OCL Pattern Matching

Tony Clark[1]

Middlesex University, London, UK

Abstract. This paper proposes an extension to OCL that addresses a concern regarding the proliferation of navigation expressions that occur when expressing predicates over objects. Declarative patterns are introduced that can be used to match against object structures so that repeated variables reduce the need for lengthy repeated navigation expressions. Patterns provide the basis for a further contribution that shows how objects can be used as functions.

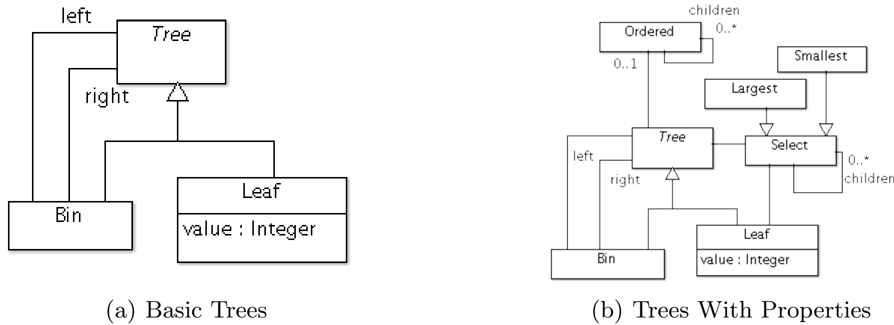
1 Introduction

OCL is a formal language that can be used to express both system invariants and pre and post conditions on system operations. The language is abstract in the sense that it hides away the details of the representation of relationships and by providing collections with associated operations such as `union` and `includes`. Constraints on the state of objects in a system can be defined by applying predicates to path expressions. A path expression navigates from a given instance using '.' to index object properties and association role ends.

Navigation via path expressions occurs frequently in OCL expressions. Typically, a constraint on an object `o` will involve a predicate `p` and two or more path expressions `o.a.b.c` and `o.i.j.k` such that `p(o.a.b.c,o.i.j.k)` is required to hold true. A specification of an invariant or a pre/post condition will require multiple constraints to hold simultaneously. Such a proliferation of constraints involving path expressions can lead to very verbose specifications as noted in [13] (although the author proposes a very different solution to that proposed here). OCL can greatly facilitate the use of models as described in [2] whose authors also note that novices require more training than usual in the use and comprehension of OCL constraints.

Programming languages such as ML [12], Racket [14] and Haskell [6] have addressed the problem of verbose path expressions by providing *patterns*. A pattern is a way of expressing multiple path expressions in a single declarative expression and the OCL Manifesto mentions functional programming languages as part of its motivation [3]. Patterns denote values in a given language; in the case of OCL, patterns denote atomic values, collections and objects. Patterns also contain variables that denote any value of an appropriate type. Therefore patterns denote sets of values, such that any value in the set matches the pattern given a collection of variable bindings. The process of taking a value from the set and constructing the associated set of variable bindings is called *pattern matching*.

This paper proposes an extension to OCL that allows patterns to be used in constraints. The extension involves two new language constructs: a `case`-expression that dispatches on a value given a collection of patterns, and an



(a) Basic Trees (b) Trees With Properties

Fig. 1. Binary Trees

object-expression that is used to construct instances of classes. Together these new constructs allow OCL `context` declarations to be used in new ways. In particular, a special case is identified that allows UML classes to be viewed as *properties* and *functions*. The language is similar to the MT language [15] where the patterns are restricted to model transformations.

2 Motivation

Consider the model shown in figure 1(a)[1]. A binary tree is either a leaf containing an integer value or a pair of trees labelled `left` and `right`.

Suppose that we want to identify ordered binary trees where the leaf values increase left to right. One way to do this is to define an `Ordered` property that holds for a sub-set of trees. A property can be defined as a class that associates 0 or more classes with a class whose instances have the property.

Figure 1(b) shows the class `Ordered` associated with class *Tree* that defines the ordered property on trees. The ordered property uses two auxiliary selection-properties: `Largest` and `Smallest`. The essential idea is that a tree is ordered when all of its sub-trees are ordered and a binary-pair of trees is ordered when the largest element on the left is less or equal to the smallest element on the right.

The OCL specification of ordered trees is shown in figure 2. The specification exhibits features that are typical of using OCL to define properties in this way:
navigation: The specification of `Ordered` and `Largest` both exhibit multiple navigation expressions that are used to apply functions and predicates to values in object slots. Given that the specification of ordered trees is relatively small compared to real-world applications, specifications involving such expressions can become complex and difficult to comprehend.
inheritance: `Tree` is an inductively defined data type: a tree is a leaf or the binary combination of two trees. Such recursive definitions can be achieved using an abstract class `Tree` and concrete sub-classes `Leaf` and `Bin`. A property definition is then defined in the context of the super-class using `oclIsKindOf` as a way of selecting the appropriate sub-class.

[1] Note that in diagrams by default: association ends have a multiplicity of 1 and are named by the attached class (pluralised where necessary). No cycles are allowed.

```
context Ordered inv:
  tree.oclIsKindOf(Leaf) or
   (tree.oclIsKindOf(Bin) and
     children->includes(tree.left.ordered) and
     children->includes(tree.right.ordered) and
     tree.left.largest.leaf.value <= tree.right.smallest.leaf.value)

context Largest inv:
  tree.oclIsKindOf(Leaf) implies leaf=tree and
  tree.oclIsKindOf(Bin) implies
    children->includes(tree.left.select) and
    children->includes(tree.right.select) and
    tree.left.select.oclIsKindOf(Largest) and
    tree.right.select.oclIsKindOf(Largest) and
    leaf.value = tree.left.select.leaf.value.max(tree.right.select.leaf.value)

context Smallest inv:
  -- similar to Largest
```

Fig. 2. Ordered Binary Trees

children: The structure of a property often follows the recursive structure of the data-type it relates to. For example, a binary tree is ordered when the two sub-trees are also ordered. Another example is that the largest element of two binary trees is the maximum of the largest element of the two sub-trees. The model in figure 1(a) shows that the property classes have self-associations called `children` that facilitate the definition of the sympathetic recursion between the property and the class it relates to.

3 Related Work

Pattern matching has been used in functional programming languages for many years. Languages such as ML, Scheme and Haskell use pattern matching in terms of case-analysis and are the basis for the proposal put forward in this paper. A difference is that functional programming languages use algebraic data types rather than classes and inheritance, however there is a straightforward translation from classes with inheritance to tagged elements in an algebraic data type.

Model transformation languages [1, 7] often use patterns to identify the parts of a source model that are to be removed and replaced. For example QVT [9] uses *object patterns* that are similar to the features proposed in this paper. However, by integrating patterns at the OCL level, they can be used to achieve a wide range of useful results including a reduction on the complexity of lengthy navigation expressions, functions and relations. Some languages such as [4] use an SQL-like syntax to match and transform models, however these approaches do not support structural matching.

Graph-based transformation systems use patterns to select parts of a graph and use results from algebra theory to provide a declarative approach to express sharing constraints to be maintained or achieved by transformation rules. Like QVT patterns, these are related to the ideas presented here, but specifically target model transformation.

Other applications of pattern matching in model based engineering involves the use of OCL-based patterns to measure model quality [5], however it appears that the authors do not use structural matching, and that the patterns are encoded using standard OCL. Patterns are described in terms of mining models in [8] but the encode the patterns using standard QVT.

```
context (Ordered)[tree=t] inv:
  case t {
    (Leaf)[value=_] → true;
    (Bin)[left=l;right=r] →
      (Ordered)[tree=l] and (Ordered)[tree=r] and v1 <= v2
        where (Leaf)[value=v1]=(Largest)[tree=l] and
              (Leaf)[value=v2]=(Least)[tree=r]
  }
```

Fig. 3. A Relational Property

4 Patterns, Relations, Functions

Our proposal is that extending OCL with patterns can make specifications more concise and therefore easier to comprehend and analyse. Concretely, our proposition takes the form of two new types of OCL expression: `case`-expressions involving patterns, and object expressions that are used to represent class-instances. Having introduced these features we can use them to extend `context`-definitions to use patterns in relations and to introduce a new form of *functional*-definition.

Figure 3 shows the definition of a relational property called `Ordered`. The body of the constraint is defined by case-analysis on the tree `t`. The use of case analysis separates out the different sub-types in a structured way. Notice how the recursive definition of `Ordered` follows the recursive structure of a binary tree. This means that the reflexive `Ordered::children` association defined in figure 1(b) is implicit in the definition.

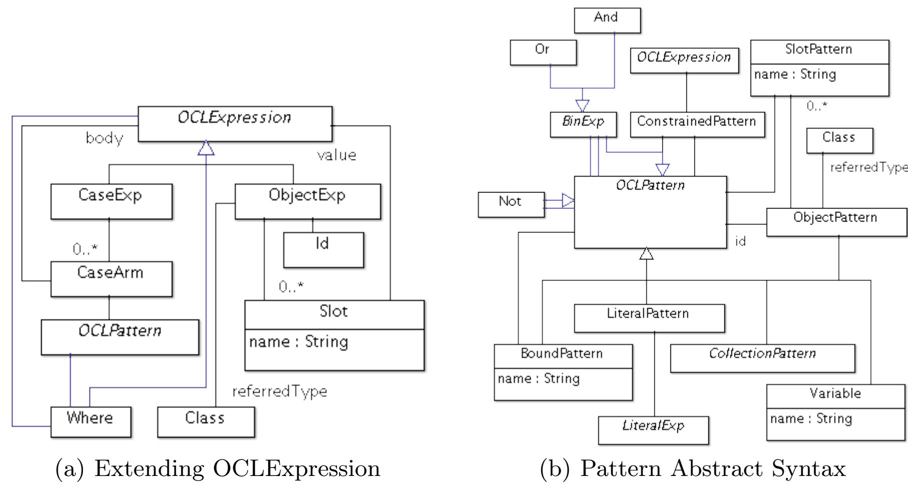
Figure 3 shows the use of a `case`-expression. The basic form is `case v` { `p` → `e; ...` } where `v` and `e` are OCL expressions and `p` is a pattern. If the value `v` matches a pattern `p` then the result of the `case`-expression is given by the corresponding `e`.

Therefore, in figure 3 the relational property `Ordered` is specified by case analysis on the tree `t`. A tree is either a leaf or a binary-tree. A leaf is always ordered. A binary tree is ordered when the sub-trees `l` and `r` are ordered. Object-expressions are used to specify the condition that the sub-trees are ordered: when an object expression such as `(Ordered)[tree=l]` is used in a boolean expression then the corresponding well-formedness constraint, defined using a `context`-pattern definition, must hold.

The `where`-condition is used as a side-condition in figure 3 to require that the greatest element of the left sub-tree is less than or equal to the least element of the right sub-tree. Such a condition uses patterns to *call* an object-expression used as a function call as described below.

```
context (Largest)[tree=t] fun:
  case t {
    (Leaf)[value=_] → t;
    (Bin)[left=l;right=r] →
       (Leaf)[value = v1.max(v2)]
          where (Leaf)[value = (Largest)[tree=l]] and
                (Leaf)[value = (Largest)[tree=r]]
  }
```

Fig. 4. A Functional Property



(a) Extending OCLExpression (b) Pattern Abstract Syntax

Fig. 5. Extending OCL Abstract Syntax

Figure 4 shows the definition of the functional property `Largest`. Such a definition is introduced using the standard `context` keyword, followed by an object pattern and `fun:` that indicates the class that is to be viewed as a function. The slots of the object-pattern define the fields and association ends that are designated as arguments to the function. The single field/association-end that is omitted defines the value that is to be viewed as the result of calling the function.

Again, notice how the `children` association is no longer required because of the recursion in the definition. The function definition is tied to the class `Largest` and therefore `oclIsKindOf` is no longer required, unlike the definition of `Largest` in figure 2.

5 Extending the Standard

The OCL standard[2] defines the language in terms of an abstract syntax, concrete syntax and a semantics. The abstract syntax takes the form of a model that describes how the class `OCLExpression` is used to attach constraints to UML model elements. OCL patterns are defined as an extension to the standard. This section defines patterns and object expressions as sub-classes of `OCLExpression` and provides a concrete syntax as a BNF grammar.

Figure 5(a) shows `OCLExpression` from the standard extended with three new types of expression: `CaseExp`; `Where`; `ObjectExp`. An object expression takes the concrete form `(C,i)[n=e; ...]` where `C` is a reference to a class, `i` is an optional object-identity, `n` is the name of a field or association-end, and `e` is an expression. Therefore, `(Point)[x=10;y=20]` is a two-dimensional point. The optional object-identities are used to specify that the *same* object occurs more than once in an expression (and correspondingly in a pattern as described below). Clearly, there are well-formedness constraints regarding the use of object-identities, however we will omit such discussions here.

[2] `http://www.omg.org/spec/OCL/2.3.1/`

Figure 5(b) shows the definition of patterns as used in `case`-expressions and `where`-expressions. Patterns are used to denote OCL values (including objects) and use variables to match arbitrary components of values. Note that the repeated use of the same variable must always denote the same OCL sub-value when a pattern is matched against a value (modulo identities).

The `ConstrainedPattern` includes the use of a boolean valued `OCLExpression` to add a *guard* to a pattern; this is a way of *escaping* from the restrictions of patterns into arbitrary boolean expressions for example the pattern `p and _ when e` requires that a given value `v` both matches the pattern `p` and satisfies the arbitrary boolean expression `e`.

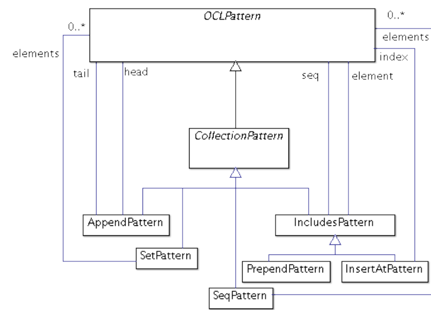


Fig. 6. Sequence Patterns

Patterns may denote collections and it is convenient to allow sequence-patterns to be non-deterministic (via operations such as `append` and `union`) so that they can be further constrained via guards as described above. Figure 6 defines the abstract syntax of collection patterns.

```
OCLExpression ::= ... as defined by the OCL standard ...
              |    CaseExp | ObjectExp | WhereExp
CaseExp ::= 'case' OCLExpression '{' CaseArm* '}'
CaseArm ::= OCLPattern → OCLExpression
ObjectExp ::= '(' Class ')' '[' Slot* ']'
Slot ::= Name '=' OCLExpression
WhereExp ::= OCLExpression 'where' OCLPattern
OCLPattern ::=
  OCLPattern '=' OCLPattern
| OCLPattern 'when' OCLPattern
| LiteralExp
| CollectionPattern
| Var
| '(' Class ')' '[' SlotPattern ']'
SlotPattern ::= Name '=' OCLPattern
CollectionPattern ::= 'Seq{' OCLPattern* '}' | 'Set{' OCLPAttern* '}'
                  |    OCLPattern '->' CollOp '(' OCLPattern* ')'
CollOp ::=  'append' | 'prepend' | 'including' | 'union' | 'insertAt'
```

Fig. 7. Concrete Syntax

Figure 7 proposes a concrete syntax for the pattern language. Notice how the syntax of collection patterns follows that of the equivalent expressions.

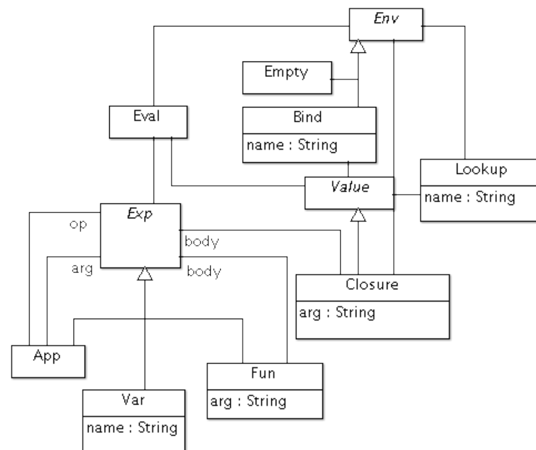


Fig. 8. A Functional Language

```
context (Eval)[exp=e;env=p] fun:
 case e {
  (Var)[name=n] → (Lookup(env=p;name=n];
  (Lambda)[arg=n;body=e] → (Closure)[arg=n;env=p;body=e];
  (App)[op=o;arg=e] →
   let v1:Value = (Eval)[exp=o;env=p]
       v2:Value = (Eval)[exp=e;env=p]
   in case v1 {
       (Clo)[arg=n;env=p;body=e] →
        let p:Env=(Bind)[name=n;value=v2;env=p] in (Eval)[exp=e;env=p]
      }
 }

context (Lookup)[env=p;name=n] fun:
 case e {
  (Empty)[] → undefined;
  (Bind)[name=m;val=v;env=p] → if n=m then v else (Lookup)[env=p;name=n] endif
 }
```

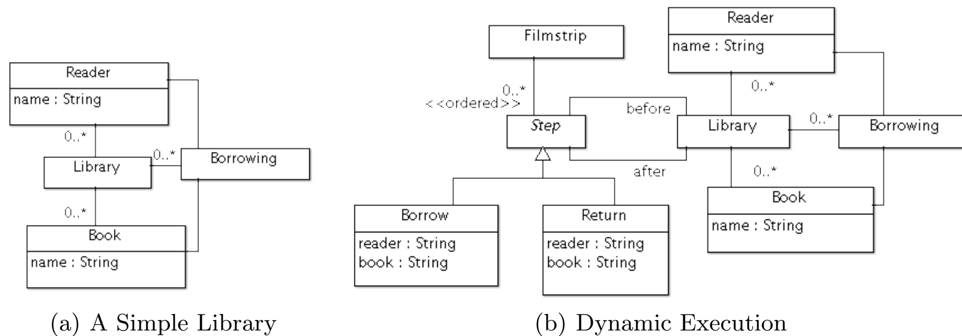
Fig. 9. Language Evaluation

6 A Language Evaluator

This section briefly shows how patterns can be used to specify a standard recursive function. Figure 8 includes a model of the simple λ-calculus and defines two properties `Eval` that maps an expression and an environment to a value, and `Lookup` that maps an environment and a name to a value. Figure 9 uses patterns to define the `Eval` and `Lookup` functions.

7 State and Collections

The previous sections have shown how patterns can be used to define functions and relations over model elements. Even though the functions are used to define evaluation, there is no implication that the specifications define how a system evolves over time. This is because the patterns and associated functions and relations are *stateless*.



(a) A Simple Library (b) Dynamic Execution

Fig. 10. A Model with State Changes and Collections

```
context (Filmstrip)[steps=S] inv:
  case S {
    Seq{} → true
    P->prepend(s1)->prepend(s2)
      where s1 = (Step)[before=l2; after=l3]
            s2 = (Step)[before=l1; after=l2]
      → (Filmstrip)[steps=P]
  }
```

Fig. 11. Specifying Execution

Models are often used to describe how a system evolves over time by showing how, complete or partial, system states are modified in response to handling an operation. Typically this is expressed using some form of state machine or process model. When an operation is performed, OCL can be used to define conditions on a *before* state, an *after* state and a relationship between the two. As with invariants, these conditions often involve complex navigation expressions.

Our proposal is that a pattern based approach can be used to define system execution in a succinct and declarative way making it easier to the user and tools to appreciate and reason about state changes. To illustrate this we use a simple model of a library shown in figure 10(a) consiting of readers, books and borrowing records. We limit ourselves to the library operations that borrow and return books.

The dynamic execution of a system can be expressed as a *filmstrip* that consists of a collection of *steps*. Each step contains a *before* and *after* state of the system. Each system operation is defined a special type of step. For simple sequential systems, filmstrips are sequences of steps where the before state of a step is the same as the after state of the immediately preceding step. For filmstrips that contain concurrency, this condition is loosened appropriately.

Figure 10(b) extends the library model with a filmstrip where the concrete steps are `Borrow` and `Return`. The general-purpose constraint that defines sequential behaviour is shown in figure 11. The invariant uses sequence patterns to require that the steps are linked through the before and after states. In particular `P->prepend(s1)->prepend(s2)` matches any sequence with two or more elements of the form `Seq{s1,s2,...}` where `P` is the sequence with `s1` and `s2` removed. Notice that the second case-arm is defined recursively and requires that the filmstrip `(Filmstrip)[steps=P]` is well-formed.

```
context
  (Borrow)[
    book   = bn;
    reader = rn;
    before =
      (Lib,l)[
        books      = Set{b=(Book)[name=bn]}->union(B);
        readers    = R=Set{r=(Reader)[name=rn]}->union(_);
        borrowings = X
      ];
    after =
      (Lib,l)[
        books      = B;
        readers    = R;
        borrowings = X->including(x)
           where x=(Borrowing)[book=b; reader=r])
      ]
  ] inv: true
```

Fig. 12. Borrowing

```
context
  (Return)[
    book   = bn;
    reader = rn;
    before =
      (Lib,l)[
        books      = B;
        readers    = R=Set{r=(Reader)[name=rn]}->union(_);
        borrowings = X->including(x)
          where  x = (Borrowing)[book=b; reader=r]
      ];
    after =
      (Lib,l)[
        books      = B->including(b=(Book)[name=bn]);
        readers    = R;
        borrowings = X
      ]
  ] inv: true
```

Fig. 13. Return

Each concrete step defines a library operation and, since these are defined as classes, they are specified separately. Figure 12 shows the definition of the step that specifies the `Borrow` operation. It is a useful example of the declarative power achieved by extending OCL with patterns because the body of the invariant form is essentially empty because all of the constraint is expressed as structural relationships between `context` and `inv:`. Note that the identity of the library object is declared to be `l` in both the before and the after which indicates that the change occurs by side-effect. The `Return` operation is specified in figure 13.

8 Conclusion

This paper has proposed extensions to OCL that aim to address the proliferation of navigation expressions that occur when expressing relationships between different parts of a model. Object expressions complete the range of values that can be denoted by general OCL expressions. Object-identities can be used to express the state changes that occur when a system performs an operation. Patterns are used to denote UML values, they include variables and pattern matching is used

to bind variables in order that a pattern explicitly denotes a value. Patterns are used in `case`-expressions and `context`-declarations in order to reduce the number of navigation expressions and to introduce a new, functional, form of declaration.

This paper has presented the OCL extensions in terms of examples and syntax definitions. Further work is needed to define the semantics of patterns and to integrate them with the semantics given in the standard. No analysis has been presented of the changes to existing tools that are implied by the addition of patterns, for example [10] examines the issues of efficient pattern matching and [11] examines type checking patterns.

References

1. András Balogh and Dániel Varró. Advanced model transformation language constructs in the viatra2 framework. In *Proceedings of the 2006 ACM symposium on Applied computing*, pages 1280–1287. ACM, 2006.
2. Lionel C Briand, Yvan Labiche, HD Yan, and Massimiliano Di Penta. A controlled experiment on the impact of the object constraint language in uml-based maintenance. In *Software Maintenance, 2004. Proceedings. 20th IEEE International Conference on*, pages 380–389. IEEE, 2004.
3. Steve Cook, Anneke Kleppe, Richard Mitchell, Bernhard Rumpe, Jos Warmer, and Alan Wills. The amsterdam manifesto on ocl. In *Object Modeling with the OCL*, pages 115–149. Springer, 2002.
4. Keith Duddy, Anna Gerber, Michael J Lawley, Kerry Raymond, and Jim Steel. Declarative transformation for OO models. *Transformation of Knowledge, Information, and Data: Theory and Applications. Idea Group Publishing*, 2004.
5. Twan van Enckevort. Refactoring uml models: using openarchitectureware to measure uml model quality and perform pattern matching on uml models with ocl queries. In *Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*, OOPSLA '09, pages 635–646, New York, NY, USA, 2009. ACM.
6. Paul Hudak and Joseph H Fasel. A gentle introduction to haskell. *ACM Sigplan Notices*, 27(5):1–52, 1992.
7. Frédéric Jouault, Freddy Allilaire, Jean Bézivin, and Ivan Kurtev. Atl: A model transformation tool. *Science of Computer Programming*, 72(1):31–39, 2008.
8. Jens Kübler and Thomas Goldschmidt. A pattern mining approach using QVT. In *Model Driven Architecture-Foundations and Applications*. Springer, 2009.
9. Ivan Kurtev. State of the art of QVT: A model transformation language standard. In *Applications of Graph Transformations with Industrial Relevance*, pages 377–393. Springer, 2008.
10. Fabrice Le Fessant and Luc Maranget. Optimizing pattern matching. In *ACM SIGPLAN Notices*, volume 36, pages 26–37. ACM, 2001.
11. Neil Mitchell and Colin Runciman. A static checker for safe pattern matching in haskell. *Trends in Functional Programming*, 6:15–30, 2005.
12. Lawrence C Paulson. *ML for the Working Programmer.* Cambridge University Press, 1996.
13. Jörn Guy Süß. Sugar for ocl. In *Proceedings of the 6th OCL Workshop at the UML/-MoDELS Conference*, pages 240–251, 2006.
14. Sam Tobin-Hochstadt. Extensible pattern matching in an extensible language. *arXiv preprint arXiv:1106.2578*, 2011.
15. Laurence Tratt. The MT model transformation language. In *Proceedings of the 2006 ACM symposium on Applied computing*, pages 1296–1303. ACM, 2006.