Model-Driven Constraint Engineering

Michael Wahler[1], Jana Koehler[1], and Achim D. Brucker[2]

[1] IBM Zurich Research Laboratory, Saeumerstrasse 4, 8803 Rueschlikon, Switzerland
[wah,koe]@zurich.ibm.com
[2] Information Security, ETH Zurich, 8092 Zurich, Switzerland
brucker@inf.ethz.ch

Abstract. A high level of detail and well-formedness of models have become crucial ingredients in model-driven development. Constraints play a central role in model precision and validity. However, the task of constraint development is time-consuming and error-prone because constraints can be arbitrarily complex in real-world models.
To overcome this problem, we propose a solution that we call *model-driven constraint engineering*. In our solution, we define the notion of computation-independent constraints that are provided in the form of meta-model integrated patterns. The parameterized patterns are transformed into platform-independent or platform-independent constraints by a model transformation. In addition, we show how our approach can be supported by a tool.

1 Introduction

In model-driven engineering, textual constraints are used to express details about a model that are either hard or even impossible to express in a diagrammatic way. For instance, hundreds of constraints are used in the specification of the UML (Unified Modeling Language [24]) meta-model. Constraints stem from different sources: there may be legal restrictions that a system needs to obey; there may be company policies that grant privileges to certain kinds of customers; there may be technical restrictions on a system [8]; there may be security restrictions [20]; and there may be facts that are implied by common sense that cannot be expressed diagrammatically.

While models were solely used for documentation and communication purposes in the past, recent model-centric development approaches such as MDA (Model Driven Architecture [19]) use models as first-class artifacts in the development process. For instance, business process models can be transformed to executable code that is run on process execution engines [17] or models in a domain-specific security language are transformed to UML [7]. To guarantee the correctness of the execution of the generated code, it is crucial that every model instance conforms to its defining model and satisfies its constraints. These validity checks can be performed automatically if the constraints are formalized. For instance, tools exist that type-check a set of OCL (Object Constraint Language [23]) constraints and validate a model against them [3]. Alternatively,

B. Demuth, D. Chiorean, M. Gogolla, J. Warmer (Eds.): OCL for (Meta-)Models in Multiple Application Domains, TUD-FI06-04-Sept. 2006, University Dresden, Germany, pp. 111–125, 2006.

validity checks can be implemented in a programming language, e.g., Java, using a model access API, e.g., EMF (Eclipse Modeling Framework [14]).

The creation and maintenance of constraints is a tedious task. In a case study in the business modeling environment that we performed, about 80 constraints were necessary to guarantee the executability of a behavioral model for business process monitoring. All the constraints are invariants on the model elements and restrict the set of allowed model instances to a set that is executable on a process execution engine. While some of these constraints were rather simple, many complex constraints needed to be formalized, which turned out to be a time-consuming and error-prone task. The formalization resulted in approximately 500 lines of OCL code, which by nature are unlikely to be bug-free.

Even when the constraint expressions or validation code do not contain any errors, they need to be adapted once the model changes. This usually results in additional time-consuming coding and debugging phases, especially in model refactorings [11,21] where models undergo frequent changes and the attached constraints need to be kept consistent with new versions of the model.

Our contribution to solving the problem of constraint development consists of three parts. Firstly, we introduce the notion of computation-independent constraints and transformations to platform-independent constraints. Secondly, we introduce constraint patterns and separate the patterns into atomic and composite patterns and add a structure to them to enhance their expressiveness and usability. Thirdly, we discuss the requirements for tool support and illustrate our prototype for Eclipse/UML2 [13].

We believe that a flexible pattern-based approach that is supported by a tool offers an important improvement for constraint engineering. Most syntactic and semantic errors can be avoided because the developer can generate OCL code instead of writing it by hand. Furthermore, our solution promises to decrease development time substantially.

2 Background

Our solution is based on the idea of constraint patterns (sometimes called *idioms*) for UML models [1,2]. A constraint pattern is a parameterized formula that can be instantiated to a constraint by providing values for its parameters. In [1], only two structural constraint patterns—*Semantic Key Attribute* and *Invariant for an Attribute Value of a Class*—are presented, which we consider too little to have a relevant impact on solving the aforementioned problem.

The semantics of a constraint pattern can be provided in any language, e.g., parameterized OCL templates such as in [1]. This has the advantage that an OCL constraint can be simply instantiated by providing values for the pattern parameters. In our solution that we call *model-driven constraint engineering* we follow the MDA approach [19] that comprises models at different levels of abstraction. We consider a constraint pattern a computation-independent model (CIM) of a constraint. A CIM constraint can be transformed into a platform-independent or platform-specific model (PIM/PSM) by a model transformation.

CIM constraints are integrated into the UML meta-model. This integration is accomplished by using the meta-model representation of model elements as parameters for a constraint instead of their textual representation. The types of these parameters are thus elements from the UML meta-model and can be both object types, e.g., *Property*, or simple types, e.g., *String*. The PIM constraint, namely the constraint in a concrete syntax of the constraint language—in our case, OCL—is automatically generated from model-integrated constraint using a model transformation.

We illustrate these concepts in Fig. 1. On the left hand side of this figure, a class *Employee* is shown. This class owns a property whose name is *name* and whose type is *String*. Class *Employee* is constrained by *C1*, which has one parameter, *targetAttribute*, which is an association to a UML *Property*. Furthermore, this constraint is stereotyped *UniqueAttributeValue* which denotes a constraint pattern. Thus, *C1* is the CIM of a constraint.

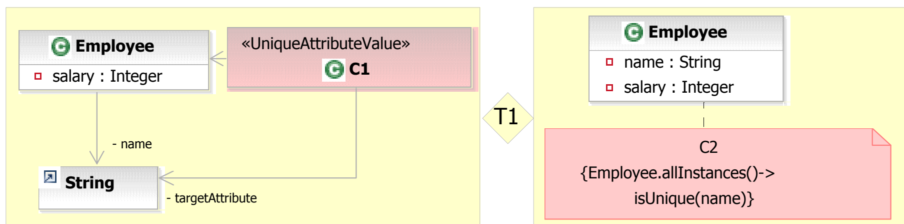

Fig. 1: Transformation from CIM to PIM

The transformation *T1* provides the semantics for the CIM *C1* which is given in this case as a parameterized OCL template. The result can be seen on the right hand side of Fig. 1 where *C2* is the result from transforming *C1* into a platform-independent constraint. By replacing the transformation *T1*, different target platforms can be served. For instance, instead of generating an OCL expression, Java code could be generated that implements a constraint in the Eclipse/EMF [14] framework. In this case, platform-specific knowledge has to be provided in the transformations because our constraint patterns are computation- and platform-independent.

3 Example Model and Constraints

In Fig. 2 we illustrate a simple model that serves as running example. The UML class diagram contains two classes, *Manager* and *Employee*. These classes are related by a many-to-many relation. An instance of *Employee worksFor* at least one manager; a manager *employs* any natural number of employees.

Besides the defined classes and associations, instances of this model are not restricted in any way. There may be managers without employees, and employees may have a salary of zero but work for multiple managers.

Fig. 2: Manager and Employee Class Diagram

We assume fictitious labor union and company IT requirements that every work environment has to satisfy. The requirements are captured in the following constraints informally in English and formally as OCL expressions.

Constraint 1. *A manager must employ at least one employee with a salary of at least 3000.*

This constraint requires that for each instance m of *Manager* there exists an instance e of *Employee* that is related to m by the relation *employs*. Furthermore, the value of the *salary* attribute of e must be at least 3000.

```
context Manager
inv: self.employs->exists( e | e.salary >= 3000)
```

Constraint 2. *A manager may not occur within his management hierarchy.*

This constraint prevents that a manager m is responsible for him-/herself by being related to him-/herself directly by the *worksFor* relation or indirectly by other managers $\{m_i, \ldots, m_j\}$ that work for m. For the corresponding OCL expression, we need to define an operation closureWorksFor(S) that computes the transitive closure [4] of the *worksFor* relation. A parameter S in which elements already processed are stored ensures the termination of this operation.

```
context Manager
def: closureWorksFor(S:Set(Manager)) : Set(Manager) =
    worksFor->union((worksFor - S)->
      collect(m : Manager | m.closureWorksFor(S->including(self)))->asSet())
inv: not self.closureWorksFor(Set{})->includes(self)
```

Constraint 3. *The company may not have more than five organizational layers.*

This constraint restricts the depth of the *worksFor* navigation path. Because a manager can employ another manager, arbitrary hierarchy levels can be realized. However, the fictitious labor union forbids more than five hierarchy levels. A recursive query pathDepth() needs to be defined to compute the path depth. This query has two parameters, *max* and *counter*, where *max* is set to the desired maximum path depth minus 1 and *counter* is initialized with 0.

```
context Manager
def: pathDepth(max:Integer, counter:Integer): Boolean =
    if (counter > max or counter < 0 or max < 0) then false
    else if (self.worksFor->isEmpty()) then true
        else self.worksFor->forAll(m:Manager|m.pathDepth(max, counter+1))
        endif
    endif
inv: self.pathDepth(4,0)
```

Constraint patterns can be identified by analyzing existing constraints and abstracting from them. For example, the constraints introduced in this section can be generalized to the following expressions. Constraint 1 requires the existence of an instance that is related to the context object and has a value restriction on an attribute. Constraint 2 prevents cyclic navigation paths in a model instance. Constraint 3 can be generally seen as a constraint that restricts the maximum length of a navigation path.

From these general expressions, constraint patterns can be derived and described using a schema similar to the one described in [1]. We call the pattern used for Constraint 1 *Exists*, the pattern derived from Constraint 2 *CyclicDependency* and the pattern from Constraint 3 *PathDepthRestriction*. These patterns are part of the taxonomy that we introduce in the following section.

4 A Taxonomy of Structured, Computation-Independent Constraint Patterns

Although the constraint pattern approach as introduced in [1] reduces both the development time and error rate for model constraints, it has one important restriction. As each pattern represents a subset of all possible constraint expressions, even with an extensive pattern library, there will be many constraints that are not expressible in terms of existing constraint patterns.

Therefore, we introduce the notion of *structured constraint patterns* that add a high degree of expressiveness to the existing constraint pattern approach by two measures. Firstly, we divide constraint patterns into *atomic* and *composite* patterns where we introduce a large set of atomic patterns. Composite patterns are recursively constructed from atomic patterns. Secondly, we introduce the logical concepts of implication and negation that allow the applicability of an instance of a constraint pattern to be restricted.

4.1 Atomic Constraint Patterns

In this section we present an extensible library of atomic constraint patterns. The constraint patterns are related with generalization associations. Therefore, we create a *taxonomy* of patterns. The taxonomy gives a structure to the set of patterns and helps one to find the right pattern for a specific purpose.

The idea of atomic constraint patterns is to identify a large set of atomic constraints that restrict fundamental concepts of a model, e.g., attribute values or relations between objects. Furthermore, atomic constraints can be referenced from composite constraints to create a complex constraint from several components. The atomic constraint patterns that we have identified are illustrated in Fig. 3 where we included the two structural patterns from [1] as *UniqueAttributeValue* and *AttributeValueRestriction*. The patterns refer to the UML meta-classes *Class* and *Property* and to the OCL meta-class *OclExpression*.

In MDA, the semantics of a model is inherent in the model transformations that generate PIM constraints from parameterized CIM constraint patterns. In

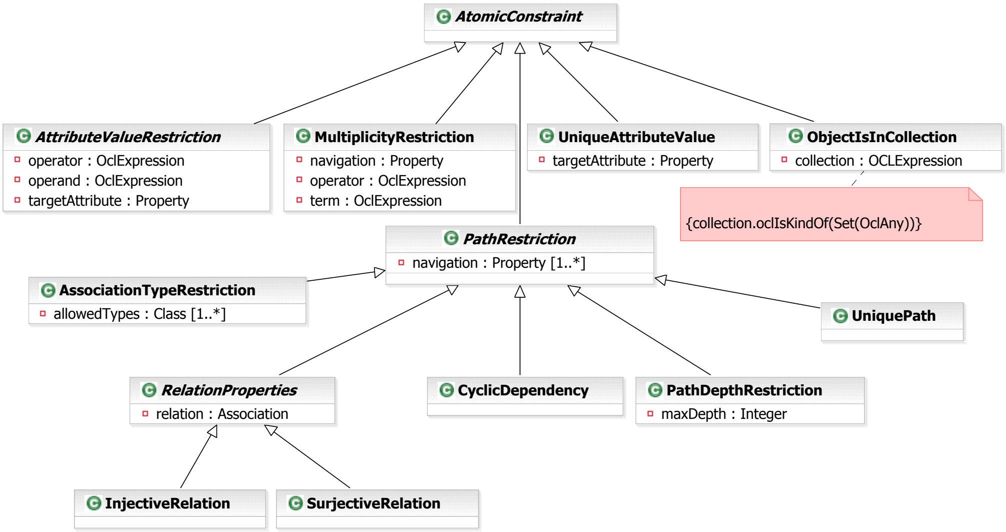

Fig. 3: UML Class Diagram of Atomic Constraint Patterns

the following, we provide informal semantics for the patterns in Fig. 3 and discuss the model transformations in Sect. 5.

Pattern descriptions. The *MultiplicityRestriction* pattern restricts the multiplicity of an association. Although the multiplicity of an association can be restricted in a UML class diagram, this pattern allows for multiplicity restrictions that depend on properties of the model instance, e.g., an attribute value.

Two constraint patterns target at attribute values. The *AttributeValueRestriction* can be used to restrict the value of an attribute of a class for all instances of the class. The *UniqueAttributeValue* pattern requires that all instances of the constrained class have distinct values for the specified target attribute.

The *ObjectIsInCollection* pattern can be used to require that the context element is in the specified collection of objects. For instance, we could require that each manager is in the set of his employees in Fig. 2.

In the lower part of Fig. 3, we show patterns that can be generalized to *PathRestriction* constraints. These patterns restrict properties of a navigation path in a model instance. The *AssociationTypeRestriction* pattern can be used to restrict an association a that is defined on a general class C_0 in a way that in an instance, only certain subclasses $C_1, \ldots, C_n$ of C_0 may participate in the relation that is defined by a.

The *CyclicDependency* pattern can be used to require cycles in the instance graph of the model. Such a cycle can occur if an instance element is related to itself with a certain navigation. This navigation is the only parameter for this constraint pattern. An example for an instance of this pattern is Constraint 2.

The *InjectiveRelation* and *SurjectiveRelation* patterns can be used to establish the mathematical concepts of injective ($f(a) = f(b) \rightarrow a = b$) and surjective ($f : X \rightarrow Y \wedge \text{range}(f) = Y$) relations. Bijective relations can be modeled by constraining an element with a constraint for injectivity and one for surjectivity.

The *UniquePath* pattern can be used to constrain that there may not be more than one path from the context element to a related element. An infamous configuration that can be excluded with this pattern is the "diamond of death" in object-oriented programming languages [22].

The *PathDepthRestriction* pattern can be used to restrict the maximum path length in a model instance for reflexive associations. Constraint 3 from our example is an instance of this pattern where the maximum length of the *employs* association to 5.

4.2 Composite Constraint Patterns

Apart from atomic constraint patterns, which each represent one property of a model element, composite constraints can be used to express complex properties of a model. Therefore, they can integrate an arbitrary number of other constraints (either atomic or composite). Thus, complex constraints can be developed by combining several simple constraints.

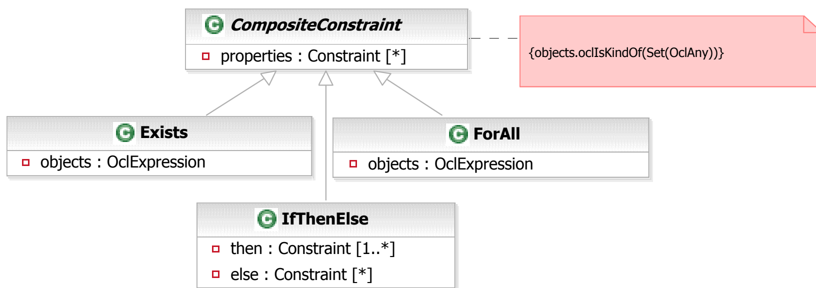

Fig. 4: Class Diagram of Composite Constraint Patterns

So far, we have identified three composite constraint patterns, *Exists*, *ForAll* and *IfThenElse*. Constraint 1 is an example instance of the *Exists* pattern: for the context element m of class *Manager* there has to exist an element e that is related to m with the navigation *employs*. This element e needs to satisfy a number of constraints, the *properties* of the composite constraint. The *ForAll* constraint pattern is similar except that *all* elements in the object collection need to satisfy the properties specified.

The *IfThenElse* pattern realizes an if-then-else expression. If the context element of the constraint satisfies all *properties*, it also needs to satisfy all *then* constraints, otherwise, it needs to satisfy all *else* constraints.

4.3 Adding Logical Structure to Constraint Patterns

We add structure to the concept of constraint patterns by introducing the concepts of negation and implication in a class *StructuredConstraint*, which is a specialization of the UML meta-class *Constraint*. Class *StructuredConstraint* has

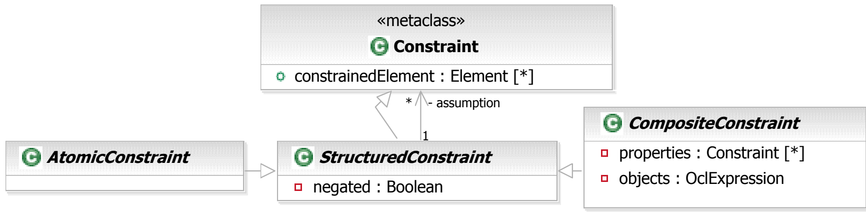

Fig. 5: UML Class Diagram Overview of Structured Constraint Classes

two child classes, namely the previously introduced classes *AtomicConstraint* and *CompositeConstraint*. This idea is illustrated in Fig. 5.

The concept of logical implication is realized as follows. Each structured constraint c can have a finite set A of assumptions that can be any kind of constraint. This is illustrated by the association *assumption* from *StructuredConstraint* to *Constraint*. This allows us to use either arbitrary constraints (defined by a UML *ValueSpecification*) or structured pattern instances as assumptions for constraints. The semantics of the *assumption* relation is defined as follows: Let c be an instance of a structured constraint and A be a finite set of constraints that is related to c with the *assumption* relation. Then the conjunction of all constraints in A implies c. The concept of logical negation is realized by the attribute *negated* of the class *StructuredConstraint*.

5 Transforming CIM to PIM

Having defined a library of CIM constraint patterns, we provide the transformation definitions that are necessary to generate PIM constraints from the parameterized patterns. In this section, we introduce a transformation that generates OCL constraints from parameterized CIM constraint patterns. The transformation transform_OCL(c) uses OCL templates to generate output. We use pseudo code that has the same expressivity as common programming languages for the definition of the operations.

Three steps are necessary to transform an atomic constraint pattern. First, the assumptions need to be generated. Therefore, we define a function transform_assumptions_OCL(c) (Listing 1.1). Then, the OCL keyword not is inserted if the pattern attribute *negated* is *true*. Finally, the variables in the templates for the constraint patterns are replaced by concrete values from the pattern specification. The OCL templates are shown in Table 1.

```
sub transform_assumptions_OCL( c: StructuredConstraint ) {
  # print the conjunction of assumptions
  foreach p in c.assumption
     print (transform_OCL(p) + " and ");

  # print the implication operator;
  # it needs to be preceded by 'true' to generate correct syntax
  print "true implies ";
}
```

Listing 1.1: Transformation Function for assumptions

For conciseness, we do not present a definition of the replace_parameters(t) function. Listing 1.2 shows complete the transformation from CIM to PIM for an atomic pattern.

```
sub transform_OCL(c:AtomicConstraint) {
  # print the assumptions of the constraint
  transform_assumptions_OCL(c);

  # print the OCL keyword 'not' if the constraint is negated
  if (c.negated) print "not ";

  # replace the variables in the template and print constraint
  print replace_parameters(template(c));
}
```

Listing 1.2: OCL Transformation Function for Path Depth Restriction Pattern

<table>
<tr><th>Pattern Name</th><th>Template</th></tr>
<tr><td>PathDepthRestriction</td><td>def: pathDepth(max:Integer, counter:Integer): Boolean =
if (counter > max or counter < 0 or max < 0) then false
else if (self.<<i>navigation</i>>−>isEmpty()) then true
else self.<<i>navigation</i>>.forAll(e|e.pathDepth(max,counter+1))
endif endif
inv: self.pathDepth(<maxDepth>−1,0)</td></tr>
<tr><td>MultiplicityRestriction</td><td>inv: self.<<i>navigation</i>>−>size() <<i>operator</i>> <<i>term</i>></td></tr>
<tr><td>AttributeValue–Restriction</td><td>inv: self.<<i>targetAttribute</i>> <<i>operator</i>> <<i>operand</i>></td></tr>
<tr><td>UniqueAttributeValue</td><td>inv: self.allInstances()−>isUnique(<<i>targetAttribute</i>>)</td></tr>
<tr><td>ObjectIsInCollection</td><td>inv: self.<<i>navigation</i>>−>includes(self)</td></tr>
<tr><td>AssociationType–Restriction</td><td>inv: self.<<i>navigation</i>>−>forAll(x | <<i>allowedTypes</i>>
−>exists(c | x.oclIsTypeOf(c)))</td></tr>
<tr><td>CyclicDependency</td><td>def: closure<<i>navigation</i>>(S:Set(OclAny)) : Set(OclAny) =
<<i>navigation</i>>−>union((<<i>navigation</i>> − S).
closure<<i>navigation</i>>(S−>union(<<i>navigation</i>>))−>asSet())
inv: not self.closure<<i>navigation</i>>(Set{})−>includes(self)</td></tr>
<tr><td>InjectiveRelation</td><td>inv: self.<<i>navigation</i>>.lower = 1 and
self.<<i>navigation</i>>.upper = 1 and
self.allInstances()−>forAll(x1,x2 |
x1.<<i>navigation</i>> = x2.<<i>navigation</i>> implies x1=x2)</td></tr>
<tr><td>SurjectiveRelation</td><td>inv: self.<<i>navigation</i>>.allInstances()−>forAll(y |
y.<<i>relation</i>>−>size() >= 1)</td></tr>
<tr><td>UniquePath</td><td>inv: self.<<i>navigation</i>>−>forAll(x |
self.<<i>navigation</i>>−>count(x)=1)</td></tr>
</table>

Table 1: OCL Templates for Atomic Constraint Patterns

The composite constraints we introduced use other constraints as *properties* for the elements in their object collections. This higher-order use of constraints makes the code generation slightly more complicated than for atomic constraints. A special transformation needs to be written for each composite pattern. For instance, a transformation function for the *Exists* pattern is shown in Listing 1.3. The transformation function for the *ForAll* is similar; only line 6 needs to be adapted. The *IfThenElse* pattern can be transformed analogously.

The OCL generation works as follows. First, the assumptions and the negation are generated if necessary (lines 2,3). Then, the header for the existential quantification over the object collection is generated (line 6) where e is the variable that represents an element of the collection. In lines 9-12, a conjunction of expressions is created from the *properties* of the *Exists* pattern. In this conjunction, every occurrence of the keyword *self* is replaced by the bound variable e. Finally, the conjunction is concluded with the constant *true* and a closing bracket is added (line 15).

```
1  sub transform_OCL( c : Exists ) {
2    transform_assumptions_OCL(c);
3    if (c.negated) print "not ";
4
5    # print the first part of the constraint body and open bracket
6    print "self."+c.objects+"→exists( e | ";
7
8    # print the properties that e needs to satisfy
9    foreach p in c.properties {
10         print transform_OCL(c.properties).replace("self", "e");
11         print "and ";
12   }
13
14   # print a finalizing 'true' and closing bracket
15   print "true )";  }
```

Listing 1.3: Transformation Functions for Composite Constraints

6 Tool Support for Model-Driven Constraint Engineering

Tool support is essential for the acceptance and success of model-driven engineering approaches. In the following, we present how we employ our idea of structured CIM constraint patterns in a model-driven development tool.

As depicted in Fig. 5, our concept of structured constraint is a specialization of the UML meta-class *Constraint*. We suggest an implementation of our approach as UML Profile where each structured constraint pattern is represented by a UML stereotype. The taxonomy of constraint patterns is realized using generalization associations between the stereotypes. The attributes of the constraint patterns become attributes of the stereotypes in the implementation. Here, one deficiency of UML 2.0 becomes critical. In UML 2.0, stereotypes may not have associations with meta-classes [24]. Thus, a *UniqueAttributeValue* constraint cannot refer to the UML meta-class *Property*. Even worse, a composite constraint cannot refer to other constraints that elements need to satisfy. However, this deficiency no longer exists in UML 2.1 [25], where associations between a stereotype and a meta-class may be defined.

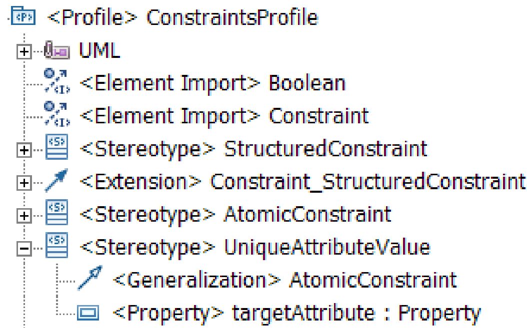

Fig. 6: Screenshot of Eclipse/UML Profile Editor

The Eclipse UML2 project [13] provides an implementation of the UML 2.1 meta-model based on the Eclipse Modeling Framework [14]. This makes Eclipse/UML2 an ideal platform for realizing tool support for structured constraint patterns. In Fig. 6 we show a screenshot of the UML Profile editor in Eclipse. As can be seen, the taxonomy of structured constraint patterns can be implemented in a straight-forward manner.

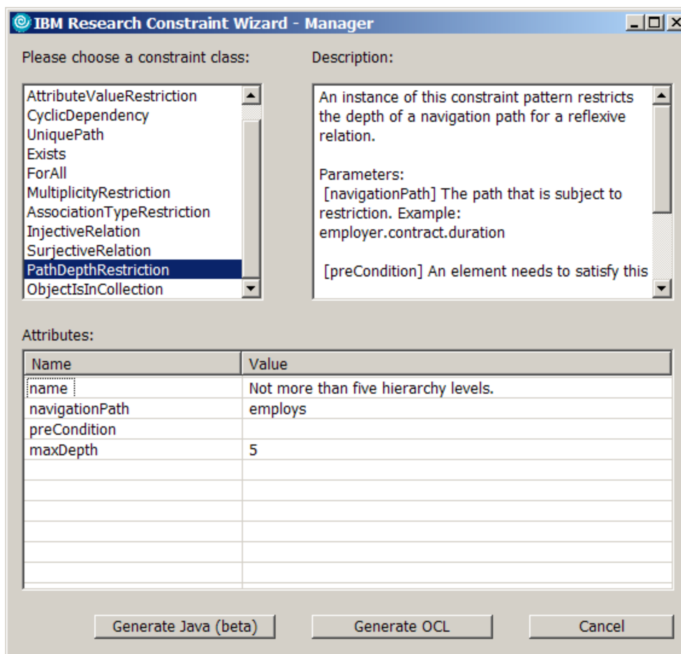

Fig. 7: Screenshot of Constraint Wizard

We prototyped a graphical user interface that guides a user during constraint creation and maintenance. In Fig. 7 we show a screenshot of our “wizard”. In the top left window, the user can choose a constraint pattern. When a pattern is selected, a description of the pattern and its parameters are shown in the top right

part of the window. In the bottom part, the attributes of the selected pattern are shown and values can be entered for them. As can be seen, the wizard implements one CIM-to-PIM transformation that generates OCL expressions and one CIM-to-PSM transformation that creates Java code for run-time model validation. Furthermore, the wizard can also be used to modify previously created structured constraints.

6.1 Applying the Tool to the Example

We have claimed throughout this paper that our approach helps to decrease development time and rate of syntactic errors. To indicate the practicability of our approach, we revisit the example from Sect. 3 and apply our method to it. Using the constraint wizard prototype, we choose appropriate patterns for Constraints 1–3 and specify their parameters.

The result can be seen in Fig. 8. The class *Manager* is constrained by two atomic constraints. An instance of the *PathDepthRestriction* pattern, representing Constraint 3, and an instance of the *CyclicDependency* pattern representing Constraint 2 are attached to *Manager*. Constraint 1 is realized as an instance of the composite pattern *Exists*: among the set of all employees (*self.employs*), at least one element needs to satisfy the *AttributeValueRestriction* property that is attached to the composite constraint.

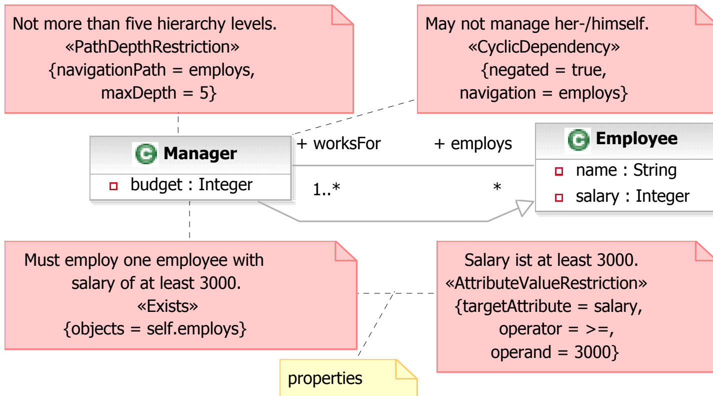

Fig. 8: Example Class Diagram with Structured Constraints Attached

This small example shows the benefits of our approach. The two complicated constraints, Constraint 2 and 3, can be specified by simply providing two parameter values each. If requirements change, these constraints can be quickly adapted without reading, adapting and testing verbose expressions. Constraint 1 is split into two parts, a quantification and a predicate part. This allows for advanced graphical input support that may help users without background in formal languages.

We believe that this small example already shows the practicability of our approach. Complicated recursive expressions are replaced by structured, concise

and easy-to-read constraint definitions. In addition, our model-driven approach enables the automatic generation of platform-independent or platform-specific constraints in various languages or modeling frameworks.

7 Related Work and Conclusion

The difficulty of developing concise and correct OCL constraints has been addressed in numerous publications. OCL is considered to be a very important formalism in today's modeling technologies, still unsolved problems make constraint development difficult [9]. Several solutions have been proposed for dealing with the complexity of constraints and syntactic hurdles. In [10], a set of recommendations is provided to increase correctness, clearness and efficiency of OCL specifications. To simplify the syntax of OCL, a visual concrete syntax for OCL is proposed in [6].

Several publications use the idea of constraint patterns, thus following up the general idea introduced in [15]. Patterns for model-driven development constraints were first mentioned in [5], where one pattern – *Singleton* – is introduced. The idea of constraint patterns is further elaborated on in [1,2], where a small number of constraint patterns are introduced along with OCL templates.

Here, we have introduced the idea of model-driven constraint engineering. Our approach goes beyond existing work in three directions. Firstly, we have introduced the notion of computation-independent patterns and transformations to platform-independent constraints. Secondly, we have introduced a library of patterns that goes far beyond existing pattern solutions in quantity and quality and provided a high degree of expressiveness to this approach by adding logical structure and classifying patterns into atomic and composite patterns. Thirdly, we provide tool support for applying the concepts in a real development environment.

We claim that our approach helps to decrease the time and error rate for constraint development. For instance, the OCL expression that is necessary to express Constraint 3 (Sect. 3) uses a recursive definition that is not easy to understand. In contrast to the lengthy and complicated OCL statement, the same constraint can be defined as an instance of the *PathDepthRestriction* pattern. Appropriate tool support (Fig. 7) further reduces the problem of defining a constraint by pointing-and-clicking to relevant model elements.

The patterns that we present in this paper were elicited from a large set of structural constraints for a model in the business process modeling domain. From the current constraint patterns, almost 90% of the constraints in our case study (cf. Sect. 1) can be instantiated. We believe that more interesting constraint patterns can be identified in other application domains, e.g., model transformations [18], ontology modeling [12] or model refactorings [16]. Therefore, we envision to make the taxonomy publicly available such that any interested user can use the approach and extend the constraint pattern library.

Future work includes the definition of new atomic and composite constraint patterns. However, a pattern-based approach such as the one that we introduce

in this paper is always a tightrope walk between simplicity and completeness with respect to the expressivity of the underlying constraint language. On the one hand, patterns are there to simplify the definition of constraints by providing abstractions for commonly used constraint expressions. On the other hand, given a set of constraint patterns, you can always find a constraint that cannot be expressed as an instance of the available patterns. Adding as many patterns in as much detail as possible to the taxonomy will eventually turn the taxonomy into a meta-model of the OCL language specification. Such a fine granularity would not help with the initial problems of time consumption and error rate.

As a rule of thumb, we discourage the introduction of trivial patterns for two reasons. Firstly, trivial patterns can always be replaced by short OCL expressions. Secondly, a large number of patterns makes it difficult to keep an overview and select the "right" pattern for a specific purpose. For this reason, we discourage the use of the *AttributeValueRestriction* pattern, which we included in this paper for "historic" reasons only. The other patterns will be subject to discussion whether they simplify matters or introduce additional overhead. We believe that further case studies can clarify this issue. Future work also includes the development of constraints for the patterns themselves that deal with problems such as meaningful input ranges or type safety for the values of pattern variables.

We would like to emphasize that although we have introduced a *wizard*, we cannot spirit away the complexity inherent to many constraints. However, we believe that that our approach offers a powerful tool for dealing with this inherent complexity.

Acknowledgements

We would like to thank David Basin, Jochen Küster, Alexander Pretschner and Ksenia Ryndina for their valuable feedback on earlier versions of this paper.

References

1. J. Ackermann. Formal Description of OCL Specification Patterns for Behavioral Specification of Software Components. In T. Baar, editor, *Workshop on Tool Support for OCL and Related Formalisms*, Technical Report LGL-REPORT-2005-001, pages 15–29. EPFL, 2005.
2. W. Ahrendt, T. Baar, B. Beckert, R. Bubel, M. Giese, R. Hähnle, W. Menzel, W. Mostowski, A. Roth, S. Schlager, and P. H. Schmitt. The KeY Tool. *Software and System Modeling*, 4(1):32–54, 2005.
3. J. A. T. Álvarez, V. Requena, and J. L. Fernández. Emerging OCL Tools. *Software and System Modeling*, 2(4):248–261, 2003.
4. T. Baar. The Definition of Transitive Closure with OCL – Limitations and Applications. In A. Ershov, editor, *Perspectives of System Informatics*, 2003.
5. T. Baar, R. Hähnle, T. Sattler, and P. H. Schmitt. Entwurfgesteuerte Erzeugung von OCL-Constraints. *Softwaretechnik-Trends*, 20(3), 2000.
6. P. Bottoni, M. Koch, F. Parisi-Presicce, and G. Taentzer. Consistency Checking and Visualization of OCL Constraints. In A. Evans, S. Kent, and B. Selic, editors, *UML 2000*, volume 1939 of *LNCS*, pages 294–308. Springer, 2000.

7. A. D. Brucker, J. Doser, and B. Wolff. A Model Transformation Semantics and Analysis Methodology for SecureUML. In O. Nierstrasz, J. Whittle, D. Harel, and G. Reggio, editors, *MoDELS 2006*, number 4199 in LNCS, pages 306–320. Springer-Verlag, Genova, 2006.
8. S.-K. Chen, H. Lei, M. Wahler, H. Chang, K. Bhaskaran, and J. Frank. A model driven XML transformation framework for Business Performance Management model creation. In *International Journal of Electronic Business*, volume 4. Inderscience, 2006.
9. D. Chiorean, M. Bortes, and D. Corutiu. Proposals for a Widespread Use of OCL. In T. Baar, editor, *Workshop on Tool Support for OCL and Related Formalisms*, Technical Report LGL-REPORT-2005-001, pages 68–82. EPFL, 2005.
10. D. Chiorean, D. Corutiu, M. Bortes, and I. Chiorean. Good Practices for Creating Correct, Clear and Efficient OCL Specifications. In *NWUML 2004*, 2004.
11. A. L. Correa and C. M. L. Werner. Applying Refactoring Techniques to UML/OCL Models. In T. Baar, A. Strohmeier, A. M. D. Moreira, and S. J. Mellor, editors, *UML*, volume 3273 of *LNCS*, pages 173–187. Springer, 2004.
12. S. Cranefield and M. Purvis. UML as an Ontology Modelling Language. In *IJCAI 99*, 1999.
13. The Eclipse UML2 Project. `http://www.eclipse.org/uml2/`.
14. The Eclipse Modeling Framework. `http://www.eclipse.org/emf`.
15. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, Boston, MA, USA, 1995.
16. M. Gogolla and M. Richters. Expressing UML Class Diagrams Properties with OCL. In *Object Modeling with the OCL, The Rationale behind the Object Constraint Language*, pages 85–114, London, UK, 2002. Springer-Verlag.
17. R. Hauser and J. Koehler. Compiling Process Graphs into Executable Code. In *Third International Conference on Generative Programming and Component Engineering*, volume 3286 of LNCS, pages 317–336. Springer, 2004.
18. R. Hauser, J. Koehler, S. Sendall, and M. Wahler. Declarative Techniques for Model-Driven Business Process Integration. *IBM Systems Journal*, 44(1), 2005.
19. A. Kleppe, J. Warmer, and W. Bast. *MDA Explained. The Model Driven Architecture: Practice and Promise.* Addison-Wesley, 2003.
20. T. Lodderstedt, D. A. Basin, and J. Doser. SecureUML: A UML-Based Modeling Language for Model-Driven Security. In J.-M. Jézéquel, H. Hußmann, and S. Cook, editors, *UML 2002*, volume 2460 of *LNCS*, pages 426–441. Springer, 2002.
21. S. Markovic and T. Baar. Refactoring OCL Annotated UML Class Diagrams. In *MODELS 2005*, volume 3713 of *LNCS*, pages 280–294, 2005.
22. R. C. Martin. Java and C++. A Critical Comparison. Online document. `www.objectmentor.com/resources/articles/javacpp.pdf`, March 1997.
23. Object Management Group (OMG). UML 2.0 OCL Final Adopted Specification. `http://www.omg.org/cgi-bin/apps/doc?ptc/03-10-14.pdf`, 2003.
24. Object Management Group (OMG). Unified Modeling Language: Superstructure. Version 2.0. OMG Document formal/05-07-04, July 2005.
25. Object Management Group (OMG). Unified Modeling Language: Superstructure. Version 2.1. OMG document ptc/06-04-02, April 2006.