwoGates* pattern as a condition (Figure 3-14 right). The pattern-use specifies values for the two parameters in a compartment below the name.

This feature may also be used to provide an alternative specification for *FullAdderBase* (Figure 3-10 left) and its variants (Figure 3-11). The new specification of *FullAdderBase* (Figure 3-15 left) is defined based on the *FullAdder* specification in Figure 3-9 but using two instances of *TwoGates* rather than *HalfAdder* as conditions. In each instances, parameter k1 is bound to the value *GateKind::xor*, while parameter k2 is bound to a parameter (t1/t2) defined on *FullAdderBase*. Finally, the *kind* attribute value of gate g5 is set to a parameter (t3) defined on *FullAdderBase*. The parameterized *FullAdderBase* pattern is then used as a condition by the two *FullAdder* variants (Figure 3-15 middle/right), where the (t1-t3) parameters are bound differently.

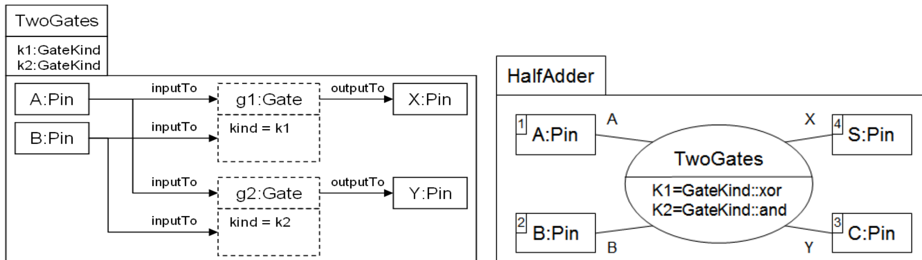

Figure 3-14 VMPL specification of a parameterized pattern (left) and its usage by HalfAdder (right)

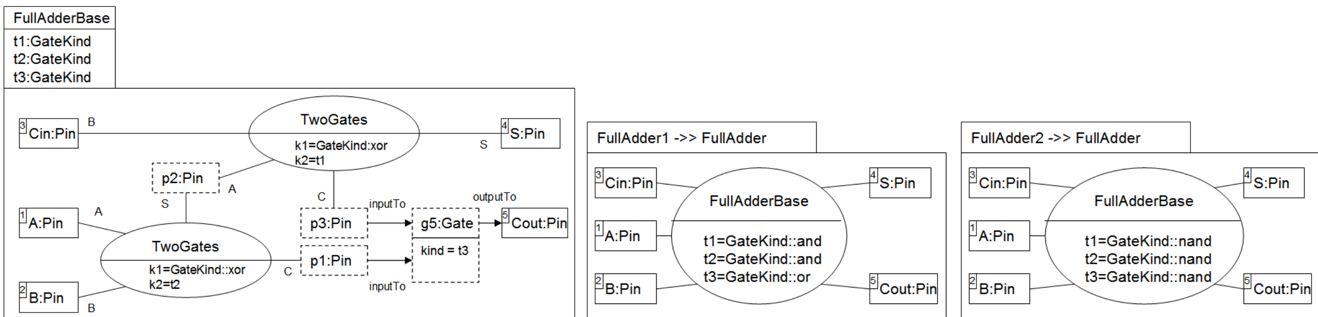

Figure 3-15 VMPL specification of FullAdderBase (right) and its use by FullAdder variants (middle/right)

3.6.4 Summary of Variability Handling

When pattern variability is large (i.e., different conditions, roles and/or interrelationships), the commonality can be specified in a base pattern that is referenced by each variant where unique parts are defined. For example, the

commonality of the *FullAdder* pattern is specified in *FullAdderBase* (Figure 3-10 left), which is referenced by its two variants (Figure 3-11). When the variability is small (e.g., different values for conditions), it can be specified with parameters on a pattern specification that get bound to differently by variants. For example, in an alternative design, the *TwoGates* pattern (Figure 3-14 right) has gate kind parameters that are bound to differently by the two variants of the *FullAdder* pattern (Figure 3-15). When a multi-variant pattern needs to be used generically as a condition of another pattern (i.e., without having to use the individual variants as conditions), the pattern can additionally be implied as a corollary of each variant. For example, the *FullAdder* specification (Figure 3-12 left) is implied as a corollary of each variant (Figure 3-13).

3.7 VPML Tooling

We developed a prototype for a VPML graphical editor on Eclipse. We first defined the VPML metamodel with EMF [15] and automatically generated a Java-based implementation for it. Then, we used the GMF [16] framework to develop an MVC style graphical editor (Figure 3-16). The editor allows us to create and edit VPML models using its graphical notation. We also integrated with some standard Eclipse views like the Property Sheet for specifying property details and Project Explorer for viewing the model in a tree representation. The editor was used to define the patterns of the running example and those of the case studies. While the case study patterns were specified on languages (UML and BPMN) with open-source EMF implementations, the example patterns were specified on CLD, for which we provided an EMF implementation.

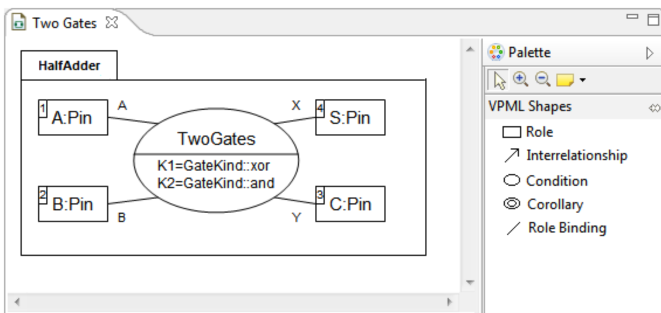

Figure 3-16 Screen shot of VPML editor

4 Design Pattern Detection with QVTR

In Section 3, we showed patterns can be specified formally in VMPL. In this section, we show VPML specifications can be used to automate pattern detection. Figure 4-1 depicts our pattern detection architecture. It is based on mapping a VPML model to a model-to-model transformation. Such a transformation would process an input model and produce an output model. The input model in this case is where pattern occurrences are detected. It conforms to a MOF-based modeling language (e.g., UML, BPMN). On the other hand, the output model is where detected pattern occurrences are reported. It conforms to a new MOF-based DSML, called pattern results (PResults), which we defined for pattern occurrence reporting. In this context, one can consider VPML as a

higher level and more concise façade to specifying patterns than a transformation language. In the remainder of this chapter, we discuss our choice of the transformation language (Section 4.1), describe the PResults DSML (Section 4.2), provide the detailed mapping from VPML to the chosen transformation language (Section 4.3), illustrate the mapping with an example (Section 4.4), and discuss the detection tool we created (Section 4.5).

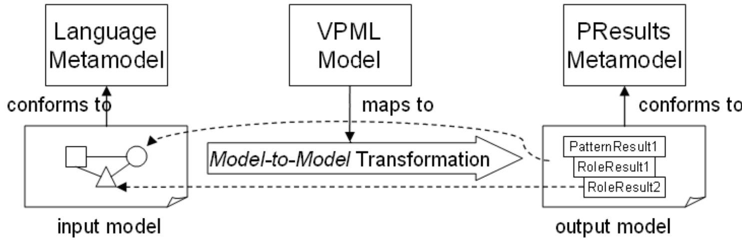

Figure 4-1 Pattern detection architecture

4.1 Selection of a Transformation Language

In order to realize our architecture, we had to select a suitable MOF-based model-to-model transformation language to use for executing the detection. Many such languages exist in the literature today, including QVT-Relations (QVTR), QVT-Operational (QVTO) and ATL. After comparing them, we chose QVTR for a couple of reasons. First, it is declarative, with its rules specified as relations between object templates (resembling patterns) and hence is simple to map to from VPML. In contrast, QVTO has imperative semantics and hence would have needed a more complex mapping. Second, QVTR is a standardized language, which increases its interoperability prospects between conforming tools. This allows creating a generic tool that produces pattern detection transformations, which seamlessly integrate with other modeling tools. In contrast, ATL is not standardized.

However, we note that QVTR is not widely implemented in the industry yet (although some implementations exist like Medini QVT [17], which we use) due to a few ambiguities in the current specification (revision 1.1), as opposed to problems with its soundness. However, we have been collaborating with the QVT revision task force by submitting issues and resolutions, based on our work, that are expected to reduce these issues in future revisions. For example, one of these issues is related to the missing support for contextual properties and operations (i.e., ones that are defined in the context of a metaclass), which is a feature in QVTO but not QVTR. This feature should have also been in QVTR and was simply omitted by mistake. We proposed adding this feature by changing QVTR's abstract syntax (metamodel) and concrete syntax to accept an optional context metaclass for attributes or operations. We also added support for this feature in Medini QVT by making its QVTR interpreter dynamically weave those contextual attributes and operations into the corresponding metaclasses.

4.2 PResults DSML

We propose a new DSML, called Pattern Results (PResults), for reporting detected pattern occurrences in a structured and scalable fashion. The PResults metamodel (Figure 4-2 left) defines class *CatalogResult*, representing the detection results of a specific catalog. Each catalog result owns objects of type *PatternResult*,

representing detected occurrences of a given pattern. Each pattern result owns in turn a tree of objects of type *RoleResult,* representing the bindings of pattern roles to elements playing those roles in the input model.

Since a pattern occurrence is a unique binding of roles to elements, it follows that each branch (from root to leaf) in a *RoleResult* tree represents a unique occurrence of a pattern (e.g., in Figure 4-2 right-a, A1-B2-C1 is a unique occurrence and A1-B2-C2 is another one, both of which share the A1-B2 ancestor branch). This can be contrasted to another representation (Figure 4-2 right-b) often used in the literature [46] where pattern occurrences are tuples of *RoleResult* objects (i.e., no node sharing). A direct advantage of the tree-based representation is being more compact (9 *RoleResult* nodes in the example), due to ancestor node sharing, than the other representation (12 *RoleResult* nodes in the example). Another advantage is that the tree makes the results easier to inspect by users as they can drill down the tree incrementally, uncovering details of the detected occurrences.

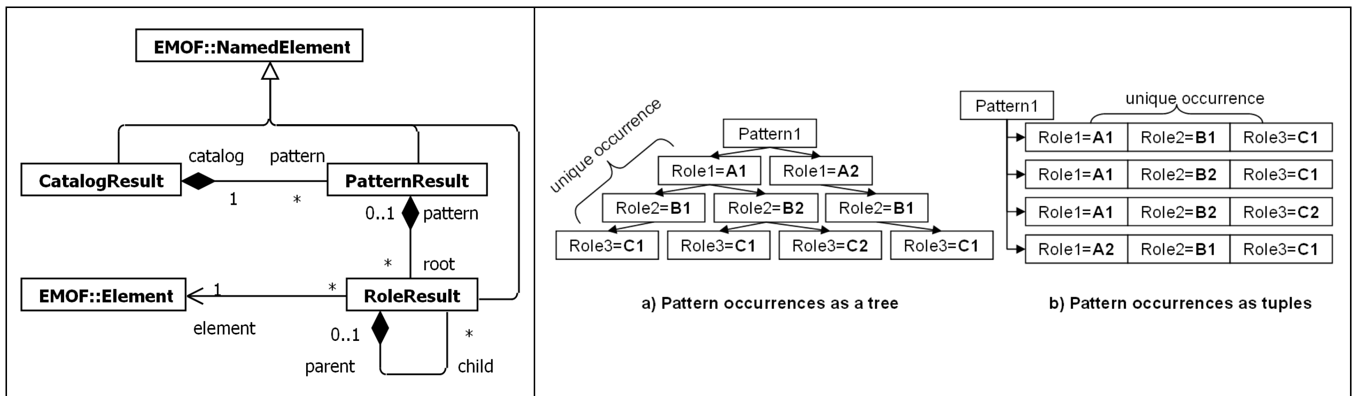

Figure 4-2 PResults metamodel (left) and different representations of pattern occurrences (right)

4.3 Mapping VPML to QVTR

In this subsection, we describe our pattern detection semantics through a mapping from the VPML metamodel to the textual syntax of the QVTR transformation language. We present the mapping incrementally while highlighting the relevant QVTR execution semantics. A summary of the mapping is given in Table 4-1.

4.3.1 Catalog Mapping

Each catalog in a VPML model maps to a QVTR transformation using the (highly abstract) QVTR template shown in Figure 4-3. The catalog's name maps to the transformation's name (line 2). The catalog's referenced metamodels (e.g., UML, BPMN) map to a (comma-separated) list of types for the transformation's *in* model parameter, while the transformation's *out* model parameter is always typed by the PResults metamodel (line 2). The list of imported catalogs maps to the (comma-separated) list of imported transformations (line 1). Similarly, the list of catalog's super-classes maps to the transformation's (comma-separated) extends list (line 3). Note that while a QVTR transformation is multi-directional (i.e., executable in the direction of any of its model parameters, making any of its models a possible input as well as output), the pattern detection transformation is only executed in the direction of the PResults *out* model to report the pattern occurrences detected in the *in* model.

Moreover, each pattern maps to a relation in the QVTR transformation. A relation is a rule that matches elements in an input model and (possibly) updates elements in the designated output model. A relation can be of two kinds: top and non-top. When a transformation executes, all its top relations execute. On the other hand, non-top relations only execute when called transitively by other relations. Based on VPML semantics (Section 3.6.2), a non-*implied* pattern maps to a top relation (line 4), while an *implied* one maps to a non-top relation (line 5).

Furthermore, recall that a catalog may define sets of attributes and operations, which simplify expressions used in the catalog's patterns (Section 3.2). Each catalog's operation maps to a query in the catalog's transformation (line 6) with the same signature. When an operation does not have a body (because it cannot be specified in OCL), it maps to a body-less query (called a *black box* query) and is implemented externally to the transformation in a platform-dependent way. If an operation is contextual to some metaclass, the query is also defined in the same context. Likewise, each catalog's attribute maps to a property of the catalog's transformation with the same signature (line 7) and derivation expression. If an attribute is contextual to some metaclass, the property is also defined in the same context.

```
01  imports <ImportedCatalogNames>;
02  transformation <CatalogName> (in:<CatalogMetamodels>, out:PResults)
03  extends <SuperCatalogNames> {
04    top relation <PatternName> {...}
05    relation <PatternName> {...}
06    query <OperationContext::><OperationSignature><{OperationBody}>
07    property <PropertyContext::><PropertySignature><=PropertyDerivationExpression>;
08  }
```

Figure 4-3 QVTR template for catalogs

4.3.2 Pattern Mapping

In the previous section, we discussed that each pattern maps to a relation in a QVTR transformation. The template for such a relation is shown in Figure 4-4. In QVTR, a relation defines a correspondence between domain variables. Each domain variable represents an object to match in one of the transformation's models (in our case: *in* or *out*). The object is defined with a template consisting of a type, a set of expected property values, and a set of conditions. If a domain variable represents a primitive value (e.g., an integer or a string), it is flagged as *primitive* (line 2). If it represents an element to match in a model only, it is flagged as *checkonly* (line 3). If it represents an element to (also) create/update in a model, it is flagged as *enforce* (line 7). When a relation executes, it tries to bind *checkonly* domain variables to elements from their corresponding input models that conform to their templates. For each unique combination of conforming elements, the relation updates the output model of the *enforce* variables to make them bind to elements conforming to their templates as well.

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

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

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

Figure 4-4 QVTR template for patterns

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

Furthermore, when a pattern has roles to report, they map to an *enforce* domain (line 7). In such a case, the *enforce* domain is defined with a template that creates a pattern occurrence in the *out* PResults model. The template consists of a number of nested local variables. Specifically, variable *c* of type *CatalogResult* (line 7) nests variable *p* of type *PatternResult* (line 8), which recursively nests variables r_1-r_n of type *RoleResult* (lines 9-11) that correspond to the ordered set of *n* reported roles. Each r_n variable is assigned the name of the corresponding reported role and references that role's variable through its *element* attribute.

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

Table 4-1 Summary of the mapping from VPML to QVTR

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

4.3.3 Organizing Pattern Occurrences

The *enforce* domain template in Figure 4-4 (lines 7-11) defines a pattern occurrence as a tree branch in a PResults model. However, for the various branches to form a tree, their elements need to share common ancestors, which need to be uniquely identified. This requires the transformation to look for and update elements in the output model instead of creating identical ones. By default, the notion of element identity is defined by the metamodel. Each metaclass can designate one attribute as uniquely identifying elements, and elements with the same value for that attribute are identical. However, this does not suffice for some of the metaclasses in PResults, since their elements are not uniquely identified by a single attribute. For example, different *PatternResults* can have the same name as long as they are nested under different *CatalogResults*. Similarly, *RoleResults* can have the same *name* and *element* value combinations only when nested in different ancestor elements.

To configure the QVTR transformation to create a tree with unique branches of PResults elements, we need to (i) tell it which combination of attribute values make each metaclass unique, and (ii) include attributes designating ancestors in these combinations. Fortunately, QVTR provides a feature called metaclass *keys*. A *key* is a subset of the metaclass's attributes that uniquely identifies its instances. In this case, we need to define *keys* (Figure 4-5) for

the PResults metaclasses (Figure 4-2 left) that also include their ancestor attributes. This configures the transformation to create new PResult elements only if similar ones do not exist under the same ancestor. Notice that class *RoleResult* has two keys, as it can be nested either by a *PatternResult* (line 3) or a *RoleResult* (line 4).

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

Figure 4-5 QVTR keys for PResults metaclass

4.4 Pattern Mapping Example

This subsection shows an example of mapping VPML to QVTR. Figure 4-6 shows the resulting QVTR transformation of mapping pattern *FullAdder1* (Figure 3-13 right), which references base pattern *FullAdderBase* (Figure 3-10 left) and implies pattern *FullAdder* (Figure 3-12 left). Notice how the *Adders* catalog (Section 3.2) maps to a transformation with the same name (line 1) between an *in* (CLD) model and an *out* (PResults) model.

The *FullAdderBase* pattern (Figure 3-10 left) is mapped to a QVTR top relation with the same name (line 2). The exposed roles of the pattern (i.e., *A, B, Cin, S, Cout, g3-5*) are defined as (*in* model) *checkonly* domain variables with the same names and types (lines 3-10). The interrelationships of those roles map to attribute value for the corresponding domain variables. For example, the *inputTo* interrelationships from role *A* to roles *g1*/g4 are mapped as values *g1/g4* for variable *A*'s *inputTo* attribute (line 3). On the other hand, the pattern's unexposed roles (i.e., g1 and g2) are mapped to local variables in the relation's *when* clause (lines 12-13). Those variables have attribute values corresponding to the roles' attribute values (e.g., *kind=GateKind::xor*) and interrelationships (e.g., *inputTo=g2:Gate {}*).

Similarly, the *FullAdder1* pattern (Figure 3-13 right) is mapped to a top relation with the same name (line 16). The exposed roles of this pattern (i.e., *A, B, Cin, S and Cout*) are mapped to (*in* model) *checkonly* domain variables with no attribute values (lines 17-21). The unexposed roles (i.e., g3-g5) are mapped to local variables in the *when* clause with no attribute values (lines 23-25). Since *FullAdder1* references *FullAdderBase* as a base pattern, its *when* clause has a call to relation *FullAdderBase* with a list of arguments that binds the domain variables of *FullAdderBase* to similarly named variables in *FullAdder1* (line 26). Additionally, since *FullAdder1* references *FullAdder* as a corollary (Figure 3-12 middle), its *where* clause has a call to relation *FullAdder* with a list of arguments binding the domain variables of *FullAdder* to similarly named variables in *FullAdder1* (line 29).

Finally, the *implied* pattern *FullAdder* (Figure 3-12 left) is mapped to a non-top relation with the same name (line 32). The exposed roles of this pattern (i.e., *A, B, Cin, S* and *Cout*) are mapped to (*in* model) *checkonly* domain variables with no attribute values (lines 33-37). Since the pattern reports on those exposed roles, the roles also map to an (*out* model) *enforce* domain variable *c* of type *CatalogResult* (line 38), which recursively nests variable *p* of type *PatternResult* (line 39), and variables *r1-r5* of type *RoleResult* (lines 40-44). Each *ri* variable is assigned the *name* of the corresponding role and references it through its *element* attribute.

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

Figure 4-6 QVTR specification of the FullAdder pattern that corresponds to its VPML specification

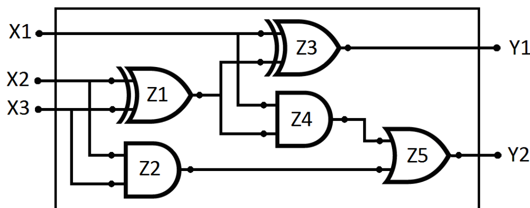

Figure 4-7 Sample CLD circuit model

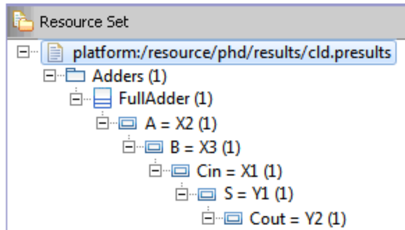

Figure 4-8 Screen shot of the PResults editor

We now describe what happens when the transformation in Figure 4-6 is executed on a sample input CLD model, like the one shown in Figure 4-7. Since the transformation has two top relations (i.e., *FullAdderBase* and *FullAdder1*), the execution engine decides which relation to check first. In this case, since *FullAdder1* uses *FullAdderBase* as a condition (line 26), the latter is checked first. Relation *FullAdderBase* has five Pin variables (A, B, S, Cin, Cout) and five Gate variables (g1-g5), while the input model has 20 pins (the black dots) and five gates. Even though the search space is large (20!/15! for pins and 5! for gates), the execution engine identifies the valid sets of variable bindings (i.e., those that satisfy all the relation's conditions) efficiently (refer to Section 4.5 for details). Since *FullAdderBase* has all the circuit's connection conditions (e.g., pin variable A is connected as input to gate variable g1, line 3), the engine finds only a single valid set of variable bindings: X2 for A, X3 for B, X1 for Cin, Y1 for S, Y2 for Cout, Z4 for g3, Z3 for g4, and Z5 for g5.

Once the engine is done checking *FullAdderBase*, it then checks *FullAdder1*, which has the same domains as *FullAdderBase*. Since *FullAdderBase* is a condition of *FullAdder1* (line 26) and their domains are bound to each other, the execution engine efficiently narrows down the candidate sets of variable bindings to those previously obtained for *FullAdderBase* (in this case, a single set of bindings). The engine additionally verifies *FullAdder1's* own conditions (e.g., on the *kind* attribute of variable g3, lines 23) and finds them to hold for the identified set of bindings: for instance, Z4 (the value of g3) is indeed an AND gate. After that, the engine turns to the *where* clause of *FullAdder1* and checks relation *FullAdder* (line 29), whose domains are bound to ones in *FullAdder1*. Since *FullAdder* does not add any new condition, the same set of bindings from *FullAdder1* applies to this relation as well. The engine then enforces the output domain of *FullAdder* (lines 38-45), which leads to creating a PResults pattern occurrence corresponding to that valid set of bindings (Figure 4-8).

4.5 Detection Tooling

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

We used EMF to define the PResults metamodel and generate a corresponding Java-based implementation for it. Then, we developed a tree-based editor for PResults models, shown in Figure 4-8 with the detection results of Section 4.4. We made the nodes in the tree show the number of pattern occurrences below them since we believe this would facilitate result inspection. For instance, it can help a user inspecting the detection results (by drilling down the tree) focus on the branches with the most occurrences.

We also developed a VPML to QVTR translator using JET [47], which is an EMF-based model-to-text transformation tool on Eclipse, thereby making the generation of QVTR transformations from VPML specifications automated. (This is how Figure 4-6 was created.) JET provides a declarative way of specifying

model-to-text transformations and was chosen due to our previous familiarity with it. Alternative frameworks on Eclipse, like Acceleo [48], could have also been used.

We used the tool Medini QVT [17] to inspect, execute and debug the resulting QVTR transformations. Early executions pointed to a scalability problem when analyzing large models. We investigated and identified two issues with the Medini implementation: 1) repeated executions of the transformation's queries, and 2) a slow variable binding algorithm for a relation. We improved Medini to address both issues (we plan to submit the improvements to the Medini project at some point). We addressed the first issue by caching the results of query evaluations, using their arguments for identification, since such results do not change (the input model where we look for pattern occurrences does not change). Therefore, before performing a new evaluation, the cache is inspected for a previous result. We addressed the second issue by improving the variable binding algorithm. The problems included the way the variable binding tree was constructed, and the order in which conditions on the variables where checked along the tree branches. Since QVTR is declarative, the order in which variables and conditions are specified should be not matter. Instead, an execution algorithm should analyze which variable is used by which condition and decide how to setup the binding tree to prune its branches as early as possible.

The original algorithm in Medini configured the variable binding tree without such an analysis and variable binding was purely based on the syntactical order of variables and conditions in a relation. For example, all variables of relation *FullAdderBase* (Figure 4-6, lines 2-15) were bound at the root of the tree (i.e., their candidate values included all elements of their types leading to a potentially huge search space), overlooking the fact that some of them, like variable *g1* and *g4* (lines 3-4), were bound as attribute values of other variables, which meant they could have had a more restricted set of candidates. Another example is the condition on the *kind* attribute of variable *g3* (line 13) that was checked towards the bottom of the tree (as it appeared towards the end of the relation) and not directly after variable *g2* was bound (line 5), which could have helped prune the tree earlier.

Our improved algorithm addresses the scalability of variable binding with three main enhancements. First, it analyzes the interrelationships between variables and constructs a depth-first search tree where less dependent variables are bound earlier in the tree than more dependent variables. For example, this makes variables A, B and C_{in} of relation *FullAdderBase* pop to the root of their tree as they are not at all dependent on other variables. Having a small number of root variables to bind can dramatically improve the performance of a binding tree as this reduces the number of initial candidate bindings. Second, the algorithm analyzes the interdependencies between variables and conditions, and uses that information to check the conditions directly after all their dependent variables are bound in the tree. For example, the *kind* value attribute condition on variable g2 (line 13) is checked immediately after the variable is bound (line 5). This causes the search space to be effectively pruned as early as possible. Third, we use the relation calls in the *when* clause (e.g., the call to relation *FullAdderBase* at line 26) to reduce the number of candidate bindings early as well. Since these called relations need to be bound

before their calling relations execute, the set of valid variable bindings for them are already known, hence can reduce the candidate bindings of the corresponding variables in calling relations.

5 Case Study 1: Detecting Gang of Four Patterns in UML

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

5.1 Case Study Design

The case study involved analyzing a UML design model by detecting occurrences of the Gang of Four (GoF) [3] design patterns (commonly used in object-oriented design). The analyzed UML model represented a high level design of a large project called the Graphical Modeling Framework (GMF) [16]. GMF was known to have used GoF patterns extensively in its design, making it a good test bed for accuracy. The model (provided in [49]) was defined by a GMF architect who was asked to document the design of the project manually with UML: structure was modeled with class diagrams and behavior was modeled with sequence diagrams and OCL constraints. The model was big enough, with 4 packages, 95 classes, 127 attributes, 440 operations, 109 interactions, 500 messages and 189 OCL constraints, to also be a realistic test bed for performance.

We analyzed the model by detecting occurrences of a representative subset (eleven out of 23) of GoF patterns in three different categories: Creational (Singleton, Abstract Factory and Factory Method), Structural (Adapter, Bridge, Composite and Decorator) and Behavioral (Chain of Responsibility, Command, Observer and Strategy). The choice of this particular subset was motivated by knowledge of its relevance to the project. In addition to documenting the design of the project, the architect was asked to report on occurrences of GoF patterns, including variants other than the official GoF variants: this resulted in a gold standard (GS) of pattern occurrences. The case study was then structured into a matrix of experiments with two variables. The first variable was the level of detail of the model. Four levels were defined: (level 1) class diagrams only, (level 2) class and sequence diagrams, (level 3) class diagrams and OCL constraints, and (level 4) class diagrams, sequence diagrams and OCL

constraints. The second variable was the variants accounted for by the pattern specifications. In one instance, only the official variants were specified. In another, all known variants were specified.

5.2 Evaluating Expressiveness

The chosen subset of GoF patterns were specified in a VPML catalog. It featured a variety of structural and behavioral conditions. Most structural conditions consisted of expected values for roles' properties (attributes and interrelationships). For example, in the AbstractFactory pattern, the Product role was supposed to be played by an abstract class so the role had an attribute value condition [*isAbstract=true*]. The constrained properties were generally defined in the UML metamodel. We had to define a couple of contextual properties in the catalog to represent transitive closures needed to generalize the patterns. For example, we defined contextual property *Class::allSuperClasses:Set(Class)* with a derivation expression of [*self->closure(superClass)*] to get the transitive closure of all ancestor classes of a context class. The property was used, for example in the AbstractFactory pattern, to relate class ConcreteProduct to class Product so they are not only direct sub/super classes. Some structural conditions were more complex and we specified them in OCL. For example, in the Composite pattern, we added a condition [*Leaf.allSuperClasses->excludes(Composite)*] to prevent class Composite from being in the super class chain or class Leaf.

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

Figure 5-1 Specification of the localCall query in QVTR

Behavioral conditions were found to take one of three forms: 1) a call to an operation on the same object, 2) a call to an operation on a different object, and 3) an operation creating objects of some class. These conditions were specified in the catalog with three Boolean operations (*localCall*, *delegationCall* and *objectCreation*, respectively). The operations analyzed two kinds of behavioral information in the UML model: sequence diagrams and OCL constraints. For example, the specification of the *localCall* operation (after being mapped to QVTR) is shown in Figure 5-1 (lines 1-3). Notice how it handles the case (lines 4-8) where a *callee* operation is called transitively by a *caller* operation via intermediate local calls. Sequence diagram analysis is specified in OCL directly over the UML metamodel (lines 14-18). However, OCL constraint analysis is specified with body-

less VPML operations that get mapped to QVTR *black-box* queries (line 19), and implemented in java. This was necessary since OCL constraints are expressed textually (e.g., the body condition '*result = getMax()/2*' of operation *getAverage* expresses a call to local operation *getMax*) and thus cannot be analyzed as OCL models. When behavioral information is not available in the UML model (line 10), the operations returned *true* to preserve recall (at the expense of precision).

Furthermore, the GoF patterns were previously found to use some common idioms [50]. We identified four of them that we considered to be building blocks for the others and specified them in VPML. The first was *Redefinition*, whereby an operation in one class is redefined by a subclass. The second was *Conglomeration*, whereby an operation in one class is locally called from another operation in the same class or a subclass. The third was *Delegation*, whereby an operation in one class is delegated to by an operation in another class using one of its properties. The fourth was *Creation*, whereby an operation creates objects of some class. We then used these idioms as conditions in the GoF specifications to make them more concise. For example, Figure 5-2 (right) shows the VPML specification of the *AbstractFactory* pattern, whose class diagram is shown in Figure 5-2 (left). The specification uses three idioms: *Delegation* from the *Client* class to the *Factory* class, *Redefinition* of the *Factory* class by the *ConcreteFactory* class and *Creation* of the *Product* class by the *ConcreteFactory* class. The three idioms are specified in VMPL in Figure 5-3.

We do not show all eleven GoF specifications here for brevity. However, we make them available online [51]. (Note that these specifications are our own interpretation of the informal GoF pattern descriptions.) We collected some statistics (Table 5-1) to help the reader assess their complexity. One observation is that each specification used 1-3 idioms, in addition to 1-5 other conditions (property values, interrelationships or extra conditions). This shows the ability of VPML to specify complex patterns concisely. One exception was the *Singleton* specification, which had a small number of roles and several unique conditions, hence used no idiom. Some of the specified GoF patterns also had variants. Those variants were specified by capturing their common roles and conditions in base patterns, referenced by each variant, along with a small number of extra conditions. Table 5-1 shows the statistics of the base patterns, separately from the variants, between parentheses.

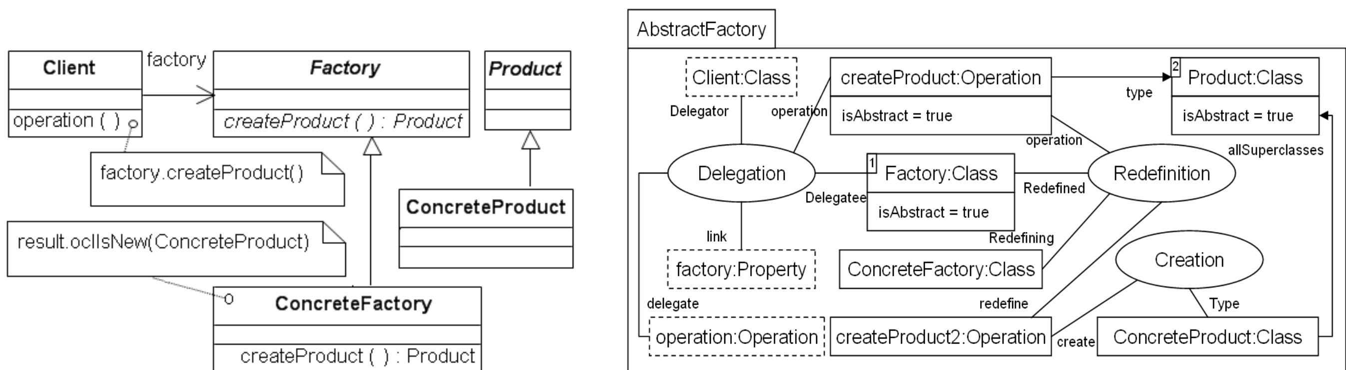

Figure 5-2 AbstractFactory class diagram (left) and its specification in VPML (right)

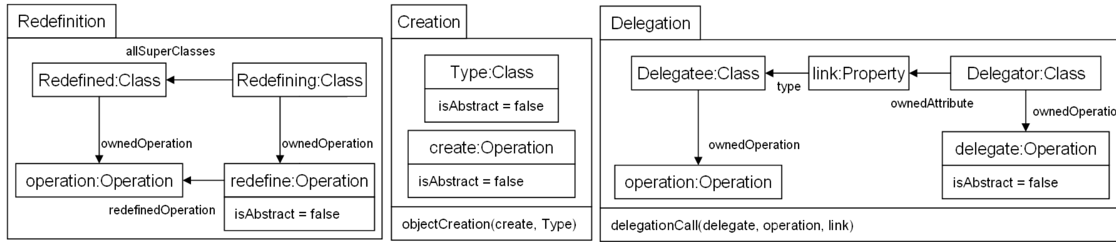

Figure 5-3 VMPL specifications of some GoF idioms (micro-patterns)

Table 5-1 Statistics of the GoF specifications in VPML

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

5.3 Evaluating Accuracy

We carried a set of experiments to assess the accuracy of detecting GoF occurrences using two standard metrics: precision (the ratio of correctly found to all found occurrences) and recall (the ratio of correctly found to all correct occurrences): correct occurrences are defined by the gold standard. Since accuracy calculations depend on the numbers of unique occurrences, which in turn depend on the roles reported for each pattern (i.e., the ones specified using the *Pattern::reportedRole* property, Section 3.3), we developed a criterion for choosing the roles to report on. Specifically, we chose roles that were played by classes that were heads of their inheritance hierarchies or related to them by association. This criterion seemed to select roles (Table 5-1 - fifth column) that sufficed, for someone familiar with the model, to judge the validity of the occurrences without further inspection. For example, for the Adapter pattern, it selected the Target and Adapter classes (since they are at the top of their inheritance hierarchy) and the Adaptee class (since it is related to Adapter by an association in the ObjectAdapter variant), which are the most important roles of the Adapter pattern. The number of unique occurrences in the gold standard, taking into account only the elements bound to those roles, was 45 (the second column in Table 5-2)

The first set of experiments assessed the impact of various levels of model details on accuracy when accounting for the official GoF variants only. Table 5-2 shows the data collected for these experiments. The acronyms used stand for the following: T (total number of occurrences), P (precision) and R (recall). When the model was specified at level 1, detection results had generally low precision (29%). The reason is that several patterns (e.g., Bridge and Strategy) did not have enough distinguishing structural features (unlike Singleton for example that required operation *getInstance* to be *static*). Conversely, for this reason, recall was relatively good

(78%) as a lot of valid occurrences, including those of non-official variants, only differed behaviorally and thus were detected. However, structurally different variants were still not detected. However, non-official variant occurrences that sufficiently differed structurally were not detected: both Singleton and Observer had those, explaining their lower number of occurrences than in the GS and hence their recall.

When the model was at level 2, we observed a radical improvement in precision (68%) as sequence diagrams allowed operation roles to be detected more accurately. However, some patterns (e.g., Adapter and Command) that were not behaviorally unique still detected invalid occurrences. Recall slightly dipped (73%) as some behaviorally different variants were not detected. At model level 3, we observed a dramatic drop in the number of detected occurrences overall, as checking OCL constraints alone was not sufficient to verify all behavioral conditions, especially for roles played by non-query operations (e.g., setter operations). This is because OCL, a side-effect free constraint language, cannot specify calls to such operations. This caused recall to fall to (11%) and precision to have mixed results. Patterns with some occurrences detected (e.g., Singleton and Adapter) had a perfect precision, while others (e.g., AbstractFactory) had their precision undefined as they had no occurrences. At model level 4, results came close to those of level 2 with precision at (70%) and recall at (78%) suggesting that checking both types of behavioral information was not much better than checking sequence diagrams alone.

The second set of experiments assessed the impact of accounting also for non-official variants identified in previous projects. To simulate such situation in the case study, we asked the GMF architect to inspect one of the four packages (Model, Edit, Figure and UI) of the model to find non-official variants. We specified the ones that were found, and then used the specifications when detecting patterns in the rest of the model. We repeated this for every package to remove any bias towards a particular package. The data of these experiments is shown in Table 5-3 in an aggregated form (for all patterns). We observed that precision slightly improved across the board (e.g., 72% at level 4) due to the fact that the additional variants were specified based on their actual application in the model and therefore were more precise. We also observed that recall improved further (e.g., 87% at level 4). This was due to more valid occurrences becoming detectable with the new variant specifications.

Table 5-2 Data of GoF accuracy experiments with official variants only ('U' means undefined due to no occurrences)

Pattern	GS	Level1			Level2			Level3			Level4		
	T	T	P	R	T	P	R	T	P	R	T	P	R
Abstract Factory	2	6	0.33	1.00	5	0.40	1.00	0	U	0.00	5	0.40	1.00
Factory Method	3	7	0.28	0.67	2	1.00	0.67	0	U	0.00	2	1.00	0.67
Singleton	4	2	1.00	0.50	0	U	0.00	2	1.00	0.50	2	1.00	0.50
Adapter	9	33	0.27	1.00	13	0.69	1.00	2	1.00	0.22	13	0.69	1.00
Bridge	5	23	0.17	0.80	8	0.50	0.80	0	U	0.00	8	0.50	0.80
Composite	2	3	0.67	1.00	2	1.00	1.00	1	1.00	0.50	2	1.00	1.00
Decorator	2	2	0.50	0.50	1	1.00	0.50	0	U	0.00	1	1.00	0.50
Chain Of Resp.	1	2	0.50	1.00	1	1.00	1.00	0	U	0.00	1	1.00	1.00
Command	6	27	0.22	1.00	10	0.60	1.00	0	U	0.00	10	0.60	1.00
Observer	5	1	1.00	0.20	1	1.00	0.20	0	U	0.00	1	1.00	0.20
Strategy	6	15	0.33	0.83	5	1.00	0.83	0	U	0.00	5	1.00	0.83
Overall	45	121	0.29	0.78	48	0.68	0.73	5	1.00	0.11	50	0.70	0.78

Table 5-3 Data of the GoF accuracy experiments with all known variants

Inspected Package	GS	Level1			Level2			Level3			Level4		
	T	T	P	R	T	P	R	T	P	R	T	P	R
Model	45	127	0.32	0.91	51	0.70	0.80	7	1.00	0.16	55	0.73	0.89
Figure	45	128	0.39	0.93	52	0.71	0.82	7	1.00	0.16	56	0.73	0.91
Edit	45	127	0.32	0.91	51	0.71	0.80	5	1.00	0.11	55	0.73	0.89
UI	45	122	0.26	0.80	49	0.70	0.76	5	1.00	0.11	51	0.70	0.80
Average	45	126	0.32	0.89	50	0.70	0.79	6	1.00	0.13	54	0.72	0.87

5.4 Evaluating Performance

The experiments of this case study were carried on a laptop with 2.4 GHz core 2 duo processor and 3GB of memory running XP. We measured, using Medini QVT, the average time (of 5 repetitions) taken to detect all patterns in each experiment excluding the time to load/unload the model. We observed that the performance of detection was slowest (~35s) at model level 1. At the other levels, performance was better due to behavioral condition checking effectively pruning the search space. Level 2 was faster (~11s) than level 3 (~16s) as the model had much less sequence diagrams to analyze than OCL constraints. When other variants were considered, detection slowed (~15% on average across levels) due to checking extra conditions. We also observed that using idioms as conditions generally enhanced performance by (~30%) as it led to pruning the search space earlier.

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

5.5 Case Study Summary

The case study shows that our approach is applicable to the problem of detecting design patterns in MOF-based models. Specifically, the approach is adequate for concisely specifying the complex GoF family of design patterns with their structural conditions, behavioral conditions and variants. It also shows that the approach is capable of detecting pattern occurrences with high accuracy, balancing both precision and recall. Best results (precision of 72% and recall of 87%) were achieved when a UML model contained appropriate details (class and sequence diagrams but not necessarily OCL constraints) and when known variants had been accounted for (suggesting it is good to document the use of variants in one project to detect them in future ones). Results also suggest that the performance of the approach is reasonable relative to the realistic size of the analyzed model and the specified patterns. The best performance was achieved when enough details existed in the model and when many pattern specifications used common idioms as conditions.

6 Case Study 2: Detecting Control Flow Patterns in BPMN

The applicability of our approach was assessed in a second case study that involved the detection of a family of design patterns for a DSML (vs. UML). The assessment criteria (expressiveness, accuracy and performance) we used in the first case study (Section 5) were also used in the second. However, the accuracy and performance experiments were simpler than in the first case study (i.e., we did not vary the model details or the considered variants). The main objective of this second case study was to confirm that our approach (of specifying design patterns with VPML and detecting their occurrences in models with QVTR) was indeed generic and could also be used to specify and detect design patterns of DSMLs with high accuracy and performance.

6.1 Case Study Design

The case study involved analyzing an industrial BPMN model by detecting occurrences of the Control Flow (CF) [5] design patterns. These patterns are commonly used in business process modeling with BPMN [52], [53]. The analyzed BPMN model was designed by IBM as a prepackaged solution to the financial services industry. The model, defined in RSA 8.0, included 91 process diagrams, defining common financial service processes, and consisting of 500 activities, 274 gateways, 330 events and 550 sequence flows.

We analyzed the model by detecting occurrences of a representative subset (10 out of 20) of the original CF patterns (some with multiple official variants) in three different categories: Basic Control Flow (Sequence, Parallel Split, Synchronization and Exclusive Choice), Advanced Branching and Synchronization (Multi Choice, Synchronizing Merge, Multi Merge and Discriminator) and Structural (Arbitrary Cycle and Implicit Termination). The choice of these specific patterns was based on what was expressible using RSA 8.0. The gold standard in this case was populated with occurrences that were discovered by manual inspection of the model complemented by input from the subject matter experts who defined the model.

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

6.2 Evaluating Expressiveness

When specifying CF patterns, we observed that their official descriptions did not identify pattern roles thus we identified them ourselves. We found that most CF patterns represented chains of activities connected by sequence flows. Therefore, activities that started or ended chains were typically specified as exposed roles (e.g., the *start* and *end* activities in the Sequence pattern's specification in Figure 6-1 right). A number of CF patterns also incorporated gateways in their chains. Such gateways were specified as non-exposed roles (e.g., the *gateway* in the ExclusiveChoice pattern in Figure 6-2 right). However, sequence flows connecting flow elements (activities and gates) were less important to report on and hence were abstracted out using derived properties in the CF catalog (e.g., property *nextFlowNode*, shown in Figure 6-3 line 9-10, derived flow nodes connected to a context node by outgoing sequence flows). In addition, intermediate activities in transitive chains were also abstracted out with derived properties (e.g., property *nextUncondActivityClosure*, shown in Figure 6-3 line 3-4, derived the transitive closure of all activities connected to context node with unconditional sequence flows).

For brevity, we do not show all CF specifications here. However, they are freely available online [54]. We collected some statistics in Table 6-1 to help assess their complexity. One observation is that the specifications are very concise with 1-4 roles and 0-3 (all structural) conditions. Some patterns also had variants. For example, the Parallel Split (Figure 6-4) was specified as a *before* activity splitting into many *after* activities in one of two ways: 1) using a parallel gateway, or 2) using unconditional sequence flows. Some patterns also used idioms. For example, Discriminator (Figure 6-5, right) used a parameterized N-out-of-M-Join idiom (Figure 6-5, left) with the N parameter set to 1. The idiom specified the merge of M divergent branches into a subset of N branches.

Table 6-1 Statistics of the CF specifications in VPML

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

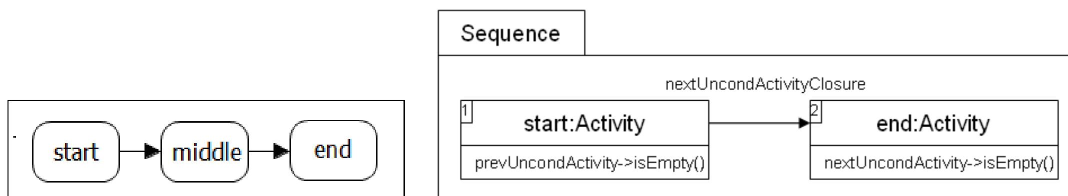

Figure 6-1 Example of the Sequence pattern (left) and its VPML pattern specification (right)

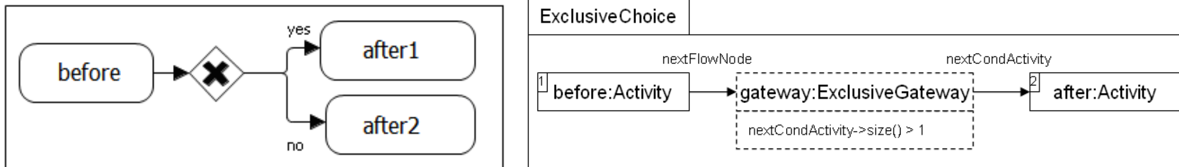

Figure 6-2 Example of the Exclusive Choice pattern (left) and its VPML pattern specification (right)

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

Figure 6-3 Specification of some derived contextual properties defined in the CF catalog

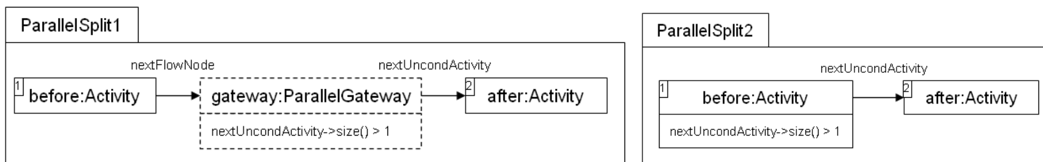

Figure 6-4 Specifications of the two variants of the Parallel Split pattern

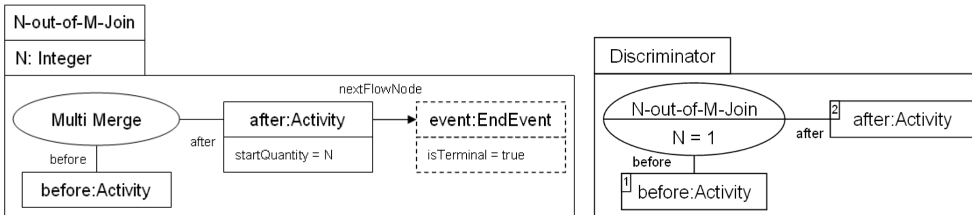

Figure 6-5 Specification of N-out-of-M-Join idiom (left) and its usage by the Discriminator specification (right)

6.3 Evaluating Accuracy

We carried an experiment to assess the accuracy of detecting CF occurrences using precision and recall as metrics. The roles reported for each pattern (third column of Table 6-1) were the activity roles (except for the Implicit Termination pattern where there was no activity), since they were the most recognizable when evaluating a CF occurrence. The number of unique occurrences in the gold standard, taking elements bound to these roles only into account, was 387. Table 6-2 shows the data collected for the experiment. We observed that the overall precision (88%) and recall (94%) values were quite high. This was due to the nature of CF patterns being small and unambiguous, hence easier to detect accurately than GoF patterns in UML. One reason for not achieving a perfect accuracy was that a few sequence flows in the model had their constraints specified in notes written in natural language instead of constraints. This caused them to pass as unconstrained flows, hurting the precision of some patterns looking for unconstrained flows (e.g., Synchronization) and the recall of others looking for constrained flows (e.g., Exclusive Choice). Another reason is the use of few unspecified variants, which affected the recall of some patterns (e.g., Discriminator). Also, some patterns (e.g., Multi Choice) did not have any occurrences causing their precision and recall to be undefined.

Table 6-2 Data of the CF detection experiment

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

6.4 Evaluating Performance

The experiment of this case study was carried on a laptop with 2.4 GHz core 2 duo processor, and 3GB of memory running XP. We measured, using Medini QVT, the average time (of 5 repetitions) taken to detect all patterns in the experiment excluding the time to load/unload the model. We observed that the overall performance of detection (shown in the last column of Table 6-2) was reasonable (~19.5s) given the large size of the model. Patterns with more detected occurrences took longer to finish (~2-3s) than those with little or no occurrences (~1-1.5s) as they needed more processing. Also, more complex patterns that used idioms (e.g., Discriminator) benefited (~0.5s) from detecting their used idioms first, as they helped prune their search space.

6.5 Case Study Summary

The case study shows that our approach is also applicable to detecting a family of design patterns for a domain-specific language. Specifically, the approach is adequate for concisely specifying the CF family of design patterns, on BPMN, with their conditions and variants. It also shows that the approach is capable of detecting pattern occurrences with high accuracy, both for precision (88%) and recall (94%). Results also suggest that the performance of the approach is reasonable (a matter of seconds) given the size of the analyzed model and specified patterns. Additionally, it confirms that the performance of complex patterns with common conditions can be enhanced through the use of common idioms as conditions.

7 Discussion

The case studies in Section 5 and Section 6 show that the proposed approach is applicable to the problem of detecting design pattern occurrences in MOF-based models. (Additional results for these two case studies and results for a third case study on anti-pattern detection in metamodels are available elsewhere [55].) While carrying out the case studies, we gained valuable insights that we discuss in this section.

The first insight is about the benefits and costs of using the proposed approach. Recall that the benefits of analyzing models by detecting design patterns include supporting model maintenance (understanding where design patterns have been applied can help preserve or evolve these applications when a model needs to change) and model comprehension (decomposing models into more abstract and highly cohesive design fragments can aid their understanding). However, the incurred costs of the approach are split between tool developers and language designers. For tool developers, the cost is incurred when developing the required tools. We incurred this cost (at least on the Eclipse platform) by developing prototypes for a VPML editor (using GMF), a VPML detection engine (using JET and Medini QVT) and a PResults viewer (using GMF). However, this cost is marginal considering that such tools can be reused for any modeling language. For language designers (or expert users) who have a deep understanding of a language, the cost is incurred when specifying the known design patterns of a language. This cost is justified by the fact that these patterns are widely used within some communities (industries, organizations or projects) and hence detecting them in models can benefit those communities. In fact, as domain-specific languages get easier to define, language designers may find themselves work on more than one language throughout their career, and hence would appreciate the consistency of the approach.

The second insight we discuss is the relevance and applicability of the approach. While the paper focuses on detecting design patterns, thus implying applicability to design languages only (e.g., UML for software design, and BPMN for process design), the approach is also applicable to non-design DSMLs with recurring patterns in their models (e.g., in a Family Tree DSML, a Twin pattern describes people with the same parents and the same birthdates). Detecting such patterns can help analyze those models. Moreover, it is hard to imagine a modeling language, however small its community (industry, organization or project), not having some recurring patterns that can benefit from our approach (even if such patterns are not officially described in a library due to lack of a formalism). Also, some modeling languages are increasingly used together to specify different aspects of a system (e.g., BPMN for the business side and UML for the technical side of the system) and as such it is not hard to imagine patterns that cross the boundary of a single language. Such patterns will need to be specified using a language-independent approach like ours.

The third insight is realizing that specifying design patterns with VPML is itself a design process. Pattern specifications are incrementally refined until acceptable levels of accuracy and performance are achieved. Such refinement is often performed using sample models having exemplary occurrences of expected pattern variants. Accuracy refinement involves relaxing conditions that are believed to be peripheral to the specifications. This is expected to result in many detected occurrences, which helps improve recall. These occurrences are then analyzed to identify valid ones and decide which conditions to gradually enable back to filter out invalid ones in order to improve precision. This process (also discusses in [7]) is repeated until a good balance between recall and precision is achieved. In addition, performance refinement involves tightening the search space by having as few independent roles as possible and by using any common idioms as conditions.

The fourth insight relates to dealing with the complexity inherent in the target metamodel (e.g., the UML metamodel is notoriously known for being complex). Typically, a pattern designer needs to be intimately familiar with the metamodel details. A good way to abstract these details is to define reusable VPML library catalogs with facilities (e.g. contextual operations, contextual properties and idioms) that abstract out some details. These facilities can greatly simplify pattern specifications making them more readable and maintainable.

8 Limitations and Future Works

Even though we show that our approach is a practical solution to the problem of specifying and detecting design pattern occurrences in MOF-based models, we also highlight some of its current limitations and discuss possible mitigations that will be investigated in future works.

Some limitations exist in VPML. One limitation is the inability of VPML to specify the same design pattern generically for different modeling languages. For example, the CF family of patterns is applicable to both BPMN (process models) and UML (activity models). Currently, VPML requires such patterns to be defined separately for each target language. A possible mitigation is to define a small metamodel representing the pattern domain, specify the patterns on it, and then map it to each target metamodel. Such mapping will be used when generating a QVTR transformation for each target metamodel. Another limitation is the inability of VPML to specify negative conditions graphically, as opposed to using OCL. We plan to enrich the VPML syntax to support this directly.

Furthermore, we showed how the proposed approach can deal with known pattern variants, as discussed in Section 3.6. We also showed, in the first case study (Section 5.3), how to account for unknown variants, which might occur when a pattern is applied differently or incorrectly, by partially inspecting the model looking for the used variants and then detecting them in the rest of the model (this led to a recall of 87%, which was quite high). Nevertheless, we plan to look for ways to achieve high recall with less manual work. One idea is to assign a confidence ratio to each condition in the pattern and report all candidate occurrences with their confidence scores.

Finally, even though we assessed the approach in two large case studies, more case studies are needed in the future on different languages and families of patterns. One particularly interesting case study would be to detect patterns of profile-based DSMLs. Recall that VPML allows specifying patterns of DSMLs defined with a MOF metamodel. However, some DSMLs (e.g., SysML [56], MARTE [57]) are rather defined with a UML profile. A profile defines DSML concepts in the form of stereotypes that can apply to UML metaclasses. We expect profile-based patterns to be specified in a similar manner to metamodel-based ones, although that needs to be verified.

9 Conclusion

Design pattern detection is an important technology in the context of model-driven engineering. Detecting occurrences of known design patterns supports the activities of model comprehension and maintenance. Pattern

detection needs to be automated as the process is very resource intensive and tedious (due to the complexity of both models and patterns) if done manually. We present a novel approach to specify design patterns in any MOF-based modeling language. The approach allows for specifying patterns visually with a new DSML called VPML and addressing many common specification concerns including complexity, accuracy and performance. Patterns specified in VPML can be translated automatically to a corresponding standard QVTR transformation that runs on an input model, where elements playing pattern roles are identified, producing a result model where pattern occurrences are reported in a concise and structured fashion (using a newly proposed language called PResults).

We also report on two case studies where the approach was prototyped on the Eclipse platform and used to detect occurrences of two popular families of design patterns in large models of two widely used modeling languages. In the first case study, 11 (out of 23) GoF design patterns were specified on UML (a general purpose modeling language) and detected in a design model (defined with different levels of detail) of a large project containing 45 unique pattern occurrences. The case study showed that the approach is adequate to specify such a complex set of design patterns. It also showed that the approach is capable of yielding high accuracy levels with a precision of 72% and a recall of 87% when the model has an appropriate level of detail and when known variants (from previous projects for instance) have been accounted for. The case study also showed that the approach performs well (less than a minute) given the model size and pattern complexity. In the second case study, 10 (out of 20) original CF design patterns were specified on BPMN (a DSML) and detected in a large BPMN industrial model containing 387 unique pattern occurrences. The case study confirmed that the approach is also adequate to specify these design patterns concisely and detect their occurrences with high accuracy (a precision of 88% and a recall of 94%) and high performance (less than 20s). Both case studies provide results that suggest that our approach is both effective and efficient to automatically detect design patterns in MOF-based models.

Based on the findings of these case studies, we believe that our approach has a high chance of working on a large, if not any, family of patterns and for a large, if not any, MOF-based language. However, more case studies need to be carried out to help further validate these findings. We also believe that VPML has characteristics that make it familiar and easy to use by pattern practitioners. Nevertheless, we plan to validate the usability of VPML rigorously using empirical studies with real users in the future.

References

[1] Model-driven engineering. http://en.wikipedia.org/wiki/Model-driven_engineering

[2] Booch, G.: "Handbook of Software Architecture". http://www.handbookofsoftwarearchitecture.com

[3] Gamma, E., Helm, R., Johnson R., Vlissides, J.: "Design Patterns: Elements of Reusable Object-Oriented Software", Addison.-Wesley, 1995.

[4] Unified Modeling Language (UML), Superstructure v2.4. http://www.omg.org/spec/UML/2.4/Superstructure/PDF

[5] Russell, N., Hofstede, A., Aalst, W., Mulyar, N.: "Workflow Control-Flow Patterns: A Revised View", BPM Center Report BPM-06-22 , BPMcenter.org, 2006.

[6] Business Process Model and Notation (BPMN) v2.0. http://www.omg.org/spec/BPMN/2.0/PDF

[7] Kerievsky, J.: “Refactoring to Patterns”, Addison-Wesley, 2004.

[8] Ambler, S.: “Agile Modeling: Effective Practices for eXtreme Programming and the Unified Process”, John Wiley & Sons, 2002.

[9] Chikofsky, E., Cross, J.: "Reverse engineering and design recovery: A taxonomy", IEEE Software, vol. 7(1), pp. 13-17, 1990.

[10] Elaasar, M., Briand, L., Labiche, Y.: "Domain-Specific Model Verification with QVT", ECMFA'11, LNCS, vol. 6698, pp. 282-298, June 2011.

[11] Meta Object Facility (MOF) Core v2.0. http://www.omg.org/spec/MOF/2.0/

[12] Object Constraint Language (OCL) v2.2. http://www.omg.org/spec/OCL/2.2/

[13] Query/View/Transformation (QVT) v1.1. http://www.omg.org/spec/QVT/1.1/

[14] ATL Transformation Language. http://www.eclipse.org/atl/

[15] Steinberg, D., Budinsky, F., Paternostro M., Merks, E.: “EMF: Eclipse Modeling Framework”, 2nd edition, 2009.

[16] Graphical Modeling Framework (GMF). http://www.eclipse.org/gmf/

[17] Medini QVT. http://projects.ikv.de/qvt

[18] Fulop, L., Ferenc R., Gyimothy, T.: “Towards a Benchmark for Evaluating Design Pattern Miner Tools”, Proceedings of CSMR, pp. 143-152, Apr. 2008.

[19] Dong, J., Zhao, Y., Peng, T.: “Architecture and design pattern discovery techniques-a review”, Software Engineering Research and Practice, pp. 621–627, 2007.

[20] Birkner, M.: "Objected-Oriented Design Pattern Detection Using Static and Dynamic Analysis in Java Software", Master Thesis, Univ. of Applied Sciences Bonn-Rhein-Sieg, Germany, August 2007.

[21] Huang, H., Zhang, S., Cao, J., Duan, Y.: “A practical pattern recovery approach based on both structural and behavioral analysis”, Journal of System Software, vol. 75 (1-2), pp. 69–87, 2005.

[22] Mikkonen, T.: “Formalizing design patterns”, Proceedings of International Conference on Software Engineering, pp. 115–124, 1998.

[23] Taibi, T., Check, D., Ngo, L.: “Formal specification of design patterns-a balanced approach”, Journal of Object Technology, vol. 2 (4), pp. 127-140, 2003.

[24] Eden, A. Hirshfeld, Y., Lundqvist, K.: “LePUS - Symbolic Logic Modeling of Object Oriented Architectures: A Case Study”, Proceedings of Second Nordic Workshop on Software Architecture, Univ. of Karlskrona, 1999.

[25] Beyer, D., Noack, A., Lewerentz, C.: “Efficient relational calculation for software analysis”, IEEE Transactions on Software Engineering, vol. 31 (2), pp. 137–149, 2005.

[26] Dijkmana, R., Dumasb M., Ouyangc, C.: "Semantics and analysis of business process models in BPMN", Information and Software Technology, vol. 50 (12), pp. 1281-1294, November 2008.

[27] Seemann, J., Gudenberg, J.: “Pattern-based design recovery of Java software”, Proceedings of the 6th International Symposium on Foundations of Software Engineering, pp. 10–16, 1998.

[28] Eppstein, D.: “Subgraph Isomorphism in Planar Graphs and Related Problems”, Proceedings of Sixth Ann. Symposium on Discrete Algorithms, pp. 632-640, Jan. 1995.

[29] Pettersson, N., Lowe, W.: “Efficient and Accurate Software Pattern Detection”, Proceedings of 13th Asia Pacific Software Engineering Conference, pp. 317-326, Dec. 2006.

[30] Rudolf, M.: “Utilizing Constraint Satisfaction Techniques for Efficient Graph Pattern Matching”, Proceedings of 6th Int’l Workshop on Theory and Application of Graph Transformations, LNCS, vol. 1764, pp. 238—251, London, 1998.

[31] Dong, J., Lad D, Zhao, Y.: “DP-Miner: Design Pattern Discovery Using Matrix”, Proceedings of 14th IEEE International Conference on Engineering of Computer-Based Systems, pp. 371-380, Mar. 2007.

[32] Tsantalis, N., Chatzigeorgiou, A., Stephanides G., Halkidis, S.: “Design Pattern Detection Using Similarity Scoring”, IEEE Transaction on Software Engineering, vol. 32 (11), pp. 896-909, November 2006.

[33] Gueheneuc, Y., Sahraoui, H., Zaidi, F.: “Fingerprinting design patterns”, Proceedings of the 11th Working Conference on Reverse Engineering, pp. 172-181, 2004.

[34] Le Guennec, A., Sunye, G., Jezequel, J.: “Precise modeling of design patterns”, Proceedings of UML 2000, LNCS, vol.1939, pp. 482–496. 2000.

[35] Mak, J., Choy, C., Lun, D.: “Precise modeling of design patterns in UML”, Proceedings of International Conference of Software Engineering, pp. 252–261, 2004.

[36] Milicev, D.: “Model-Driven Development with Executable UML”, Wiley, 2009.

[37] Elaasar, M., Briand, L., Labiche, Y.: “A Metamodeling Approach to Pattern Specification”, Model Driven Engineering Languages and Systems, LNCS, vol. 4199 pp.484-498, 2006.

[38] Kim, D.-K., Shen, W.: “Evaluating pattern conformance of UML models: a divide-and-conquer approach and case studies”, Software Quality Journal, vol. 16 (3), pp. 329–359, 2008.

[39] Maplesden, D., Hosking, J., Grundy, J.: “Design pattern modeling and instantiation using DPML”, Proceedings of TOOLS Pacific, pp. 3–11, 2002.

[40] Gueheneuc, Y.-G., Antoniol, G.: "DeMIMA: A Multilayered Approach for Design Pattern Identification", IEEE Transactions on Software Engineering, vol. 34 (5), pp. 667 - 684, 2008.

[41] Bayley, I., Zhu, H.: "Formal Specification of the Variants and Behavioral Features of Design Patterns", Journal of System and Software volume, vol. 83 (2), pp. 209-22, 2010.

[42] Milicev, D.: “Automatic Model Transformations Using Extended UML Object Diagrams in Modeling Environments”, IEEE Transactions on Software Engineering, vol. 28 (4), pp. 413-431, Aprl 2002.

[43] Varro, G., Friedl, K., Varro, D.: "Adaptive Graph Pattern Matching for Model Transformations using Model-sensitive Search Plans", Proceedings of GraMot 2005, vol. 152, pp. 191-205, 2005.

[44] Fujaba Tool Suite RE. http://www2.cs.uni-paderborn.de/cs/ag-schaefer/Lehre/PG/FUJABA/projects/reengineering/index.html

[45] Wikipedia: Adder. http://en.wikipedia.org/wiki/Adder_(electronics)

[46] Pettersson, N., Lowe, W., Nive, J.: “Evaluation of Accuracy in Design Pattern Occurrence Detection”, IEEE Transactions on Software Engineering, vol. 36 (4), pp. 575-590, 2009.

[47] Model to Text (M2T). http://www.eclipse.org/m2t/

[48] Acceleo: transforming models into code. http://www.eclipse.org/acceleo/

[49] GMF Design Model. https://sites.google.com/site/designpatterndetection/gmf-design-model

[50] Smith, J.: “SPQR: Formal Foundations and Practical Support for the Automated Detection of Design Patterns from Source Code", PhD Dissertation, Computer Science Dept., Univ. of North Carolina at Chapel Hill, Dec, 2005.

[51] GoF Specifications in VPML. https://sites.google.com/site/designpatterndetection/gof-specifications

[52] Atwood, D.: “BPM Process Patterns: Repeatable Design for BPM Process Models”. BPTrends, May 2006 http://www.bptrends.com/publicationfiles/05-06-WP-BPMProcessPatterns-Atwood1.pdf

[53] Gschwind, T., Koehler, J., Wong, J.: “Applying Patterns During Busness Process Modeling”, Proceedings of the 6th Int’l Conference on Business Process Management, LNCS, vol. 5240, pp. 4-19, 2008.

[54] CF Specifications in VPML. https://sites.google.com/site/designpatterndetection/cf-specifications

[55] Elaasar, M., Briand, L., Labiche, Y.: "An Approach to the Specification and Detection of MOF-Based Modeling Languages", Ph.D. Thesis, School of Systems and Computer Engineering, Carleton University, 2012.

[56] Systems Modeling Language (SysML), v1.2. http://www.omg.org/spec/SysML/1.2/

[57] MARTE: Modeling and Analysis of Real-Time Embedded Systems v1.0. http://www.omg.org/spec/MARTE/1.0/