# OCL as a Constraint Generation Language

Bastian Ulke and Friedrich Steimann

Lehrgebiet Programmiersysteme
Fernuniversität in Hagen
bastian.ulke@feu.de, steimann@acm.org

**Abstract.** In programming as well as in modelling, artefacts are required to comply with the rules of well-formedness given by their underlying language definition or their meta model, respectively. In many cases (e.g. the UML specification or the meta models implemented with EMF), these rules are given as Boolean typed OCL expressions, whose evaluation results indicate whether an artefact is well-formed. In this work, we present a set of transformation rules that allows for the direct use of OCL assertions as input for the generation of a Constraint Satisfaction Problem (CSP) representing a particular user activity, i.e., a CSP in which some constraint variables do not (yet) have values. The constraint generation process is sensitive of the activity's variability in that immutable parts are evaluated at constraint generation time, thus reducing the generated CSP's complexity. Our previous work on constraint based programming and modelling tools (allowing for example behaviour-preserving refactoring and well-formedness preserving completions or changes) can thus be applied directly to instances of an arbitrary language or meta model, as long as its invariants are specified in OCL.

## 1 Introduction

The Object Constraint Language (OCL) is commonly used to express assertions about object graphs, that is, instances of classes linked via attributes and references (together referred to as properties [5]). Although an assertion language, OCL has the look and feel of an object-oriented and functional expression language (specifically: a subset of Smalltalk). And indeed, OCL has fairly straightforward operational semantics: relational operators (=, <, etc.) and logical connectives (and, or, implies, etc.) are evaluated as usual, and quantifiers as well as Smalltalk-style collection iterators are evaluated in loops, accumulating values that represent the results.

On the surface OCL has only little to do with constraints in the usual sense, i.e., the constraints that constitute a constraint satisfaction problem (CSP). Unlike an OCL expression, a CSP, which consists of a set of variables (each with an associated domain) and a set of constraints on these variables, is not evaluated, but is rather solved, by assigning its variables values from their domains such that all constraints are satisfied, i.e., evaluate to true. Constraint solving is a search problem, involving constraint propagation and backtracking.

In previous work, we have shown how constraint solving can be used to implement certain development tools that offer controlled changes ("in place transformations") of an artefact to their users, such as refactoring, well-formed completion, or fixing ill-formedness [7–9]. These tools require that the artefact to be changed is translated into a CSP first. The variables of this CSP then represent properties of the artefact's elements, including those that are to be changed and those that, via constraints, restrict possible changes. The difficulty of getting the development tools right then reduces to producing the right constraints. This is usually done using constraint generating rules (called constraint rules for short), which are applied to the abstract syntax tree (AST) or meta model instance representing artefact and which are manually crafted after the language specification (which in turn may include OCL expressions).

In this paper, we show how OCL assertions (i.e., OCL invariants, preconditions, and postconditions)[1] can be interpreted as constraint generating rules or, more specifically, how a set of constraints that can be submitted to a constraint solver can be generated from the evaluation of an OCL assertion on an object graph. Differing from other, related work, our constraint generation is sensitive to the changeability of properties, which usually differs from application to application. This allows us to generate much fewer constraints (when compared to generating all) which, if the artefact for which the constraints are to be generated is large, is necessary for the constraint approach to be feasible.

## 2 Related Work

Although our target is very different, our work is technically related to UMLtoCSP [1] and its successor EMFtoCSP [3] (and hence indirectly related to some other work discussed in [3], which we do not discuss here).

In UMLtoCSP and EMFtoCSP, OCL invariants are translated to CSPs in order to let the constraint solver prove certain correctness properties of models (including their invariants), such as freeness of inconsistencies ("satisfiability") or freeness of redundancy. For this, models of increasing size are generated whose elements' properties (i.e., attributes and links between elements), represented as constraint variables, are constrained by constraints generated from the OCL invariants of the model. The solver then tries to find and assignment to the properties that satisfies all constraints and that thus represents a valid model instance. This amounts to a form of bounded verification (similar to model checking) which works because the sizes of the models are strictly limited (because of the size of the resulting search problem, maximum model size is usually rather small). By contrast, our approach starts with an existing model instance of arbitrary size (which may however be ill-formed, i.e., violate OCL invariants) and tries to find possible assignments for certain model elements only.

Constraints have been used for program analysis and compiler optimization for a long time [4]. More recently, it has been discovered as an effective means for implementing program refactoring tools, type-related ones in particular [10]. Our group has first extended the application to arbitrary declaration-related refactorings [7] and later from programming to modelling [8]. Here, the relationship between well-formedness rules and the constraint-generating rules required for constraint-based refactoring was

[1] OCL expressions can also evaluate to other data types (including objects), for instance to compute derived attributes. These expressions can also be interpreted as generating solver constraints, but are not the target of our research.

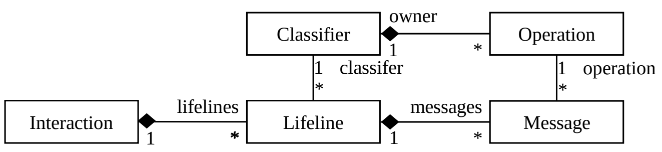


**Figure 1**: Meta model for OCL invariant `OpOwnerMatchesClassifier`

first described. More recently we extended the application of the constraint rules of the kind required for model refactoring to model completion and fixing [9]. Because behaviour can be neglected in these applications, the rules of well-formedness are sufficient. However, despite the fact that certain languages (such as UML) specify their well-formedness rules using OCL invariants, no attempt has been made so far to generate the constraints from OCL directly. The present paper reports on the current status of our work in this area.

## 3 Constraint Generation from OCL invariants

### 3.1 Conventional OCL evaluation vs. Transformation to a CSP

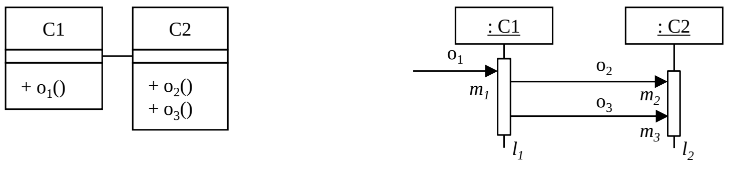
Figure 1 shows an exemplary meta model, for which OCL assertion (1) is assumed to be specified[2], together with an exemplary model instance. The assertion requires that if a lifeline has a classifier specified, each of its incoming messages that refers to an operation will refer to an operation of that classifier[3].

```
context Interaction inv OpOwnerMatchesClassifier:
  self.lifelines->forAll(l |
    l.classifier <> null implies l.messages->select(m |
      m.operation <> null
    )->forAll(m |                                          (1)
        m.operation.owner = l.classifier
    )
  )
```

An OCL interpreter, evaluating an assertion (i.e. a Boolean typed expression) against a particular artefact, typically descends down the expression's abstract syntax tree and evaluates the individual nodes from the bottom to the top until the root node is reached. In case of an assertion, the root node finally represents a Boolean value, indicating whether the expression holds for the artefact.

[2] Given the composition relationship between Lifeline and Message, the meta model implicitly specifies another constraint, requiring, that every message is referred to by exactly one instance of Lifeline. Due to space restrictions, we assume this constraint to be implemented implicitly.

[3] Note that due to inheritance, `m.operation.owner` could also be a superclass of `l.classifier`. However, we do not deal with inheritance here; the interested reader is referred to [10].



(a) (b)

**Figure 2**: Sample class and sequence diagram

Another way to evaluate such an expression is to not evaluate it directly, but to transform it into a (set of) logical expression(s), that consist(s) of variables, constants and operators and that are required to hold if and only if the iterative evaluation returns true. In contrast to the OCL assertion itself, the variables[4] used then represent an individual element's property (e.g. $l_1.classifier$), rather than an arbitrary element's one (e.g. `l.classifier` with `l` being a variable for any lifeline in `self.lifelines`). In case of a plain evaluation, all variables' domains only consist of a single value, namely that of represented property's current value. Together with the logical expressions (the constraints) generated, the variables and their domains constitute a constraint satisfaction problem (CSP), which can be submitted to a standard CSP solver. If all variables have singleton domains, the solver only verifies if the domains' single values satisfy the constraints, i.e., if the expression the constraints are generated from holds. The result is then equivalent with that of the conventional evaluation of the OCL assertion.

While the conventional evaluation of an assertion, as done by a common OCL interpreter, can only return the Boolean result value, specifying an equivalent constraint set has the advantage that single variables (i.e. properties of model elements) can be declared variable in the corresponding CSP. This is done by assigning it a domain containing all (or a subset of) the values of the property's type, each being a potential value for that property. The solutions, determined by the constraint solver then consist in value assignments for all variables ensuring that all constraints are satisfied, i.e., for which the original assertion holds. As we have shown in [9], this behaviour can be exploited to implement completion and fixing functionality, suggesting values for properties that ensure the artefact's well-formedness.

## 3.2 Transformation to a full constraint set

The challenge of this approach is to create constraints, which faithfully reflect the original expression's semantics. Relational expressions such as `m.operation <> null` can obviously be translated directly to a constraint such as $m_1.operation \neq null$. However, expressions operating on multi-valued properties (especially iterator expressions, e.g., `forAll` or `select`) are more difficult to translate. Some major challenges of the procedure are illustrated by transforming assertion (1) to a constraint set.

Since common solvers lack support for quantification, the call of `forAll` on `self.lifelines` has to be unrolled for any $l_i$ that could potentially be referred to by

[4] Here and in the following we use a computer font for OCL expressions, and a mathematical typesetting for the constraints generated from them.

self.lifelines. This is done by instantiating the body expression (i.e. the parameter expression to forAll) for $l_1$ and $l_2$, yielding the constraint stubs[5]:

$$\begin{aligned} l_1 \in self.lifelines \rightarrow (l_1.classifier \neq null \rightarrow \cdots) \\ l_2 \in self.lifelines \rightarrow (l_2.classifier \neq null \rightarrow \cdots) \end{aligned} \tag{2}$$

Note how the implications' conditions ensure that the assertion expressed in the outer forAll's body is only required for any $l_i$ contained in the set value of self.lifelines. In case $l_i \notin self.lifelines$, the OCL assertion (1) does not pose any requirement and so the constraints in (2) are true.

Different from the forAll iterator, select does not evaluate to a Boolean value, but to a set of objects. In the context of an assertion, i.e. an expression returning true or false only, its result can therefore only be argument to an outer expression e₀ (for example another operation or iterator call as in (1)). Consequently, the transformation of a select expression to a constraint requires providing the result value in a way it can be used for the transformation of e₀. Therefore, a temporary variable is introduced, which is required to only contain the subset of elements satisfying the select condition. The outer expression can then be transformed to a constraint by unrolling over all potential members of that accumulator variable guarded with an implication constraint, whose condition is that the particular member is a member of that variable. For instance, for the expression fragment l.message->select(.)->forAll(.) in (1), if l is instantiated with $l_1$, we get the constraint stub (3), which has to be created and conjoined for every $m_i$.

$$\begin{gathered} \big((m_i \in l_1.messages \wedge m_i.operation \neq null) \leftrightarrow m_i \in R_{select}\big) \\ \wedge (m_i \in R_{select} \rightarrow \cdots) \end{gathered} \tag{3}$$

More subtle than in these two iterator expressions is the complexity of transforming an expression such as m.operation.owner to a constraint. As stated in section 3.1, an element's (e.g. $m_1$) property (e.g. $m_1.operation$) directly translates to a variable; but here we have a property (…owner) reached by an indirection from another property (m.operation). Depending on the value of $m_1.operation$, $m_1.operation.owner$ can either evaluate to the value of the variables $o_1.owner$, $o_2.owner$ or, $o_3.owner$. Following the pattern of conditional constraints as above, the chained property access of the inner forAll iterator's body expression is translated to constraint stub (4), with l being instantiated with $l_1$ and m with $m_1$ [8]. Again, instances for all $o_i$ have to be conjoined.

$$m_1.operation = o_i \rightarrow o_i.owner = l_1.classifier \tag{4}$$

Although the single stubs are quite simple, their rollout and their nesting makes the resulting constraint set rather complex, consisting of expressions whose ASTs have a total of 82 terminals referring to 13 variables and 46 terminals referring to ten literals representing the model elements and the special null value[6].

The advantage of the full constraint set, as generated above, is that any property can be declared variable and the same constraint set can be used for finding a valid assignment for all of them. On the other hand, as we have seen [7], constraint size and

[5] The ellipsis represents a placeholder for the transformation of the expression following self.lifelines.

[6] Due to space restrictions, the complete resulting constraint set is not displayed here.

complexity is crucial to the solving time and thus to user acceptance. At the same time, in most development tools we have seen thus far (excluding [1, 3]) concrete applications only require distinguished properties to be variable. Creating the full constraint set therefore is in most cases not justified; instead, a generation process is required that systematically exploits constancy of particular properties by evaluating them at constraint generation time and generating constraints restricting the variable properties' values only.

## 3.3 Variability-sensitive constraint generation

A generation process that is sensitive to a distinguished variability of properties has to interleave evaluation of the constant parts of the expression (respectively its subexpressions) with the construction of constraints. The general idea of the variability-sensitive generation process is sketched with the help of some examples.

- If only the classifier property of all lifelines is variable, self.lifelines is constant, so that the condition $l_i \in self.lifelines$ of the constraint stub from (2) can be dropped. With $l_i\,.classifier$ being variable, the translation of the first forAll's body expression has to be translated directly, so that we get the constraint stubs

  $$l_1.classifier \neq null \rightarrow \cdots \text{ and } l_2.classifier \neq null \rightarrow \cdots \quad (5)$$

  To complete these constraints, the expression to the right side of the implies keyword needs transformation. As the messages property as well as the operation property are constant, the select operation can be evaluated to $\{m_1\}$ for $l_1$ and $\{m_2, m_3\}$ for $l_2$, so that the forAll can be translated by conjoining the respective $m_j.operation.owner = l_i.classifier$ constraints. However, in $m_j.operation.owner$, both referred properties are constant, so that they can be evaluated to $C_1$ for $m_1$ and $C_2$ for $m_2$ and $m_3$. With the tautology $A \wedge A = A$, we get the constraint set

  $$\begin{aligned} l_1.classifier \neq null \rightarrow C_1 = l_1.classifier \\ l_2.classifier \neq null \rightarrow C_2 = l_2.classifier \end{aligned} \quad (6)$$

  With four terminals referring to two variables and four terminals referring to three literals, the constraint set's complexity has strongly been reduced.
- The situation becomes more intricate, when considering the property operation to be variable for all messages: Neither can the value of the select calls be evaluated at constraint generation time, nor can the value of m.operation.owner, although the property owner is not variable. Furthermore, some algebraic optimizations allow for further simplification at the end.

  Just like in the former example, the implication stubs of (2) can be dropped, as self.lifelines is constant. As classifier is also a constant property which is not null for $l_1$ as well as for $l_2$, the expression to the left of the implies keyword is constantly true (and can thus also be dropped), so that the right side has to hold for both $l_i$. Constraint generation thus proceeds with transformation of l.messages->select(.), for which $l_i.messages$ can be constantly evaluated again, yielding $\{m_1\}$ for $l_1$ and $\{m_2, m_3\}$ for $l_2$. Compared to (3), the condition $m_j \in l_i.messages$ can therefore be evaluated at constraint generation time: If it is true, only the right side of the inner conjunction, $m_j.operation \neq null$, remains; if it is false; so is the conjunction, so that the right side of the equivalence requires negation ($m_k \notin R_{select;l_i}$ for some $k$):

$$\left((m_1.operation \neq null) \leftrightarrow m_1 \in R_{select;l_1}\right) \wedge \left(m_1 \in R_{select_{l_1}} \rightarrow \cdots\right) \tag{7a}$$

$$\left(m_2 \notin R_{select;l_1}\right) \wedge \left(m_2 \in R_{select;l_1} \rightarrow \cdots\right) \tag{7b}$$

$$\left(m_3 \notin R_{select;l_1}\right) \wedge \left(m_3 \in R_{select;l_1} \rightarrow \cdots\right) \tag{7c}$$

$$\left(m_1 \notin R_{select;l_2}\right) \wedge \left(m_1 \in R_{select;l_2} \rightarrow \cdots\right) \tag{7d}$$

$$\left((m_2.operation \neq null) \leftrightarrow m_2 \in R_{select;l_2}\right) \wedge \left(m_2 \in R_{select;l_2} \rightarrow \cdots\right) \tag{7e}$$

$$\left((m_3.operation \neq null) \leftrightarrow m_3 \in R_{select;l_2}\right) \wedge \left(m_3 \in R_{select;l_2} \rightarrow \cdots\right) \tag{7f}$$

The alert reader will have noted that in the conjunction $\left(m_k \notin R_{select;l_i}\right) \wedge \left(m_k \in R_{select;l_i} \rightarrow \cdots\right)$ the second term can be dropped as it is always true due to the expression of the first term. The remaining stubs require transformation of the subsequent expression, `m.operation.owner = l.classifier`, with the indirection of the chained property access being rolled out as in (4). With this, with $l_1.classifier = C_1$ and with the $o_n.owner$ being the operations' declaring classes, we get for (7a):

$$\begin{gathered}\left((m_1.operation \neq null) \leftrightarrow m_1 \in R_{select;l_1}\right) \\ \wedge \left( m_1 \in R_{select;l_1} \rightarrow \begin{pmatrix} m_1.operation = o_1 \rightarrow C_1 = C_1 \wedge \\ m_1.operation = o_2 \rightarrow C_2 = C_1 \wedge \\ m_1.operation = o_3 \rightarrow C_2 = C_1 \end{pmatrix} \right)\end{gathered} \tag{8}$$

A consideration of the inner conjunction now allows further evaluation: the implication for the case of $m_1.operation$ being $o_1$ is always true and can thus be dropped; for the cases of $o_2$ and $o_3$ the consequence $C_2 = C_1$ is constantly false. With $A \rightarrow B = \neg B \rightarrow \neg A$, we can simplify (8) to:

$$\begin{gathered}\left((m_1.operation \neq null) \leftrightarrow m_1 \in R_{select;l_1}\right) \\ \wedge \left( m_1 \in R_{select;l_1} \rightarrow \begin{pmatrix} m_1.operation \neq o_2 \wedge \\ m_1.operation \neq o_3 \end{pmatrix} \right)\end{gathered} \tag{9}$$

We finally get six constraints, consisting of 16 terminals referring to five variables and 13 terminals referring to six literals, which is still a considerable reduction.

To summarize, although the simple procedure for the generation of the full constraint set (being generic in that is supports variability of an arbitrary property) leads to considerable constraint complexity even for small examples, we have seen that the unchangeable properties' fixedness can be systematically exploited to simplify the generated constraints. By evaluating those parts of the OCL assertion that are not affected by any variable property, and by simplifying the resulting logical expressions according to the rules of Boolean algebra, we can reduce the number of terminals in the resulting constraint set, allowing for a faster solving.

After the exemplary discussion of individual optimization steps, the following section introduces a set of transformation rules for common OCL expression that individually reflect variability of its operands.

## 3.4 Transformation Rules

We define the rules per iterator type (node type of the abstract syntax tree), i.e., we provide rules for `forAll`, `select`, `collect` and `iterate` (see figure 3). For each involved subexpression (typically the source, i.e., the expression on whose result the iterator is

| EXPRESSION SOURCE | BODY EXPRESSION *variable* | *constant* |
|---|---|---|
| e₁->forAll(v\|e₂(v)) | | |
| *variable* | $\frac{v \in [e_1]}{v \in e_1 \rightarrow e_2(v)}$ | $\frac{v \in [e_1] \wedge \neg e_2(v)}{v \notin e_1}$ |
| *constant* | $\frac{v \in e_1}{e_2(v)}$ | $\frac{v \in e_1 \quad e_2(v)}{}$ |
| e₁->select(v\|e₂(v)) =: e | | |
| *variable* | $\frac{v \in [e_1]}{v \in e_1 \wedge e_2(v) \leftrightarrow v \in \mathrm{R}}$ | $\frac{v \in [e_1] \wedge e_2(v)}{v \in e_1 \leftrightarrow v \in R}$ $\frac{v \in [e_1] \wedge \neg e_{2(v)}}{v \notin R}$ |
| *constant* | $\frac{v \in e_1}{e_2(\mathrm{v}) \leftrightarrow v \in R}$ | $\frac{R \coloneqq \bigcup_{v \in e_1, e_2(v)} \{v\}}{}$ |
| e₁->collect(v\|e₂(v)) =: e | | |
| *variable* | $\frac{\mathrm{w} \in [e_2(v)]}{w \in R \leftrightarrow \bigvee_{\mathrm{v} \in [e_1]} v \in e_1 \wedge w \in e_2(v)}$ | $\frac{\mathrm{w} \in [e_2(v)]}{w \in R \leftrightarrow \bigvee_{v \in [e_1], w \in e_2(v)} v \in e_1}$ |
| *constant* | $\frac{\mathrm{w} \in [e_2(v)]}{w \in R \leftrightarrow \bigvee_{v \in e_1} w \in e_2(v)}$ | $\frac{R \coloneqq \bigcup_{v \in e_1} e_2(v)}{}$ |
| e₁->iterate(v; acc:=v₀\| e₂(v, acc)) | | |
| *variable* | $\frac{o_i \in [e_1]}{o_i \in e_1 \rightarrow R_i = e_2(o_i, v_{i-1})}$ $\frac{o_i \in [e_1]}{o_i \in e_1 \rightarrow R_i = R_{i-1}}$ | *depends on the structure of* $e_2$ |
| *constant* | $\frac{o_i \in e_1}{R_i = e_2(o_i, R_{i-1})}$ | *depends on the structure of* $e_2$ |

**Figure 3**: Transformation rules for selected OCL expressions

applied, and a body expression), we consider the case that it is fixed and that it is variable (typically giving four cases to consider). The result of an expression involving at least one variable subexpression is variable; otherwise, it is fixed. We write a rule as

$$\frac{generation}{solving}$$

in which $generation$ is replaced with expressions that typically introduce variables along with their domains and that can be evaluated at constraint generation time. These variables also occur in expressions that replace $solving$ and that serve as a pattern for the constraints to be generated. For constraint generation, $solving$ is instantiated by replacing its variables for every tuple of values that satisfies $generation$.
In $generation$, we write $[e]$ for the type of expression $e$ and $a \in [e]$ for an $a$ to be an instance of the type of $e$, whose extension is a set of objects $\{o_1, \dots, o_n\}$.

The four rules for forAll are to be read as follows (in pseudocode):

```
/* Let e1 -> forAll( e2) be the expression to generate constraints from. */
if e1 is variable then
  for all instances v of the type of e1
                                  /* i.e., for all v that could possibly be in e1 */
    if e2 is variable then
      generate the constraint
       "$v \in e_1 \rightarrow e_2(v)$"      /* i.e., if v is really in e1, than e2 has to hold on v */
    else /* i.e., e2 is constant */
      if e2 does not evaluate to true on v, then
                                  /* i.e., forAll will not be true if v is really in e1 */
       generate the constraint
       "$v \notin e_1$"                 /* i.e., avoid that v becomes member of e1 */
else /* i.e., e1 is constant */
  for all members of e1    /* i.e., for all v that are in e1 */
    if e2 is variable then
      generate the constraint
       "$e_2(v)$"                 /* i.e., ensure, that e2 holds on v */
    else /* i.e., e2 is constant */
      evaluate e2 on v at constraint generation time
```

Note that, as stated above, the result of the expression is variable (i.e., is computed by the constraint solver) if $e_1$ or $e_2$ are variable; otherwise, it is constant and computed as usual, i.e., by conjoining the results of $e_2(v)$ for all v in $e_1$ (i.e., no constraint is generated, and the expression is fully evaluated at constraint generation time).

For $e_1$ -> select($e_2$), the rules are analogous. The main difference here is (as explained above) that a select expression does not evaluate to a Boolean (and as such cannot directly be translated to a constraint); instead, it evaluates to a subset of the objects of $e_1$. However, we can transform such an expression into a constraint by introducing a new constraint variable, $R$, and requiring that this variable equals the result of the expression. Like for forAll, the result can be computed during constraint generation if both subexpressions are constant; the OCL interpreter/constraint generator can simply construct $R$ by conjoining all $v \in e_1$ for which $e_2(v)$ holds. Transformation rules for reject can be derived by negating the selection predicate $e_2(v)$.

A similar strategy can also be used for the transformation of collect calls. In contrast to select and reject, however, collect generally does not produce a subset of the source expression but (a bag of) objects of arbitrary types, which can be reached from each element of the source expression via an arbitrary navigation path or operation call. The constraint variable $R$, representing the expression's result, is therefore restricted to contain only those elements w for which a v in the expression source $e_1$ exists so that w is in $e_2(v)$. If both $e_1$ and $e_2(v)$ are constant, $R$ can be evaluated at generation time again by conjoining $e_2(v)$ for all v in $e_1$.

The rule for iterate is generic for all iterators (including quantifiers). However, it differs in that the accumulator, which is an updatable variable in OCL, needs to be replaced by a series of constraint variables $v_i$, each one (except $v_0$) constrained to the value of its predecessor joined with the update operation. The result of the expression is then the value of the last $v_i$. Note that all OCL iterators that can be interpreted as special cases of iterate can be translated using this scheme; in the case of `forAll`, accumulation is implicit (all generated constraints must be satisfied); in the case of `select`, a single set-valued variable suffices as accumulator.

## 4 Future Work

This work presents our current approaches to interpreting OCL assertions as the source for constraint generation as we need it, e.g., for our generic model assist research [9]. A further challenge, for example, is to verify applicability of the given rules for further collection types besides set, such as bag and sequence. As most solvers do not support all of them, emulation is required; for instance, in the case of the Choco solver [2] (which we have used in our previous work), `Set` is the only collection type directly supported. Still, bags and sequences can be represented as sets of pairs of a member and an integer which allows being optimistic. Finally, we plan to enhance existing model editors with assist functionality based on the OCL invariants that the meta modeller specified.

## 5 Summary and Conclusion

In this paper, we have shown how OCL invariants can be transformed to constraints amenable to constraint solving. Differing from earlier work by others, we have made the transformation sensitive to the variability of constrained properties, saving us the generation of constraints that are not needed. Applications of our work are constraint-based refactorings [7, 8, 10] and other constraint-based development tools [9] that formerly required the formulation of constraint rules in a formalism specific to the tools; they can now rely on pre-existing OCL expressions, making the tools readily available for all languages whose well-formedness is specified using OCL. Further applications of our work seem to abound; it can be used in all areas in which OCL assertions are to hold after an update, for instance for change propagation and consistency preservation.

## References

1. J Cabot, R Clarisó, D Riera "UMLtoCSP: A tool for the formal verification of UML/OCL models using constraint programming" in: *Proc. of ASE* (2007) 547–548.
2. CHOCO Team *choco: an Open Source Java Constraint Programming Library* (Research Report 10-02-INFO, Ecole des Mines de Nantes, 2010).
3. CA González, F Büttner, R Clarisó, J Cabot "EMFtoCSP: A tool for the lightweight verification of EMF models" in: *Proc. of Formal Methods in Software Engineering: Rigorous and Agile Approaches* (FormSERA) (2012).
4. F Nielson, HR Nielson, C Hankin *Principles of Program Analysis* (Springer, 2005).
5. Object Management Group *Object Constraint Language* Version 2.2 (`http://www.omg.org/spec/OCL/2.2`).
6. J Palsberg, MI Schwartzbach *Object-Oriented Type Systems* (Wiley 1994).
7. F Steimann, C Kollee, J von Pilgrim "A refactoring constraint language and its application to Eiffel" in: *Proc. of ECOOP* (2011) 255–280.
8. F Steimann "Contraint-based model refactoring" in: *Proc. of MODELS* (2011) 440-454.
9. F Steimann, B Ulke "Generic model assist" in: *Proc. of MODELS* (2013) to appear.
10. F Tip, A Kiezun, D Bäumer "Refactoring for generalization using type constraints" in: *Proc. of OOPSLA* (2003) 13–26.