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

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

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

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

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

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

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):

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

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:

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

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*	BODY EXPRESSION *constant*
e_1->forAll(v|e_2(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	$\frac{v \in e_1}{e_2(v)}$	$\frac{v \in e_1 \quad e_2(v)}{}$
e_1->select(v|e_2(v)) =: e		
variable	$\frac{v \in [e_1]}{v \in e_1 \wedge e_2(v) \leftrightarrow v \in \mathrm{R}}$	$\frac{v \in [e_1] \wedge e_2(v)}{v \in e_1 \leftrightarrow v \in R} \quad \frac{v \in [e_1] \wedge \neg e_{2(v)}}{v \notin R}$
constant	$\frac{v \in e_1}{e_2(\mathrm{v}) \leftrightarrow v \in R}$	$\frac{R \coloneqq \bigcup_{v \in e_1} \{v\}}{e_2(v)}$
e_1->collect(v|e_2(v)) =: e		
variable	$\frac{\mathrm{w} \in [e_2(v)]}{w \in R \leftrightarrow \bigvee_{v \in [e_1]} v \in e_1 \wedge w \in e_2(v)}$	$\frac{\mathrm{w} \in [e_2(v)]}{w \in R \leftrightarrow \bigvee_{\substack{v \in [e_1] \\ w \in e_2(v)}} v \in e_1}$
constant	$\frac{\mathrm{w} \in [e_2(v)]}{w \in R \leftrightarrow \bigvee_{v \in e_1} w \in e_2(v)}$	$\frac{R \coloneqq \bigcup_{v \in e_1} e_2(v)}{}$
e_1->iterate(v; acc:=v_0| e_2(v, acc))		
variable	$\frac{o_i \in [e_1]}{o_i \in e_1 \rightarrow R_i = e_2(o_i, v_{i-1})} \quad \frac{o_i \in [e_1]}{o_i \in e_1 \rightarrow R_i = R_{i-1}}$	*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 $\{o_1, \ldots, o_n\}$.

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 ∈ e1 → e2(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 ∉ e1"              /* 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
    "e2(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.