Translating OCL to Graph Patterns Extended Version[*]

Gábor Bergmann

Budapest University of Technology and Economics,
Department of Measurement and Information Systems,
1117 Budapest, Magyar tudósok krt. 2.
`bergmann@mit.bme.hu`

Abstract. Model-driven tools use model queries for many purposes, including validation of well-formedness rules and specification of derived features. The majority of declarative model query corpus available in industry appears to use the OCL language. Graph pattern based queries, however, would have a number of advantages due to their more abstract specification, such as performance improvements through advanced query evaluation techniques. As query performance can be a key issue with large models, evaluating graph patterns instead of OCL queries could be useful in practice.
The current paper presents an automatic mapping from a large sublanguage of OCL expressions to equivalent graph patterns in the dialect of EMF-IncQuery. Validation of benefits is carried out by performance measurements according to an existing benchmark.
Keywords: model query, OCL, graph pattern, incremental evaluation

1 Introduction

Model queries are important components in model-driven tool chains. They are widely used for specifying reports, derived features, well-formedness constraints, and guard conditions for behavioural models, design space rules or model transformations. Although model queries can be implemented using a general-purpose programming language (Java), declarative query languages may be more concise and easier to learn, among other advantages. Popular modeling platforms (e.g. the Eclipse Modeling Framework *(EMF)* [1]) support various query languages.

OCL [2] is a standard declarative model query language widely used in industry. OCL queries specify chains of navigation among model objects in a functional programming style. However, query languages inspired by *graph patterns* [3,4] (such as SPAQL [5]) resemble logic programming, where the order of model exploration is freely determined by the query engine at evaluation time. Such more

[*] This work was partially supported by the European Union and the State of Hungary, co-financed by the European Social Fund in the framework of TÁMOP 4.2.4. A/-11-1-2012-0001 'National Excellence Program', and by the CERTIMOT (ERC_HU-09-01-2010-0003) and EU FP7 MONDO (ICT-2013.1.2) projects.

abstract query specifications have numerous advantages. The steps of graph pattern matching can be automatically optimized for performance in advance by a query planner [6,7] or during evaluation by a dynamic strategy [8]. For further performance gains in case of evolving models, incremental graph pattern matcher techniques [9] can deeply analyze the query to store and maintain the result of subqueries (as in EMF-IncQuery [10], see Sec. 2.3). In search-based software engineering, if the goal condition is a graph pattern, its structure can be inspected to automatically guide [11] the design space exploration towards reaching the goal. When analyzing behavioural models, pre/post condition graph patterns can be inspected for efficient model checking [12,13] or to prove confluence [14]. It is possible to automatically generate instance models (e.g. for tool testing) that satisfy a given graph query [15] more efficiently than OCL [16].

Since the majority of declarative model query corpus available in industry appears to be OCL, the above mentioned benefits can only be reaped by translating OCL queries into graph patterns. This is not always possible, as OCL is more expressive. Nevertheless, by extending prior work [15], an automated mapping is presented in the current paper that transforms a large sublanguage of OCL expressions to equivalent graph patterns in the dialect of EMF-IncQuery.

From the benefits listed above, query performance was chosen for validating the approach, as it can be a key issue with large models. This task is carried out by performance measurements according to an existing benchmark [10].

The running example and query formalisms are introduced in Sec. 2. The mapping is specified in Sec. 3. Performance measurements are presented in Sec. 4, Sec. 5 summarizes related work, and Sec. 6 adds concluding remarks.

Beyond the contents of the conference proceedings version of the paper, this extended format also includes Appendix A, which details the proposed mapping for each operation in the OCL Standard Library.

2 Preliminaries

2.1 Running example

Several concepts will be illustrated using a simple state machine modeling language. The metamodel, defined in EMF [1] and depicted by Fig. 1, describes how state automata contain states and transitions, where the latter have a source state, a target state, and a triggering input symbol. Model queries can support the application of this metamodel in many ways (such as simulation, model checking, code generation, etc.), two of which will be explored in greater detail.

A sample instance model containing a single `Automaton`, `States` $s_1 \ldots s_6$ and the `Transitions` listed by Table 1a will be used to demonstrate model queries.

An instance model of this Ecore package is only considered *well-formed* if certain criteria are met. One such important sanity criterion is that the source and target states of a transition must both belong to the same automaton that contains the transition. A modeling environment could automatically validate instance models by issuing a model query that finds *violations* of this constraint.

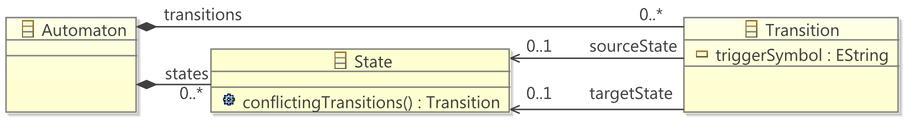


Fig. 1: Ecore Diagram of State Machine metamodel package

Table 1: Sample instance model with `conflictingTransitions` query results

Transition	source	trigger	target
t_1	s_1	A	s_2
t_2	s_1	A	s_3
t_3	s_1	B	s_4
t_4	s_1	B	s_5
t_5	s_1	C	s_6
t_6	s_3	C	s_6

(a) Transitions

State	return value
s_1	$\{t_1, t_2, t_3, t_4\}$
s_2	$\emptyset$
s_3	$\emptyset$
s_4	$\emptyset$
s_5	$\emptyset$
s_6	$\emptyset$

(b) OCL results

`conflictingTransitions`	
`self`	`t1`
s_1	t_1
s_1	t_2
s_1	t_3
s_1	t_4

(c) Pattern match set

Another use case of model queries is the definition of *derived features* - references or attributes that are not freely chosen, but are rather computed automatically from the values of other features (i.e. via a model query). The derived reference `conflictingTransitions` of class State identifies those outgoing transitions that are in conflict, i.e. share their triggering input symbol with one or more other outgoing transitions from the same state. Such a derived reference could be useful for exploring the nondeterminism of the behavioural model.

If the model is being continuously edited, the results of validation and derived feature queries have to be repeatedly updated. In case of large models, this could lead to performance problems unless incremental techniques are applied.

2.2 The OCL Language

OCL [17] is a pure functional language for defining expressions in context of a metamodel, so that the expressions can be evaluated on instance models of the metamodel. The language is very expressive, surpassing the power of first order logic by constructs such as collection aggregation operations (`sum()`, etc.). OCL queries taking a model element as input can be applied in use cases such as specifying well-formedness constraints (*invariants*).

Example 1. The OCL version of the derived feature is included as Lst. 1. When evaluated at a given `State` object, for each outgoing transition it collects the other outgoing transitions with the same trigger symbol, and the returns the accumulated set. The `Set`-valued expression is built by navigating from the `State`

Listing 1 OCL expression specifying the derived feature conflictingTransitions

```
context State def: conflictingTransitions: Set(Transition) =
  let a : Automaton = self.automaton in
    a.transitions->select(t1|t1.sourceState=self and
      a.transitions->exists(t2| t1<>t2 and
        t2.sourceState = self and t1.triggerSymbol = t2.triggerSymbol))
```

along references, and filtering the results according to attribute conditions. Results on the sample instance model are listed by Table 1b.

The rest of the section gives a basic overview of the most important characteristics of OCL expressions that will be necessary for understanding the paper; the reader is referred to the OMG standard [17] for more information.

OCL Values and Types OCL can express *values* of various *types*. *Primitive types* include character strings, integer and real numbers, etc.; `Boolean` is especially significant, e.g. for expressing well-formedness constraints. Classes in metamodels are OCL types; *instance model elements* are OCL values conforming to them, with subclassing. OCL allows constructing tuple types and collection types (`Set`, `Bag`, `OrderedSet` and `Sequence`) from any OCL type. In the current paper, primitive and metamodel types are collectively referred as *ground types*, while collection and tuple types are referred as *structured types*.

OCL Expressions OCL expressions are functions expressed on a set of *input variables* (also known as *free variables*), each with an associated type. When a type-compatible OCL value is substituted for each of these input variables, the expression evaluates to a single result value, which is compatible with the type of the expression. For an OCL expression O taking input parameters $X_1, X_2, \ldots, X_n$, let $G \models y = O(x_1, x_2, \ldots, x_n)$ denote that expression O parametrized by actual parameter values $x_1, x_2, \ldots, x_n$ yields the result y if evaluated over model G.

Expressions are compositional: an expression may have sub-expressions whose results contribute to the result of the expression. Input variables of sub-expressions are often free variables of the whole expression as well.

OCL has *literal expressions* for various types. Primitive literals have no input variables and return constants. Collection or tuple literals contain zero or more sub-expressions yielding the elements of the collection or the tuple; note that such a structure literal may have input variables due to these sub-expressions.

A *variable reference* OCL expression returns the value of its input variable. The inputless `allInstances()` expression returns a `Set` of all instances of a given metamodel type; `oclIsKindOf()` tests membership of this `Set`. The constructs `let-in` and `if-then-else` combine the results of their subexpressions in the expected way. *Property call expressions* express *navigation* from tuples to their field values, or along (single- or multi-valued) model element features; the source of navigation is identified by a single sub-expression called *source*.

Example 1 demonstrates a derived feature specification as a `let-in` OCL expression taking a `State` as input and yielding a `Set` of `Transitions` as output. The first subexpression is navigation `self.automaton`, initializing variable `a`.

Operation call expressions evaluate operations associated with the type of their *source* sub-expression. The operation takes the result of the source as its argument, and in some cases the result of other sub-expressions as additional arguments. Some significant operations will be discussed in the following.

OCL Operations Classes may declare *read-only model operations* (such as derived features) that OCL expressions can invoke on their instances. These operations can be specified as model queries (often written in OCL).

OCL also supports built-in operations on primitive types, including *arithmetic operations*, logical connectives, or comparisons (`<>` for inequality, `<=`, etc.).

Collection operations include membership testing, union, etc. of `Set`s. Operations that aggregate a collection into a single value include `size()` and `sum()`.

Iterator expressions are a special kind of collection operations that take a lambda expression (the *body*) as their argument. When evaluating the iterator expression, the body is evaluated repeatedly, with collection members substituted for one or more of its input variables (called the *iterator*). The iterator expression `select()` will evaluate a Boolean-valued body predicate on each element of a collection, and form a resulting subset/subsequence/etc. containing those elements that evaluated to true. Similarly, `exists()` returns a Boolean indicating whether any members of the collection satisfy the body predicate.

Example 1 demonstrates operations `=`, `<>`, `and`, `select()`, `exists()`.

2.3 Graph Patterns and EMF-IncQuery

Graph Patterns as a Query Language The EMF-IncQuery framework [10] aims at the efficient definition and evaluation of incremental model queries over EMF-based models, building on the idea of *graph patterns*. The query language is detailed in [18], only a brief overview is given here.

A basic graph pattern consists of *pattern constraints* expressed over *pattern variables* that represent model elements or primitive values. The *parameter variables* of a graph pattern are a subset of the pattern variables that are exposed to the query user. Pattern variables that are not parameters are called *local variables*. *Structural constraints* prescribe the existence and interconnection of graph nodes and edges of given types. *Attribute constraints* are defined by pure, deterministic *expressions* given in a Java-based language.

Basic patterns can be *composed* in numerous ways, thus the query language has the expressiveness [4] of first-order formulae over the model. *Disjunction* (OR) is expressed by several basic patterns (*pattern bodies*) defining alternative constraint sets (and local variables) for the same parameters. A *pattern call* reuses a pattern within another pattern as a single constraint expressed over its actual parameters (quantifying away the local variables of the called pattern). A *negative application condition* (NAC) is a pattern call constraint with negation, i.e. it is satisfied iff the called pattern isn't.

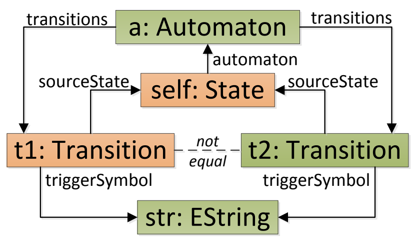
A *match* of a graph pattern is a value substitution of the parameters, so that the local variables of at least one pattern body can be assigned values to satisfy all pattern constraints of that body. The result of an (unbound) model query is the set of all matches, called the *match set*. Matches of a pattern are all tuples of the same format (one entry for each pattern parameter), and that the result of pattern matching is the set of valid matches in the model, therefore the pattern essentially evaluates to a *mathematical relation* on elements of the model and primitive values, where the arity of the relation corresponds to the number of pattern parameters, and members of the relation are the matches of the pattern.

$P(X_1, X_2, \ldots, X_n)$ will denote a pattern P having parameters $X_1, X_2, \ldots, X_n$. The fact that the tuple $\langle x_1, x_2, \ldots, x_n \rangle$ is a match of the pattern P over model G will be denoted as $G \models \langle x_1, x_2, \ldots, x_n \rangle \in MatchSet^P$.

Example 2. The derived feature in the example metamodel can be specified by the pattern `conflictingTransitions` (Fig. 2). The single pattern body imposes 8 structural constraints (existence of connecting edges, inequality) on local pattern variables `a`, `t2`, `str` and parameters `self`, `t1`. Each pattern match means that transition `t1` is included in the derived set `conflictingTransitions` of state `self`. See Table 1c for the match set on the sample model.

```
pattern conflictingTransitions (
  // parameters
  self : State, t1 : Transition
) = { // constraints of single body
  State.automaton(self, a);
  Automaton.transitions(a, t1);
  Automaton.transitions(a, t2);
  Transition.sourceState(t1, self);
  Transition.sourceState(t2, self);
  Transition.triggerSymbol(t1, str);
  Transition.triggerSymbol(t2, str);
  t1 != t2;
}
```

(a) Textual syntax



(b) Graphical form, parameters highlighted

Fig. 2: Graph pattern specifying the derived feature

Incremental Evaluation A powerful feature of EMF-IncQuery is its *incremental query evaluation*. This means that the match sets of graph patterns are cached and continuously updated as the model evolves. This choice increases memory consumption and imposes a run-time maintenance overhead on model manipulation; on the other hand, query results can be instantaneously retrieved without re-traversing the model. This characteristic can be beneficial in use cases including model validation, simulation andé derived feature computation [19,20].

The particular algorithm used in EMF-IncQuery is Rete [9], which caches match sets of subpatterns as well, with the benefit that maintenance cost is proportional to the change only, independently of model size (see [21]).

3 Mapping OCL Expressions to EMF-IncQuery

An approach for constructing semantically equivalent EMF-IncQuery graph patterns for certain kinds of OCL expressions is proposed in the following sections. Note that the graph pattern of Example 2, disregarding minor beautification, was automatically constructed from the OCL expression of Lst. 1 by a partial prototype implementation of this strategy (available at [22]).

3.1 Overview of the Approach

Graph patterns evaluate to match sets that are relations in the mathematical sense, while OCL expressions are typed functions. Thus the proposed approach aims to find relations that are equivalent to the original OCL functions, and then construct graph patterns that in turn express exactly these relations. For instance, the pattern of Example 2 is equivalent to the OCL expression of Example 1, as demonstrated on the sample instance model (Table 1).

One of the main challenges of defining such a mapping is making sure that relation domains (columns) are of ground types, as the graph pattern formalism does not support variables representing collections of model elements.

By structural recursion, the proposed approach first maps each OCL subexpression to a pattern; then these *helper patterns* are used for translating the whole expression. The helper pattern will often be included via pattern composition. In lieu of positive pattern composition, it is also possible to construct the whole pattern as a modified copy of the helper pattern, by *augmenting* it with additional pattern constraints, and/or modifying the set of pattern parameters - this approach may yield more concise output and potentially better run-time query performance. In case of multiple such subexpressions, several helper patterns can be *unified* into a single one that contains all their constraints.

An abstract specification of the proposed mapping will be provided in Sec. 3.2, by introducing possible relational representations for various kinds of OCL expressions. Then Sec. 3.3 provides the actual mapping of OCL language elements to graph patterns whose match sets will correspond to the appropriate mathematical relation specified in Sec. 3.2. The mapping is applicable to many graph query languages; only a few cases discussed in Sec. 3.4 require EMF-IncQuery-specific constructs. For the sake of brevity, the complete coverage of the OCL Standard was only included in Appendix A. Limitations will be discussed in Sec. 3.5.

3.2 Abstract Mapping to a Relational Representation

Single-valued non-Boolean expressions An OCL expression O with ground-typed inputs $X_1, X_2, \ldots, X_n$ and a ground-typed, non-Boolean result type will be mapped to a graph pattern P_O such that $G \models y = O(x_1, x_2, \ldots, x_n) \Leftrightarrow G \models$

$\langle x_1, x_2, \ldots, x_n, y \rangle \in \mathit{MatchSet}^{P_O}$ for any instance model G and appropriately typed $x_1, x_2, \ldots, x_n, y$. Simply speaking, the function is mapped to a relation expressed on the function inputs and results. From Example 1, the OCL subexpression `t1.triggerSymbol` (a function that maps a transition to a string) is equivalent to the single-constraint pattern `Transition.triggerSymbol(t1, str)` that evaluates to a relation between transitions and strings. For the instance model of Table 1a, the relation is $\{\langle t_1, A\rangle, \langle t_2, A\rangle, \langle t_3, B\rangle, \langle t_4, B\rangle, \langle t_5, C\rangle, \langle t_6, C\rangle\}$.

Note that if at least one of $x_1, x_2, \ldots, x_n, y$ has a primitive type with practically infinite instance set (e.g. 2^{64} integers), the above definition of P_O may appear to yield a practically infinite match set size, making it unfeasible to apply fully incremental evaluation model query, where all matches have to be enumerated and stored. However, as we will see below, the value of these primitive-typed variables are in many practical cases either equated to literal values, or available as an attribute value of an instance model element, or (transitively) inferrable by expression evaluation from other primitive variables that have these properties. Augmentation also improves *finiteness*: even if a helper pattern for a subexpression does not meet this condition, its augmented version associated with the composite expression may do so. Therefore typically the match set will still be finite and computable by the query engine. The proposed approach does not support cases where this condition is violated. Another limitation is that the relation domains have to be of ground types, since domains of structured types would put the relation beyond the expressive power of graph patterns.

Boolean-valued expressions An OCL expression O with ground-typed inputs $X_1, X_2, \ldots, X_n$ and a Boolean result type can be mapped to a graph pattern P_O similarly as above. Additionally, it can also be mapped to graph patterns P_O^+ or P_O^- that match those inputs for which the expression evaluates to true respectively false: $G \models \mathit{true} = O(x_1, x_2, \ldots, x_n) \Leftrightarrow G \models \langle x_1, x_2, \ldots, x_n \rangle \in \mathit{MatchSet}^{P_O^+} \Leftrightarrow G \models \langle x_1, x_2, \ldots, x_n \rangle \notin \mathit{MatchSet}^{P_O^-}$ for any instance model G and appropriately typed $x_1, x_2, \ldots, x_n, y$. From Example 1, let O be the OCL subexpression `t1 <> t2` (a function that maps two transitions to a Boolean); then binary pattern P_O^+ has the constraint `t1 != t2` (and implicit type restrictions) and no Boolean variables; while P_O^- has `t1 == t2` and evaluates to $\{\langle t_1, t_1\rangle, \langle t_2, t_2\rangle, \langle t_3, t_3\rangle, \langle t_4, t_4\rangle, \langle t_5, t_5\rangle, \langle t_6, t_6\rangle\}$ for the model of Table 1a.

For each Boolean-valued OCL expression O, it is sufficient to define one of the three mappings P_O, P_O^+, P_O^-, as it can then be trivially transformed into the other two, unless a simpler mapping is known for them. P_O^+ (respectively P_O^-) can be synthesized from P_O by asserting `y == true;` (respectively `y == false;`) as an additional pattern constraint, and removing y from the pattern parameters. P_O^+ and P_O^- transform into each other via negative pattern call. Finally, P_O can be derived from P_O^+ (respectively P_O^-) by counting its matches, and then evaluating the Boolean expression that the number of matches is positive (respectively zero).

The reason for having three possible images P_O, P_O^+, P_O^- for a Boolean-valued expression O is that OCL often uses Boolean variables as conditions (e.g. in `if`, `select()`, or logical connectives), in which cases it is natural to include a pattern composition constraint of P_O^+ or P_O^- (or augment it, as discussed before). Thus the mapping result is simplified (potentially gaining run-time query performance benefits as well) in case P_O^+ or P_O^- are simpler to express than P_O.

Tuple-valued and tuple-consuming expressions Since tuples consist of a statically known number of components, a tuple-typed variable can always be substituted with a set of variables, one for each tuple field. This principle can be applied to expression inputs and results in an analogous way; the latter case is elaborated in more detail below.

An OCL expression O with ground-typed inputs $X_1, X_2, \ldots, X_n$ and a k-ary tuple-typed result can be mapped to a graph pattern P_O such that $G \models \langle y_1, y_2, \ldots, y_k \rangle = O(x_1, x_2, \ldots, x_n) \Leftrightarrow G \models \langle x_1, x_2, \ldots, x_n, y_1, y_2, \ldots, y_k \rangle \in MatchSet^{P_O}$ for any instance model G and appropriately typed $x_1, x_2, \ldots, x_n$ as well as $y_1, y_2, \ldots, y_k$. Simply speaking, the function is mapped to a relation expressed on the function inputs and tuple components of the result.

If the result is a tuple of ground-typed fields, then the domains of the relation are of ground types. Tuples containing tuples can be trivially flattened before the mapping to tuples containing ground-typed values only. For tuples having one or more collections as components, see the following paragraphs.

Multi-valued expressions An OCL expression O with ground-typed inputs $X_1, X_2, \ldots, X_n$ and a collection result type will be mapped to a graph pattern P_O such that $G \models y \in O(x_1, x_2, \ldots, x_n) \Leftrightarrow G \models \langle x_1, x_2, \ldots, x_n, y \rangle \in MatchSet^{P_O}$ for any instance model G and appropriately typed $x_1, x_2, \ldots, x_n, y$. Simply speaking, the function is mapped to a relation expressed on the function inputs and elements appearing in the result, where each element of the result collection corresponds to a separate element of the associated relation. From Example 1, the OCL subexpression `a.transitions` (a function that maps an automaton to a set of transitions) is equivalent to the single-constraint pattern `Automaton.transitions(a, t1)` that evaluates to a relation between automatons and transitions, with one row for each transition. Similarly, the graph pattern of Example 2 evaluates to a relation (see Table 1c) that associates a `State` with individual `Transitions`, as opposed to a `Set` of transitions, which is what the equivalent OCL derived feature of Example 1 yields (see Table 1b).

If the element type of the collection is a ground type, then the domains of the relation are of ground types. Tuples can be dealt with as described in Sec. 3.2. Collections of collections (as well as tuples of more than one collection) are not supported by the approach due to the limitations discussed before.

Relations (pattern match sets) have set semantics, without multiplicity or ordering. Thus only `Set` collections can be faithfully mapped (and also `Bag`s in case input and internal variables together make the output unique); other collection types are not supported in general. However, many collection operations (such as `isEmpty()`) and iterator expressions (such as `select()`) behave equivalently for the various collection types, in which case the collection can be implicitly cast to a `Set` by `asSet()` for the sake of the mapping.

The proposed approach does not support collection-typed input variables in OCL expressions, as collection operations are typically mapped to pattern composition constructs that call the pattern associated with the expression that defines the collection. Note that a collection can be used as an argument of an OCL operation, if it is provided as the result of a sub-expression (typically navigation along a multi-valued property); collection types are unsupported for free variables only. In practice, this limitation is not directly relevant for class invariants and derived features (due to single non-collection input); so OCL-defined model operations and preconditions are restricted only in their parametrization. The iterator input variable of an iterator expression body can be a collection only in case of a collection of collections, which is unsupported anyway. The only other way a new variable can be introduced is a `let` expression, in which case the initialization expression of the variable can replace the variable references in the `in` branch for the sake of the mapping, so once again it will not matter whether the type is a collection.

3.3 Concrete Mappings for Simple Expressions

The following paragraphs construct mappings of the simplest OCL expression into graph patterns according to the specifications in Sec. 3.1. The mappings result in single-bodied patterns unless indicated otherwise.

Navigation and variable references If O is a navigation expression along property *edgeType* and with source expression O^{source}, where O^{source} is mapped to pattern $P_{O^{source}}$ with parameters $x_1, x_2, \ldots, x_n, y^{source}$, then O is mapped to P_O with parameters $x_1, x_2, \ldots, x_n, y$. P_O is constructed by augmenting $P_{O^{source}}$ by a new structural constraint $edgeType(y^{source}, y)$ and replacing pattern parameter y^{source} with y. This works both for single-valued and multi-valued (collection-typed) properties. Mapping variable references is trivial.

For instance, `self.automaton` from Example 1 is translated in Example 2 to `State.automaton(self, a)`; note the variable reference `self` as source expression. On the other hand, a hypothetical `self.automaton.transitions`, containing the former OCL expression as its source expression, would augment this pattern by a second pattern constraint `Automaton.transitions(a, y)`.

Type checks and literals If O is T`.allInstances()` for metamodel class T, it is mapped to the pattern P_O with parameter y and single pattern constraint $T(y)$; the same pattern is P_O^+ if O is y`.oclIsKindOf(`T`)`. If O is a primitive-typed literal of value c, it is mapped to the pattern P_O with parameter y and the single pattern constraint c==y. For treatment of tuple literals, see Sec. 3.2. `Set` literals are mapped to a disjunction of helper patterns mapped from subexpressions.

Arithmetic operations If O is an arithmetic operation *op* on subexpressions $O^1, O^2, \ldots, O^m$, then O is mapped to P_O with parameters consisting of all input parameters of $P_{O^1}, P_{O^2}, \ldots, P_{O^m}$ in addition to y, and with the attribute constraint y==`eval`($op(y^1, y^2, \ldots, y^m)$) (where y^i is the result variable of P_{O^i}) augmenting the unification of P_{O^1}, P_{O^2}, ..., P_{O^m}. For instance, OCL expression $p < q+r$ is mapped to pattern constraints y^1==`eval`($q+r$) and y==`eval`($p < y^1$).

If O is an equality, it can be more effectively mapped to P_O^+ using a pattern constraint y^1==y^2 and to P_O^- as y^1`!=`y^2 instead of the `eval` construct. Vice versa for inequality; e.g. `<>` from Example 1 is mapped to a `!=` constraint in Example 2.

Similarly, many Boolean operations have simpler mappings. In case of `and`, the single body of P_O^+ is the unification of $P_{O^1}^+$ and $P_{O^2}^+$ (as applied repeatedly in the running example); while P_O^- would have two bodies: $P_{O^1}^-$ and $P_{O^2}^-$.

If-then-else and let-in In a let-in expression, the result of the `let` subexpression is used to parameterize the `in` subexpression. If O is a let-in expression with subexpressions O^{let}, O^{in}, then O is mapped to P_O with parameters consisting of y along with input variables of $P_{O^{let}}$ and input variables of $P_{O^{in}}$ except for the result variable of $P_{O^{let}}$; with the pattern body unifying $P_{O^{let}}$ with $P_{O^{in}}$. For instance, constraint `State.automaton(self, a)` in Example 2 is from $P_{O^{let}}$.

If O is an if-then-else expression with subexpressions $O^{condition}$, O^{then}, O^{else}, then O is mapped to P_O with parameters consisting of all input parameters of $P_{O^{condition}}$, $P_{O^{then}}$, $P_{O^{else}}$ in addition to y, and with two pattern bodies, one with y^{then}==y augmenting the unification of $P_{O^{then}}$ and $P_{O^{condition}}^+$, the other with y^{else}==y augmenting the unification of $P_{O^{else}}$ and $P_{O^{condition}}^-$. Can be simplified to Boole-logic if the result type is Boolean.

First-order collection expressions Many collection operations and iterator expressions are trivial to translate to first-order logic formulae, which are within the power of graph patterns [4]. A few cases will be briefly outlined below.

For instance, a collection is non-empty iff the mapped pattern has any matches with the given values of input variables. If O is an `isEmpty()` expression with subexpression O^{source}, then O is mapped to P_O^-, which is the same as $P_{O^{source}}$, with its result variable removed (quantified away) from the parameters.

If O is a `select()` expression with subexpressions O^{source}, O^{body}, then P_O is $P_{O^{source}}$ and $P_{O^{body}}^+$ unified, with the result variable of the former substituted for the iterator variable of the latter (and both removed from the parameters). For `exists()`, P_O^+ is constructed similarly, but the result variable is removed from the parameters. Example 1 demonstrates both cases.

3.4 Mapping Higher-order OCL Constructs

Some OCL constructs are not expressible using first-order formulae, but the EMF-IncQuery language provides extensions over conventional graph patterns that may suffice in some cases. As above, details will be omitted here.

EMF-IncQuery supports transitive closure [23], so a `closure()` iterator expression can be mapped by (1) mapping first the body expression to a graph pattern, (2) taking the transitive closure of this graph pattern, and (3) augmenting the graph pattern mapped from the source expression with the transitive call.

The simplest case of aggregation is the `size()` collection operation returning the number of elements of a set. A `count find` constraint in EMF-IncQuery can aggregate matches of the graph pattern corresponding to the source expression defining said set. An analogous solution is proposed for OCL aggregation operations `sum()`, etc.; but the corresponding EMF-IncQuery aggregators, while included in the language specification, are not fully implemented as of today.

3.5 Miscellaneous Cases and Limitations

Operation calls toward metamodel-defined custom (read-only) operations are trivial to support if they are defined as OCL expressions (or EMF-IncQuery patterns, as in [20]). Operations implemented in a generic-purpose programming language are not supported in general, as there is no universal way to ensure that the incremental engine is notified of changes in the computation result, which is necessary for incremental maintenance. A solution [24] has been proposed which records all model reads during the computation to invalidate the result when these parts of the model are affected by a change, but this approach has its own practical limitations, as it would require wrapping all model processing - including the implementation of the metamodel-defined read-only operation - into a compliant model access layer.

As discussed throughout Sec. 3, the proposed approach has limitations. Due to the lack of support for ordering in the relational representation, iterator expressions `sortedBy()` and `iterate()` cannot be mapped, similarly to order-sensitive operations (e.g. `first()`, `at()`) on ordered collections. Representation of multiplicity (i.e. `Bag` collection) has limitations as well. Support for collections of collections is also lost due to the relational approach. As discussed before, the usage of collections of primitive types and primitive-typed top-level arguments is restricted due to finiteness / computability limitations of EMF-IncQuery.

OCL has two special *undefined values*, `null` and `invalid`, which conform to (almost) all OCL types, but are not equivalent to each other. The proposed approach does not support them at the moment, partly due to type system incompatibility, and also due to semantic issues [25]; see [16] for a possible workaround.

Altogether it is clear that the mapped sublanguage is significantly weaker than OCL. Still, practice has shown that the supported OCL constructs are expressive enough to be useful in many cases.

4 Performance Measurements

The justification of the proposed mapping is that one can deliver efficient, incremental query evaluation for a subset of OCL expressions by transforming them to graph patterns of equivalent semantics, and applying EMF-IncQuery. To demonstrate this, a subset of an existing performance benchmark for well-formedness (invariant) constraint checking was applied.

Table 2: SignalNeighbor evaluation times for the instance model of 213K elements

Tool	Java	OCL	OCL-CG	OCL-IA	EIQ	OCL2IQ
Batch Validation [ms]	169 867	36 157	126 461	36 444	6 142	6 205
Continuous Validation Time [ms]	167 891	32 237	126 723	331 523	2	1
Memory Footprint [kB]	14 009	15 304	17 755	26 073	108 435	118 319

4.1 Measurement Setup

The Train Benchmark [10] defines a number of well-formedness constraints (of which only `SignalNeighbor` is used here) in a custom metamodel, and measures the constraint checking performance of various model query tools as they process automatically generated instance models of various sizes conforming to the metamodel. The goal is to provide near instantaneous feedback on constraint violations as the (simulated) user is editing a large model. The workload and measured performance indicators involve: (*phase 1*) reading the model, (*phase 2*) checking it for any inconsistencies as defined by the well-formedness constraint, (*phase 3*) simulating a transformation / manual editing of the model that performs a predefined sequence of modifications, and (*phase 4*) checking the updated model as well for inconsistencies. For fair comparison [10] of stateless tools against incremental ones, the most relevant performance indicators are *phase 1+2* ("Batch Validation") execution time and *phase 3+4* ("Continuous Validation") execution time (and of course the memory footprint). The workflow actually executes *phase 3+4* repeatedly; the reported values are the average time of one repetition (small modification + 1 query).

The run-time performance of the following solutions were compared[1]. **Java**: a naive Java implementation of the constraint check, as a hypothetical programmer would quickly implement it, without any special effort to improve performance. **EIQ**: hand-written graph patterns evaluated incrementally by EMF-IncQuery. **OCL**: the OCL interpreter [2] of Eclipse, as it evaluates the OCL representation of the constraint check. **OCL-CG**: is Java code generated from the same OCL expression by Eclipse OCL [2]. **OCL-IA**: the OCL Impact Analyzer [26] toolkit, as it incrementally evaluates the same OCL expression. **OCL2IQ**: graph patterns automatically derived from the same OCL expression by a prototype partial implementation of the proposed mapping, likewise interpreted incrementally by EMF-IncQuery (new contribution extending [10]).

4.2 Results

Results obtained from the input model of 213K elements (nodes+edges) are presented in Table 2; details and further experiments are reported at [22] along with instructions for reproduction.

The incremental strategy of EMF-IncQuery performs extremely well in the "Continuous Validation" workload, delivering practically immediate feedback after model manipulation, at the cost of increased memory footprint. Furthermore, comparison against benchmark instances with different model sizes [22] confirms the theoretical result that this "Continuous Validation" time is practically independent of the size of unchanging parts of the model; EMF-IncQuery memory consumption and "Batch Validation" time was found to scale approximately proportionally to model size, while OCL execution times are between a linear and quadratic proportion to model size. Finally, the graph queries automatically generated using the proposed transformation (OCL2IQ) perform similarly to manually written EMF-IncQuery code (EIQ), outperforming pure Java as well as stateless or incremental OCL-based approaches.

The advantage of graph patterns at "Batch Validation" time likely stems from automatic query planning, while "Continuous Validation" times are a consequence of the deep caching of the Rete incremental evaluation strategy; these are two of the benefits of the proposed approach foreseen in Sec. 1. Thus translating OCL code to graph patterns is justified in this scenario.

4.3 Remarks and Threats to Validity

Diverging from [10] at the suggestion of Eclipse OCL leader Ed Willink, OCL evaluation was not invoked by substituting each model element as `self`, but only on a prefiltered list of instances of the context type of the constraint.

The performance of incremental techniques may depend on what kind of changes are performed in *phase 3*. The presented results were obtained from the *UserScenario* mode of Train Benchmark. The "Continuous Validation" times for OCL-IA are significantly worse in this case than with the alternative model manipulation workload *ModelXFormScenario* (see [22]), where OCL-IA re-evaluation is quick after a change, leading to efficient incrementality. EIQ and OCL2IQ are much less sensitive to this option, in line with theoretical predictions [21].

Note that the OCL query was produced by non-experts. Hand-optimized queries may perform better. However, the OCL2IQ approach received the same unoptimized query as input, so the comparison is fair.

The benchmark scenario was deliberately chosen as one where incremental approaches have potential advantages, and the selected query was complex to

[1] Experimental setup: Dell Latitude E5420 Laptop, Intel Core i5-2430M @ 2.4Ghz CPU, 16GB of DDR3-1066/1333 RAM, Samsung SSD 830; Eclipse Kepler on Java SE 1.7.0_05-b06 (with 2G maximum heap size) on Windows 7 x64; Eclipse OCL pre-release version 3.4.0.v20140124-1452, EMF-IncQuery 0.8.0 (nightly at 2014-03-05).

increase the role of automatic query optimization. Therefore the results do not show universal superiority of one tool over another, merely produce evidence that the proposed approach has legitimate use cases.

5 Related work

5.1 Translating OCL to Logic-based Languages

A similar translation procedure from OCL to graph patterns was utilized in [15], focusing on providing a means to automatically generate large instance models (e.g. for testing) that conform to a metamodel with OCL invariants. Compared to the proposed approach, [15] handles a smaller subset of OCL, translates it into a slightly different graph query language, and does not investigate query performance. Due to conceptual differences, the translation method proposed here is not a straightforward extension of theirs, even if there are some common elements. Particularly focusing on differences between the supported subsets of OCL, [15] has the following shortcomings: (i) support is focused on Boolean-valued OCL expressions only (though non-Boolean navigations can be used in certain ways); (ii) set operations such as `select()`, `collect()`, `union()`, etc. are not supported; (iii) aggregations such as `sum()` are not supported; (iv) the result of `size()` can only be compared against constants; (iv) the result of two paths of navigation can only be compared for equality. Thus e.g. the derived feature of Lst. 1 cannot be translated for multiple reasons.

Metamodel consistency checkers UML2Alloy [16] and UMLtoCSP [27] compile OCL to a constraint or logic language, similarly to the proposed approach; but without “flattening" collections to relational semantics (contrast Sec. 3.2). Thus the expressive power of OCL is preserved (at least for [27]), but the Rete algorithm (and some other benefits foreseen in Sec. 1) cannot be applied.

Mappings to formal semantic domains such as HOL (higher-order-logic) revealed [25] inconsistencies and ambiguities in the OCL standard. Fortunately, they have low impact on the OCL sublanguage supported in the current paper. Such transformations could not be directly reused for the same reason as above.

5.2 Incremental Evaluation of OCL

Due to the expressive power of OCL constructs, the Rete-based approach used in EMF-IncQuery is not applicable for all queries formulated as OCL expressions. There are, however, alternative approaches for incremental evaluation of OCL queries, though they have a lower level of incrementality [21] than Rete.

Cabot's approach [28] and the Impact Analyzer [26] extension of the freely available query engine Eclipse OCL [2] rely on static analysis of OCL expressions when computing an over-estimate of query inputs that need to be re-evaluated from scratch for given elementary model change.

The Groher-Reder-Egyed approach [24] for incremental constraint checking is independent from the constraint language, but can be instantiated for OCL. The strategy is to wrap the model into a model access layer that records elementary model access operations, such as retrieving the value of an attribute, during the query evaluation; later the query can be re-evaluated for the given input if any of the recorded elementary queries are affected by a change. Some re-evaluations can be saved by language-specific maintenance [29] of a Boolean validation tree.

Case study-driven comparative performance benchmarking of incremental model query evaluation technologies is a currently ongoing effort [30,31,10].

6 Conclusion

The paper presented a general specification for mapping a large subset of OCL expressions to equivalent graph patterns, and provided concrete translations conforming to this scheme for numerous OCL constructs and Standard Library operations, while clearly indicating any limitations of the approach.

Experiments have demonstrated that query performance can be increased by evaluating the generated graph patterns (using EMF-IncQuery) instead of the original OCL expressions, which was one of the benefits of the approach foreseen in Sec. 1. Although the measurements do not constitute a comprehensive performance assessment of the various tools, they suffice for proving the existence of cases where the proposed mapping can be directly useful.

The author wishes to thank Ed Willink for his advice on Eclipse OCL.

A Mapping Standard OCL Operations

The following sections present the proposed mapping (if any) for each of the operations defined in the OCL Standard Library [17], organized by OCL type. Departing from the organizing principle found in the standard, mapping of overridden operations is indicated at the topmost defining type; overridden operations are not listed at the subtypes.

When specifying the mapping of an OCL expression, the name O will always be assumed to refer to the current expression under consideration. When specifying P_O, the variable `y` is assumed to represent the result.

Operation sources and arguments are always given here as free variables; the source is assumed to be the variable `self`. If the OCL expression contains other kinds of source or argument subexpressions, then (according to Sec. 3.1) the return value of the subexpression should be used instead of the given free variable; the subexpression itself should be mapped separately to a helper pattern. Usually, the helper pattern is then implicitly unified with the mapping of the main expression given below. The exception to this implicit unification is when the translated form of the subexpression is explicitly used in the mapping (e.g. for complex pattern composition / unification); for this purpose the helper pattern will be referred to using the notation for mapped OCL expressions (such as P_{self}, P^+_{self} or P^-_{self}), where the variable name stands for the subexpression to be mapped.

For multi-valued subexpressions, the variable corresponding to the subexpression will always refer to an element of the collection, as opposed to the `Collection` itself, as explained in Sec. 3.2. Tuple-valued subexpressions or return types will be decomposed into ground-typed variables, in accordance with Sec. 3.2.

In case of Boolean-typed expressions, when one or more of P_O, P_O^+ and P_O^- are omitted from the explicit mappings given below, they should be derived from one of the explicitly given forms as described in Sec. 3.2; thus all three mappings are implicitly provided here.

A.1 Operations on Special OCL Types

OclAny

=(object2 : OclAny) : Boolean

- When comparing primitive-typed values or model elements, mapped to the single-constraint P_O^+ as `self==object2`, or to P_O^- as `self!=object2`, or to P_O as `y==eval(self == object2)`. Note that OCL distinguished values `null` and `invalid`, along with their special semantics, are currently unsupported.
- When comparing tuples, element-wise comparison is applied (see Sec. 3.2).
- When comparing `Set`s, treated exactly as the equivalent OCL expression `self->symmetricDifference(object2)->isEmpty()`.
- Other collection types are unsupported.

<>(object2 : OclAny) : Boolean

- When comparing primitive-typed values or model elements, mapped to the single-constraint P_O^+ as `self!=object2`, or to P_O^- as `self==object2`, or to P_O as `y==eval(self != object2)`. Note that OCL distinguished values `null` and `invalid`, along with their special semantics, are currently unsupported.
- When comparing tuples, element-wise comparison is applied (see Sec. 3.2).
- When comparing `Set`s, treated exactly as the equivalent OCL expression `self->symmetricDifference(object2)->notEmpty()`.
- Other collection types are unsupported.

oclAsSet() : Set(T)

No-op, i.e. such an expression is treated equivalently to its source expression, due to the set semantics of graph patterns.

oclIsNew() : Boolean

Postcondition-specific language aspects are unsupported.

oclIsUndefined() : Boolean

OCL distinguished values `null` and `invalid` are currently unsupported.

oclIsInvalid() : Boolean

OCL distinguished values `null` and `invalid` are currently unsupported.

oclAsType(type : Classifier) : T

No-op, i.e. such an expression is treated equivalently to its source expression.

oclIsTypeOf(type : Classifier) : Boolean

Unsupported, as EMF-IncQuery currently provides no way of expressing a direct instantiation constraint. If such a pattern constraint is added to the target language in the future, the operation should be handled analogously to `oclIsKindOf`.

oclIsKindOf(type : Classifier) : Boolean

Mapped to the single-constraint P_O^+ as `type(self)`.

oclIsInState(statespec : OclState) : Boolean

State-specific language aspects are unsupported.

oclType() : Classifier

Meta-queries are currently unsupported.

oclLocale : String

The mapping currently does not address the question of localization.

OclVoid

OCL distinguished values `null` and `invalid` are currently unsupported.

OclInvalid

OCL distinguished values `null` and `invalid` are currently unsupported.

OclMessage

Message passing is unsupported.

A.2 Operations on Primitive Types

Real

OCL type `Real` is represented as Java type `java.lang.Double`.

+(r : Real) : Real

Mapped to the single-constraint P_O as
`y==eval(self + r)`.

-(r : Real) : Real

Mapped to the single-constraint P_O as
`y==eval(self - r)`.

***(r : Real) : Real**

Mapped to the single-constraint P_O as
`y==eval(self * r)`.

- : Real

Mapped to the single-constraint P_O as
`y==eval(-self)`.

/(r : Real) : Real

Mapped to the single-constraint P_O as
`y==eval(self / (double)r)`.

abs() : Real

Mapped to the single-constraint P_O as
`y==eval(java.lang.Math.abs(self))`.

floor() : Integer

Mapped to the single-constraint P_O as
`y==eval((int) java.lang.Math.floor(self))`.

round() : Integer

Mapped to the single-constraint P_O as
`y==eval((int) java.lang.Math.round(self))`.

max(r : Real) : Real

Mapped to the single-constraint P_O as
`y==eval(java.lang.Math.max(self, r))`.

min(r : Real) : Real

Mapped to the single-constraint P_O as
`y==eval(java.lang.Math.min(self, r))`.

<(r : Real) : Boolean

Mapped to the single-constraint P_O as
`y==eval(self < r)`.

>(r : Real) : Boolean

Mapped to the single-constraint P_O as
`y==eval(self > r)`.

<=(r : Real) : Boolean

Mapped to the single-constraint P_O as
`y==eval(self <= r)`.

>=(r : Real) : Boolean

Mapped to the single-constraint P_O as
`y==eval(self >= r)`.

toString() : String

Mapped to the single-constraint P_O as
`y==eval(java.lang.Double.toString(self))`.
If a is statically known to be an `Integer`,
`y==eval(java.lang.Integer.toString(self))`
is used instead.

Integer

OCL type `Integer` is represented as Java type `java.lang.Integer`.

div(i : Integer) : Integer

Mapped to the single-constraint P_O as
`y==eval(self / i)`.
Note the difference to OCL operation `/`.

mod(i : Integer) : Integer

Mapped to the single-constraint P_O as
`y==eval(self % i)`.

String

OCL type `String` is represented as Java type `java.lang.String`.

+(s : String) : String

Mapped to the single-constraint P_O as
`y==eval(self + s)`.

size() : Integer

Mapped to the single-constraint P_O as
`y==eval(self.length())`.

concat(s : String) : String

Equivalent to operation +.

substring(lower : Integer, upper : Integer) : String

Mapped to the single-constraint P_O as
`y==eval(self.substring(1 + lower, upper))`.

toInteger() : Integer

Mapped to the single-constraint P_O as
`y==eval(java.lang.Integer.valueOf(self))`.

toReal() : Real

Mapped to the single-constraint P_O as
`y==eval(java.lang.Double.valueOf(self))`.

toUpperCase() : String

Mapped to the single-constraint P_O as
`y==eval(self.toUpperCase())`.
Note that only the default locale is supported currently.

toLowerCase() : String

Mapped to the single-constraint P_O as
`y==eval(self.toLowerCase())`.

indexOf(s : String) : Integer

Mapped to the single-constraint P_O as
`y==eval(1 + self.indexOf(s))`.

equalsIgnoreCase(s : String) : Boolean

Mapped to the single-constraint P_O as
`y==eval(self.equalsIgnoreCase(s))`.
Note that only the default locale is supported currently.

at(i : Integer) : String

Mapped to the single-constraint P_O as
`y==eval(java.lang.String.valueOf(self.charAt(i)))`.

toBoolean() : Boolean

Equivalent to OCL Expression `self = 'true'`.

<(s : String) : Boolean

Mapped to the single-constraint P_O as
`y==eval(self.compareTo(s) < 0).`
Note that only the default locale is supported currently.

>(s : String) : Boolean

Mapped to the single-constraint P_O as
`y==eval(self.compareTo(s) > 0).`
Note that only the default locale is supported currently.

<=(s : String) : Boolean

Mapped to the single-constraint P_O as
`y==eval(self.compareTo(s) <= 0).`
Note that only the default locale is supported currently.

>=(s : String) : Boolean

Mapped to the single-constraint P_O as
`y==eval(self.compareTo(s) >= 0).`
Note that only the default locale is supported currently.

Boolean

OCL type `Boolean` is represented as Java type `java.lang.Boolean`.

or (b : Boolean) : Boolean

Mapped to P_O^+ as a disjunction between P_{self}^+ and P_b^+. Mapped to P_O^- as the unification of P_{self}^- and P_b^-. Mapped to the single-constraint P_O as
`y==eval(self || b).`

xor (b : Boolean) : Boolean

Mapped to P_O^+ as a disjunction between two bodies; the first is the unification of P_{self}^+ and P_b^-; the second is the unification of P_{self}^- and P_b^+. Mapped to P_O^- as disjunction between two bodies; the first is the unification of P_{self}^+ and P_b^+; the second is the unification of P_{self}^- and P_b^-. Mapped to the single-constraint P_O as
`y==eval(self ^ b).`

and (b : Boolean) : Boolean

Mapped to P_O^+ as the unification of P_{self}^+ and P_b^+. Mapped to P_O^- as a disjunction between P_{self}^- and P_b^-. Mapped to the single-constraint P_O as
`y==eval(self && b).`

not : Boolean

Mapped to P_O^+ as P_{self}^-. Mapped to P_O^- as P_{self}^+. Mapped to the single-constraint P_O as
`y==eval(!self)`.

implies (b : Boolean) : Boolean

Mapped to P_O^+ as a disjunction between P_{self}^- and P_b^+. Mapped to P_O^- as the unification of P_{self}^+ and P_b^-. Mapped to the single-constraint P_O as
`y==eval(!self || b)`.

toString() : String

Mapped to the single-constraint P_O as
`y==eval(java.lang.Boolean.toString(self))`.

UnlimitedNatural

The `unlimited` value is currently unsupported; otherwise the type behaves equivalently to `Integer`.

A.3 Operations on Collection Types

Collection

size() : Integer

In general, only supported for `Set` or `OrderedSet`. Mapped to the single-constraint P_O as
`y==count find` P_{self}.

Not supported for `Bag` or `Sequence` in general, except for the special case detailed at the description of `collect()` in Sec. A.4.

includes(object : T) : Boolean

Mapped to the single-constraint P_O^+ as
`object==self`.

excludes(object : T) : Boolean

Mapped to the single-constraint P_O^- as
`object==self`.

count(object : T) : Integer

In general, only supported for `Set` or `OrderedSet`. Mapped to the single-constraint P_O as
`y==count find` $P^{+}_{includes}$,
where helper pattern *includes* refers to the OCL expression
`self->includes(object)`.

Not supported for `Bag` or `Sequence` in general, except for the special case detailed at the description of `collect()` in Sec. A.4.

includesAll(c2 : Collection(T)) : Boolean

Mapped to P^{-}_{O} as the augmentation of P_{self} with two additional constraints, `self==c2` and a negative application condition of P_{c2}.

excludesAll(c2 : Collection(T)) : Boolean

Mapped to P^{-}_{O} as the unification of P_{self}, P_{c2} and the additional constraint `self==c2`.

isEmpty() : Boolean

Mapped to P^{+}_{O} as a negative application condition of P_{self}. Mapped to P^{-}_{O} as P_{self}, where the `self` is demoted to a local variable (i.e. it is quantified).

notEmpty() : Boolean

Mapped to P^{+}_{O} as P_{self}, where the `self` is demoted to a local variable (i.e. it is quantified). Mapped to P^{-}_{O} as a negative application condition of P_{self}.

max() : T

Mapped to the single-constraint P_O as
`y==maximum(self) find` P_{self}.
Note that the EMF-IncQuery aggregator `maximum()`, while included in the language specification, is not fully implemented as of today.

min() : T

Mapped to the single-constraint P_O as
`y==minimum(self) find` P_{self}.
Note that the EMF-IncQuery aggregator `minimum()`, while included in the language specification, is not fully implemented as of today.

sum() : T

In general, only supported for `Set` or `OrderedSet` collections of `Integer` or `Real`. Mapped to the single-constraint P_O as
`y==sum(self) find` P_{self}.

Note that the EMF-IncQuery aggregator `sum()`, while included in the language specification, is not fully implemented as of today.

Not supported for `Bag` or `Sequence` in general, not even for such collections of `Integer` or `Real`, except for the special case detailed at the description of `collect()` in Sec. A.4.

product(c2 : Collection(T)) : Set(Tuple(first: T, second: T2))

Mapped to P_O as the unification of P_{self}, P_{c2} and two additional constraints `self==first`, `c2==second`.

selectByKind(type : Classifier) : Collection(T)

Mapped to P_O as constraints `type(self)` and `y==self`.

selectByType(type : Classifier) : Collection(T)

Unsupported, as EMF-IncQuery currently provides no way of expressing a direct instantiation constraint. If such a pattern constraint is added to the target language in the future, the operation should be handled analogously to `selectByKind()`.

asSet() : Set(T)

Mapped to single-constraint P_O as
`y==self`.
The resulting pattern parameters are `y` and the free variables of the OCL expression; thus iterator variables introduced by the workaround described at `collect()` (see Sec. A.4) are demoted to local variables (in other cases, this is a no-op).

asOrderedSet() : OrderedSet(T)

Treated equivalently to the `asSet()` operation.

asSequence() : Sequence(T)

No-op, i.e. such an expression is treated equivalently to its source expression, due to the set semantics of graph patterns.

asBag() : Bag(T)

No-op, i.e. such an expression is treated equivalently to its source expression, due to the set semantics of graph patterns.

flatten() : Collection(T2)

Collections of collections are, in general, unsupported.

Common Operations of Set, Bag and Sequence

The following `Set` operations are treated equivalently to their polymorphic counterparts in `Bag` and `Sequence` (as well as argument polymorphic variants in `Set`), since membership multiplicity is not supported.

union(s : Set(T)) : Set(T)

Mapped to P_O as a disjunction of two bodies. The first body augments P_{self} by constraint `y==self`. The second body augments P_s by constraint `y==s`.

intersection(s : Set(T)) : Set(T)

Mapped to P_O as constraints `y==self` and `y==s`.

including(object : T) : Set(T)

Mapped to P_O as a disjunction of two bodies. The first body augments P_{self} by constraint `y==self`. The second body augments P_{object} by constraint `y==object`.

excluding(object : T) : Set(T)

Mapped to P_O as the augmentation of P_{self} with three constraints `y==self`, `y==object` and `neg find` P_{object}.

Set

-(s : Set(T)) : Set(T)

Mapped to P_O as the augmentation of P_{self} with three constraints `y==self`, `y==s` and `neg find` P_s.

symmetricDifference(s : Set(T)) : Set(T)

Mapped to P_O as the disjunction of two bodies. The first body is the augmentation of P_{self} with three constraints `y==self`, `y==s` and `neg find` P_s. The second body is the augmentation of P_s with three constraints `y==self`, `y==s` and `neg find` P_{self}.

Bag

All operations of `Bag` have already been discussed.

Common Operations of OrderedSet and Sequence

The following `OrderedSet` operations are treated equivalently to their polymorphic counterparts in `Sequence`, since membership multiplicity is not supported.

append(object: T) : OrderedSet(T)

Treated equivalently to the `Set` operation `self->including(object)`, since ordering is not supported.

prepend(object: T) : OrderedSet(T)

Treated equivalently to the `Set` operation `self->including(object)`, since ordering is not supported.

insertAt(index : Integer, object : T) : OrderedSet(T)

Treated equivalently to the `Set` operation `self->including(object)`, since ordering is not supported.

reverse() : OrderedSet(T)

Treated as a no-op, since ordering is not supported.

OrderedSet

All supported operations of `OrderedSet` have already been discussed; the rest is unsupported, since the relational semantics of graph queries ignores ordering.

Sequence

All supported operations of `Sequence` have already been discussed; the rest is unsupported, since the relational semantics of graph queries ignores ordering.

A.4 Iterator Expressions on Collection Types

Note that the general iterator expression `iterate`, unsupported by the mapping, is missing from the list below as it is not actually a part of the OCL Standard Library.

any(iterator : T | body : Boolean) : T

Mapped to P_O^+ as the unification of P_{self} and P_{body}^+, augmented with
`iterator==self`
and
`y==self`.

closure(iterator : T | body : T) : Collection(T)

Mapped to P_O as a disjunction of two bodies. The first body is the augmentation of P_{self} with `y==self`. The second body is the augmentation of P_{self} with three constraints: irreflexive transitive closure constraint `find` P_{body} `+`,
`iterator==self`
and
`y==body`.

collect(iterator : T | body : T2) : Collection(T3)

Mapped to P_O as the unification of P_{self} and P_{body}, augmented with
`iterator==self`
and
`y==body`.

Note that if the source subexpression is a `Set` or `OrderedSet`, and furthermore `body` is either single-valued or evaluates to either a `Set` or an `OrderedSet` (often true for navigation), then the only way for the resulting collection to have multiple membership is if the body subexpression evaluates to the same result for multiple values of the iterator. This means that if the iterator variable is promoted to a pattern parameter of P_O (in addition to inputs and output), then the match set relation of the resulting pattern will have exactly as many matches as the size of the OCL collection (including duplicates). Therefore in this case the multiple membership of `Bag` and `Sequence` collections can be represented faithfully using the relational semantics of graph patterns. This remains true even if further similar `collect()` steps are performed, or the result is filtered by `excludes()`, `selectByKind()` operations and `select()`, `reject()` iterators. With this workaround, multiplicity-dependent operations `size()`, `count()` and `sum()` can be correctly translated for these `Bag` and `Sequence` collections.

collectNested(iterator : T | body : T2) : Collection(T2)

Treated equivalently to `collect()` if `body` is single-valued; otherwise unsupported due to nested collections.

exists(iterator : T | body : Boolean) : Boolean

Mapped to P_O^+ as the unification of P_{self} and P_{body}^+, augmented with
`iterator==self`
and
`y==self`.

forAll(iterator : T | body : Boolean) : Boolean

Mapped to P_O^- as the unification of P_{self} and P_{body}^-, augmented with
`iterator==self`
and
`y==self`.

isUnique(iterator : T | body : T2) : Boolean

Mapped to P_O^- as the unification of P_{self}, P_{self2}, P_{body} and P_{body2}, augmented with the four constraints
`iterator==self`,
`iterator2==self2`,
`iterator!=iterator2`
and

`body == body2;`
where `self2` is a copy of the source subexpression expressed on the same input variables, and `body2` is a copy of the iterator body subexpression expressed on the same input variables but on iterator variable `iterator2`.

one(iterator : T | body : Boolean) : Boolean

Treated exactly as the equivalent OCL expression
`self->select(iterator | body)->size() = 1`.

reject(iterator : T | body : Boolean) : Collection(T)

Mapped to P_O as the unification of P_{self} and P^{-}_{body}, augmented with
`iterator==self`
and
`y==self`.

select(iterator : T | body : Boolean) : Collection(T)

Mapped to P_O as the unification of P_{self} and P^{+}_{body}, augmented with
`iterator==self`
and
`y==self`.

sortedBy(iterator : T | body : T2) : Collection(T)

Treated as a no-op, since ordering is not represented.

References

1. The Eclipse Foundation: Eclipse Modeling Framework. `http://www.eclipse.org/emf/`.
2. Eclipse Model Development Tools Project: MDT-OCL website (2011) `http://www.eclipse.org/modeling/mdt/?project=ocl`.
3. Ehrig, H., Engels, G., Kreowski, H.J., Rozenberg, G., eds.: Handbook on Graph Grammars and Computing by Graph Transformation. Volume 2: Applications, Languages and Tools. World Scientific (1999)
4. Rensink, A.: Representing first-order logic using graphs. In: Proc. 2nd International Conference on Graph Transformation (ICGT 2004), Rome, Italy. Volume 3256 of LNCS., Springer (2004) 319–335
5. W3C SPARQL Working Group: SPARQL Query Language for RDF. Technical report, W3C (2008) `http://www.w3.org/TR/rdf-sparql-query/`.
6. Ákos Horváth, Gergely Varró and Dániel Varró: Generic search plans for matching advanced graph patterns. In: Proc. of the Sixth International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT 2007), Braga, Portugal, Electornic Communications of the EASST (March 31- Apr. 1 2007) 57–68
7. Veit Batz, G., Kroll, M., Geiß, R.: A first experimental evaluation of search plan driven graph pattern matching. In: Applications of Graph Transformations with Industrial Relevance. Volume 5088 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2008) 471–486

8. Giese, H., Hildebrandt, S., Seibel, A.: Improved flexibility and scalability by interpreting story diagrams. ECEASST **18** (2009)
9. Forgy, C.L.: Rete: A fast algorithm for the many pattern/many object pattern match problem. Artificial Intelligence **19**(1) (September 1982) 17–37
10. Ujhelyi, Z., Bergmann, G., Ábel Hegedüs, Ákos Horváth, Izsó, B., Ráth, I., Szatmári, Z., Varró, D.: EMF-IncQuery: An integrated development environment for live model queries. Science of Computer Programming (0) (2014) –
11. Hegedüs, Á., Horváth, Á., Ráth, I., Varró, D.: A model-driven framework for guided design space exploration. In: 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011), Lawrence, Kansas, USA, IEEE Computer Society, IEEE Computer Society (11/2011 2011)
12. Rensink, A., Distefano, D.: Abstract graph transformation. Electron. Notes Theor. Comput. Sci. **157**(1) (May 2006) 39–59
13. Baldan, P., Corradini, A., König, B.: Unfolding graph transformation systems: Theory and applications to verification. In: Concurrency, Graphs and Models. Volume 5065 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2008) 16–36
14. Heckel, R., Küster, J., Taentzer, G.: Confluence of typed attributed graph transformation systems. In: Graph Transformation. Volume 2505 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2002) 161–176
15. Winkelmann, J., Taentzer, G., Ehrig, K., Küster, J.M.: Translation of Restricted OCL Constraints into Graph Constraints for Generating Meta Model Instances by Graph Grammars. In: Proceedings of the Fifth International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT 2006). Volume 211., Amsterdam, The Netherlands, Elsevier Science Publishers B. V. (April 2008) 159–170
16. Anastasakis, K., Bordbar, B., Georg, G., Ray, I.: UML2Alloy: a challenging model transformation. In: Model Driven Engineering Languages and Systems. Volume 4735 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2007) 436–450
17. Object Management Group: Object Constraint Language Specification (Version 2.4). (2014) `http://www.omg.org/spec/OCL/2.4/`.
18. Bergmann, G., Ujhelyi, Z., Ráth, I., Varró, D.: A graph query language for EMF models. In: Theory and Practice of Model Transformations, Fourth International Conference, ICMT 2011, Zurich, Switzerland, June 27-28, 2011. Proceedings. Volume 6707 of Lecture Notes in Computer Science., Springer, Springer (2011) 167–182
19. Bergmann, G., Horváth, Á., Ráth, I., Varró, D.: A benchmark evaluation of incremental pattern matching in graph transformation. In: Proc. 4th International Conference on Graph Transformations, ICGT 2008. Volume 5214 of Lecture Notes in Computer Science., Springer, Springer (2008) 396–410
20. Ráth, I., Ábel Hegedüs, Varró, D.: Derived features for EMF by integrating advanced model queries. In: 8th European Conference on Modelling Foundations and Applications, Kgs. Lyngby, Denmark, Springer, Springer (06/2012 2012)
21. Bergmann, G.: Incremental model queries in model-driven design. Ph.D. dissertation, Budapest University of Technology and Economics, Budapest (10 2013)
22. Bergmann, G.: Graph patterns from OCL: a performance evaluation. (03 2014) `https://incquery.net/content/graph-patterns-ocl-performance-evaluation`.
23. Bergmann, G., Ráth, I., Szabó, T., Torrini, P., Varró, D.: Incremental pattern matching for the efficient computation of transitive closure. In: Sixth Interna-

tional Conference on Graph Transformation. Volume 7562/2012., Bremen, Germany, Springer, Springer (09/2012 2012) 386–400

24. Groher, I., Reder, A., Egyed, A.: Incremental consistency checking of dynamic constraints. In: Fundamental Approaches to Software Engineering (FASE 2009). Volume 6013 of Lecture Notes in Computer Science., Springer (2010) 203–217
25. Brucker, A.D., Doser, J., Wolff, B.: Semantic issues of OCL: Past, present, and future. In: 6th OCL Workshop at the UML/MoDELS Conference. (2006)
26. Uhl, A., Goldschmidt, T., Holzleitner, M.: Using an OCL Impact Analysis Algorithm for View-Based Textual Modelling. In: Proc. 11th workshop on OCL and Textual Modelling (OCL 2011). Volume 44 of ECEASST. (2011)
27. Cabot, J., Clarisó, R., Riera, D.: Verification of UML/OCL class diagrams using constraint programming. In: Proceedings of the 2008 IEEE International Conference on Software Testing Verification and Validation Workshop. ICSTW '08, Washington, DC, USA, IEEE Computer Society (2008) 73–80
28. Cabot, J., Teniente, E.: Incremental integrity checking of UML/OCL conceptual schemas. J. Syst. Softw. **82**(9) (2009) 1459–1478
29. Reder, A., Egyed, A.: Incremental consistency checking for complex design rules and larger model changes. In: Model Driven Engineering Languages and Systems. Volume 7590 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2012) 202–218
30. Bergmann, G., Ákos Horváth, Ráth, I., Varró, D., Balogh, A., Balogh, Z., Ökrös, A.: Incremental evaluation of model queries over EMF models. In: Model Driven Engineering Languages and Systems, 13th International Conference, MODELS'10, Springer, Springer (10/2010 2010)
31. Izsó, B., Szatmári, Z., Bergmann, G., Horváth, Á., Ráth, I.: Towards precise metrics for predicting graph query performance. In: 2013 IEEE/ACM 28th International Conference on Automated Software Engineering (ASE), Silicon Valley, CA, USA, IEEE, IEEE (11/2013 2013) 412–431