Optimising OCL synthesized code

Jesús Sánchez Cuadrado[1]

Universidad de Murcia
jesusc@um.es

Abstract. OCL is a important element of many Model-Driven Engineering tools, used for different purposes like writing integrity constraints, as navigation language in model transformation languages or to define transformation specifications. There are refactorings approaches for manually written OCL code, but there is not any tool for the simplification of OCL expressions which have been automatically synthesized (e.g., by a repair system). These generated expressions tend to be complex and unreadable due to the nature of the generative process. However, to be useful this code should be as simple and resemble manually written code as much as possible.
In this work we contribute a set of refactorings intended to optimise OCL expressions, notably covering cases likely to arise in generated OCL code. We also contribute the implementation of these refactorings, built as a generic transformation component using bentō, a transformation reuse tool for ATL, so that it is possible to specialise the component for any OCL variant based on Ecore. We describe the design and implementation of the component and evaluate it by simplifying a large amount of OCL expressions generated automatically showing promising results. Moreover, we derive implementations for ATL, EMF/OCL and SimpleOCL.

Keywords: Model-Driven Engineering, Model Transformations, OCL, Refactoring

1 Introduction

OCL [25] is used in Model-Driven Engineering (MDE) in a wide range of scenarios, including the definition of integrity constraints for meta-models and UML models, as a navigation language in model transformation languages and as input for model finders, among others. The most usual scenario is that OCL constraints are written by developers who can choose their preferred style, and thus tend to write concise and readable code. An utterly different scenario is the automatic generation of OCL constraints. In this setting, the style of the generated constraints is frequently sub-optimal, in the sense that it may contain repetitive expressions, unnecessary constructs (e.g., too many let expressions), trivial expressions (e.g., $false = false$), etc. This is so since synthesis tools typically use templates (or sketches in program synthesis [28]) whose holes are filled by automatic procedures.

Faced with the task of writing a non-trivial OCL synthesizer, the implementor could try to design it in a way that favour the generation of concise and simple expressions, but this introduces additional complexity at the core of the synthesizer which can be hard to manage. An alternative is to generate the OCL constraints in the easiest way from the synthesizer's implementation point of view, and then have a separate simplifying process to handle this task. In this work we propose a catalogue of simplifications for OCL expressions, especially targeted to OCL code generated automatically. The simplifications range from well-known rewritings for integers and booleans to more specific ones related to type comparisons (i.e., oclIsKindOf). This work fills the gap between refactorings and conversions targeted to manually written code [6, 5] and the initial work by Giese and Larsson [19] about OCL simplifications.

On the other hand, there are several OCL implementations like EMF/OCL [17], SimpleOCL [31], the embedding of OCL in the ATL language [21], etc. Hence, commiting to a single variant would limit the practical applicability of the catalogue. To overcome this issue we have implemented it as a generic transformation component using bentō [8]. Bentō is a transformation reuse tool for ATL, which allows the development of generic transformations that are later specialized to concrete meta-models. In this paper we describe the design and implementation of this component and the main elements of the catalogue. Finally, we have evaluated the catalogue by applying it to a large amount of OCL expressions and specializing it for ATL, EMF/OCL and SimpleOCL in order to show its reusability

Altogether, this work presents the following contributions. (1) A new set of simplification refactorings for OCL, which complements the ones proposed in [19] and reuses some of ones described in [6, 26]. (2) A design based on the notion of generic transformation [8] which allows mapping one definition of the refactorings to several variants of OCL. (3) A working implementation (BeautyOCL) implemented with bentō[1], for which the ATL/OCL specialization has been integrated in anATLyzer[2], our IDE for ATL model transformations. The catalogue can be easily extended with new simplifications and specializations by submitting pull requests to the available GitHub repository.[3]

Organization. Section 2 presents related work, and Sect. 3 motivates the work through a running example. Section 4 introduces the framework used to develop the catalogue of simplifications, and Sect. 5 describes the catalogue. The work is evaluated in Sect. 6, and Sect. 7 concludes.

2 Related work

The closest work to ours was proposed by Giese and Larsson [19]. The motivation was to simplify constraints generated for UML diagrams in the context of design patterns. Simplifications for primitive types and collections are proposed by

[1] http://github.com/jesusc/bento
[2] http://anatlyzer.github.io
[3] http://github.com/jesusc/beautyocl

means of examples. More complex cases including conditionals, let expressions and the treatement of oclIsKindOf expressions are not handled. We depart from this work and propose a more extensive catalogue. Moreover, we have developed the catalogue using a reusable approach, with the aim of fostering its usage.

Correa et al. investigated the impact of poor OCL constructs on understandability [7], finding that refactored expressions are more understandable. The experiments were carried out on hand-written expressions, thus, it is likely that refactorings for expressions generated automatically have an even bigger impact on understandability. In [32], a catalogue of refactorings for ATL transformations is presented. Some of them are applicable to OCL, but they do not target simplifications. Moreover, the authors point out the possibility of implementing the refactorings in a language independent way, which is now achieved with our framework. The work of Correa and Werner presents a set of refactorings for OCL [6]. Some of them are of interest for our case, particularly refactorings for verbose expressions, while others are particularly useful for hand-written OCL expressions. A complementary work with additional refactorings is presented in [26]. Cabot and Teniente [5] proposes a set of transformations to derive equivalent OCL expressions. Some of these transformations are simplifications, but they generally focus on equivalent ways of writing a given OCL expression. Similarly, a set of optimizations patterns to improve the performance of OCL expressions in ATL programas is presented in [15]. Another source of related works is expression simplification rules developed with program transformation systems [22].

Regarding the applicability of our approach, it is targeted to complement tools which generate or transform OCL constraints. Some of them are based on filling in a pre-defined template from a given model [19, 29, 1]. Other works modify OCL expressions as a response to meta-model evolution [20]. These approaches could be benefited by our implementation. Nevertheless, given that our target is automatically generated code, it is specially well suited to complement approaches related to the notion of program synthesis and program repair. This is so since they tend to generate "alien code" [23] which may be problematic when humans need to maintain the generated code. To the best of our knowledge, there are only a few systems of this kind in the MDE and OCL ecosystem, like our work in quick fixing ATL transformations [11] and the generation or pre-conditions [24, 12, 14]. Hence, we believe that this work will be also valuable to complement OCL synthesis tools likely to appear in the future.

3 Motivation and running example

In this section we present the running example which will be used throughout the paper. We also use it to motivate the need for our catalogue of simplification refactorings. In previous work we implemented a large set of quick fixes for ATL transformations [11], with which it is possible to fix many types of non-trivial problems. This is integrated in the anATLyzer IDE [13] allowing users to automatically generate and integrate pieces of OCL code that fixes problems in

their transformations. From the usability point view the main concern of our tool was that the generated OCL expressions where often accidentally complex due to the automatic procedure used to generate them. In practice, this means that users may not use the quick fix feature because the OCL expressions which are automatically produced are unnecessarily too complex, difficulting its understanding. This problem is not exclusive of our approach, but it is acknowledged in other works [19][23].

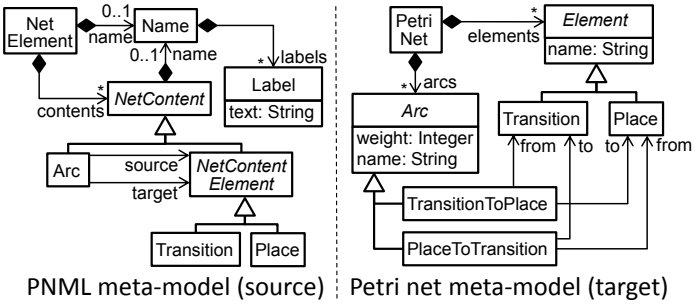

Fig. 1: Source and target meta-models of the running example.

To illustrate this issue we will use an excerpt of the PNML2PetriNet transformation, from the *Grafcet to PetriNet* scenario in the ATL Zoo[4], slightly modified to show interesting cases. Fig. 1 shows the source and target meta-models of the transformation and Listing 1 shows an excerpt of the transformation.

```
1  rule PetriNet {
2    from n : PNML!NetElement
3    to p : PetriNet!PetriNet (
4      elements ← n.contents,
5      arcs ← n.contents→select(e | e.oclIsKindOf(
           PNML!Arc))
6    )
7  }
8
9  rule Place {
10   from n : PNML!Place
11   to   p : PetriNet!Place ( ... )
12 }
13
14 rule Transition {
15   from n : PNML!Transition
16   to   p : PetriNet!Transition ( ... )
17 }
18
19 rule PlaceToTransition {
20   from n : PNML!Arc (
21     n.source.oclIsKindOf(PNML!Place) and
22     n.target.oclIsKindOf(PNML!Transition)
23   )
24   to p : PetriNet!PlaceToTransition (
25     "from" ← n.source,
26     "to" ← n.target
27   )
28 }
29
30 rule TransitionToPlace {
31   from n : PNML!Arc  (
32     -- The developer forgets to add n.source.
         oclIsKindOf(PNML!Transition)
33     n.target.oclIsKindOf(PNML!Place)
34   )
35   to p : PetriNet!TransitionToPlace (
36     "from" ← n.source, -- Problem here, n.
         source could be a Place
37     "to" ← n.target
38   )
39 }
```

Listing 1: Excerpt of the PNML2PetriNet ATL transformation.

[4] http://www.eclipse.org/atl/atlTransformations/

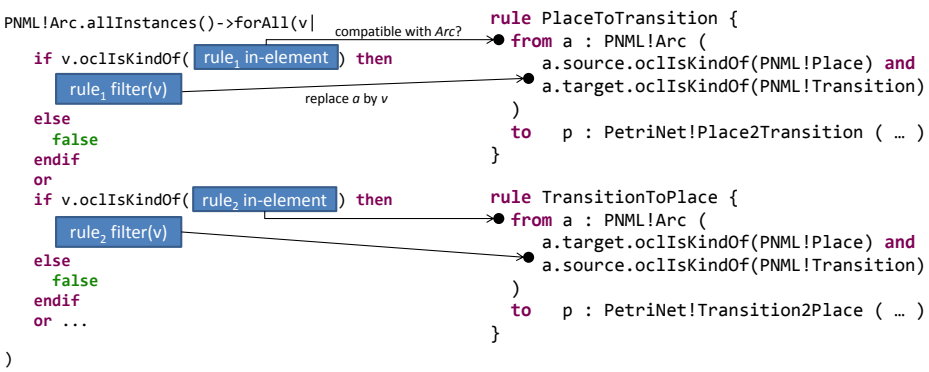

Fig. 2: Schema for constraint generation.

Consider the bug introduced in line 32 due to a missing check in the filter which enables the assignment of a Place object to a property of type Transition. A valid fix would be to extend the rule filter with not n.source.oclIsKindOf(PNML!Place). This is, in fact, what ANATLYZER generates by default since it just uses the typing of the from ← n.source binding (line 36) to deduce a valid fix. However, a simpler and more idiomatic expression would be n.source.oclIsKindOf(PNML!Transition).

Once fixed, we could be interested in generating a meta-model constraint for PNML to rule out invalid arcs (e.g., an arc whose source and target references point both to places (or both to transitions)). The implementation of quick fixes in ANATLYZER will generate a constraint like the one shown in Listing 2. The constraint is generated in the most general way, not taking into account the possible optimizations that could be made.

```
1  Arc.allInstances()→forAll(v1 |
2    if v1.oclIsKindOf(Arc) then
3      v1.source.oclIsKindOf(Transition) and v1.target.oclIsKindOf(Place)
4    else false endif or
5    if v1.oclIsKindOf(Arc) then
6      v1.source.oclIsKindOf(Place) and v1.target.oclIsKindOf(Transition)
7    else false endif
```

Listing 2: Automatically generated invariant to rule out invalid arcs in a Petri net

In general, a synthesizer uses a template and tries to fill the holes using some automated procedure. Fig. 2 shows the schema to generate pre-conditions used in ANATLYZER. To generate a constraint that will be attached to the PNML meta-model, our system would identify all rules dealing with Arc elements, that is, any rule whose input pattern has Arc or one of its subtypes (if any). Then, it would merge rule filters replacing occurrences of the a variable defined in the input pattern of the rules with the iterator variable v. Please note that this schema based on "if-then-else" is cumbersome, but it is necessary because there is no short-circuit in OCL and therefore a simple and expression would not work in the general case. Hence, our goal is to simplify these kind of expressions into more idiomatic code, as shown in the Listing 3.

```
1  Arc.allInstances()→forAll(v1 |
2    (v1.source.oclIsKindOf(Transition) and v1.target.oclIsKindOf(Place)) or
3    (v1.source.oclIsKindOf(Place) and v1.target.oclIsKindOf(Transition))
```

Listing 3: Simplified invariant to rule out invalid arcs in a Petri net

In the rest of this paper we present our approach to beautify OCL code, in particular targeting automatically synthesized OCL code. As we will see, it is expected that such code has unnecessary complexity, contains repetitive expressions and it is many times difficult to read. The next section describes the framework and the following presents the current catalogue.

4 Framework

This section describes the design of the reusable component to simplify OCL expressions. We have designed the catalogue of simplifications as a set of reusable transformations using the notion of concept-based transformation components [8]. Our aim is to deal with the fact that there are several implementations of OCL which could be benefited from automatic simplifications. In the EMF ecosystem, we can find the standard OCL distribution (EMF/OCL), SimpleOCL, OCL embedded in the ATL language, the OCL variant of Epsilon, etc. These implementations are incompatible among each other, due to a number of reasons, including different representations of the abstract syntax tree, questions related to the integration of OCL in another language, different OCL versions, different supported and unsupported features (e.g., closure operation is not supported in ATL), access to typing information, etc.

4.1 Overview

Our framework is based on the notion of *concept* and generic transformation component. A concept is a description of the structural requirements that a meta-model needs to fulfil to allow the instantiation of the component with a concrete meta-model (e.g., a particular OCL implementation in this case). A generic transformation component consists of a transformation template and one or more concepts. To instantiate the component for a specific meta-model, a binding describing the correspondences between the meta-model and the concept is written, which in turn induces an adaptation of the template to make it compatible with the meta-model. Please note that the approach of rewriting the original transformation is more adequate than using a pivot meta-model plus a transformation since the OCL expressions need to be modified in-place.

Figure 3 shows the architecture of the solution, which is technically implemented using the facilities provided by bentō to develop reusable transformation components [10]. In this design the simplification component has a set of small transformations, each one targeting only one kind of simplification. On the contrary to previous approaches which assumed one concept per transformation [27, 8], in this work all transformations share a common OCL-based concept plus two additional concepts to enable parameterized access to type information and

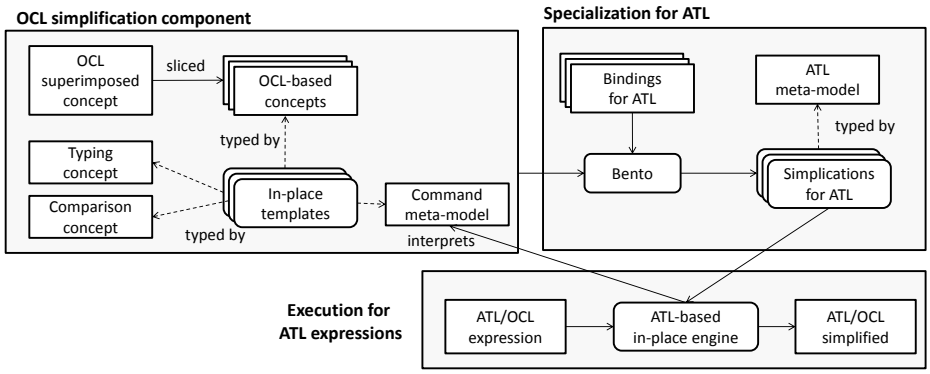

Fig. 3: Architecture of the generic component and its application to simplify ATL/OCL expressions.

expression comparison facilities (see Section 4.2). The output is a set of rewriting commands, which will be intepreted by a custom in-place engine. Given a specific OCL implementation for which we want to reuse the simplification component, we must implement a binding between the concrete OCL meta-model and the OCL concept meta-model. The binding establishes the correspondences between the concrete language meta-model (ATL/OCL in the figure) and the OCL concept. The bentō tool takes the binding and the component and derives a new simplification component specialised for ATL. This is fed into the in-place engine to apply the simplifications to concrete ATL expressions.

4.2 Transformation templates

We use ATL as our implementation language to develop the transformation templates. The in-place mode of ATL is quite limited, and it is not adequate to perform the rewritings required to implement our catalogue. Thus, we extended the in-place capabilities of ATL by creating a simple command meta-model to represent rewriting actions, which is later interpreted by a custom in-place engine. The rationale of choosing ATL despite of its limitations for in-place transformations is due to practical matters. First, we wanted to reuse the infrastructure provided by bentō, which currently supports ATL as the language to develop templates. Secondly, given the motivation of integrating the simplications within anATLyzer it seems logical to use ATL to avoid extra dependencies. Finally, Henshin [2] was also considered but developing rewritings like the ones of this work is not very natural either, since one needs to specify every possible container type of an expression that is going to be replaced, so that the replacement action can be "statically" computed. Other languages like Viatra [30] or EOL [18] have action languages which are imperative which complicates its integration with bentō (i.e., it is not a declarative language and thus is is difficult to write a HOT for it, which is the main mechanism used by bentō to adapt transformations).

Listing 4 shows a simplification rule written in ATL. An if expression like if true then thenExp else elseExp endif is rewritten to thenExp. The execution of the rule creates a Replace command which indicates which element (source) needs to be substituted by which element (target).

```
1  helper context OCL!OclExpression def: isTrue() : Boolean = false;
2  helper context OCL!BooleanExp    def: isTrue() : Boolean = self.booleanSymbol;
3
4  rule removeIf {
5    from o : OCL!IfExp ( o.condition.isTrue() )
6    to a : ACT!Replace
7    do {
8      a.source ← o;
9      a.target ← o.thenExpression;
10   }
11 }
```

Listing 4: Simplification rule

After the execution of the transformation our in-place transformation engine interprets and applies replacement commands over the source model. If there are no applicable actions, another transformation of the catalogue is tried. Thus, the in-place engine works by executing transformations and evaluating commands using an iterative, as-long-as-possible algorithm. We support commands for replacing elements, cloning and modifying pieces of abstract syntax tree and setting properties. This simple approach is enough for our implementation needs. All the transformations are executed in a pre-defined order, and termination has to guaranteed by ensuring that the generated commands only reduce the given expression. In this sense, it is possible to extend anATLyzer to enforce this property, which is part of our future work.

4.3 Concept design

The transformation template is a regular ATL transformation, typed against Ecore meta-models which act as transformation concepts. A key element in a generic component is the design of such concepts. Our framework requires three concepts, which are depicted in Figure 4. The *OCL concept* represents the elements of the OCL language which will be subject to simplifications. The *Typing* concept provides a mechanism to access typing information for OCL expressions, whereas the *Comparison* concept provides a way to determine if two expressions are equal. These latter two concepts are hybrid concepts, as defined in [16], since they provide hook methods which will be implemented by each specialization.

OCL concept. A concept should contain only the elements required by the transformation template. This is intended to facilitate its binding when it is going to be reused and to remove unneccesary complexity from the transformation template implementation. However, if we strictly use this approach to implement the catalogue, we would have many transformations whose concepts have many shared elements. For example, all simplification transformations which use operators would need to define a new OperatorCallExp class. It is thus impractical to build each concept separately. Moreover, it would require to have as many bindings as reused transformations. Therefore, we have designed a superimposed

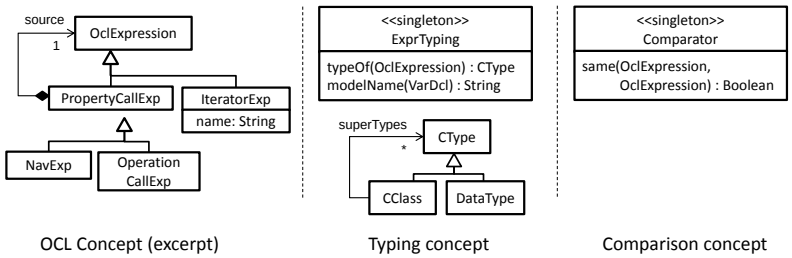

Fig. 4: Concepts used in the simplification component.

concept which contains all the elements required by the transformations of the catalogue. From this concept we automatically extract the minimal concept of each transformation using the approach described in [9], so that each individual rewriting could be used in isolation if needed. Please note that the superimposed OCL concept (i.e., it merges all the concepts used by the individual rewritings) do not necessarily need to be exactly like the OCL specification, but it may have less elements which are not handled by the simplifications (e.g., the property name in a navigation expression is irrelevant, while the name of an iterator is important). The OCL concept currenly implemented contains only 20 classes and 27 features. This is much smaller than the 85 classes of the ATL meta-model and the 54 classes of the EMF/OCL meta-model.

Typing concept. There are a number of transformations in the catalogue which require access to the types of the abstract syntax of the OCL expression. One alternative would be to extend the OCL concept with elements to represent typing information. However, this approach is not flexible enough since it assumes that concrete OCL meta-models have their expressions annotated with types. An alternative design is to have a separate concept with operations to retrieve the typing information. Each concrete binding is in charge of providing access the typing information computed by underlying OCL type checker. This design is, to some extent, similar to the idea of mirrors [3].

Comparison concept. The comparison concept is also a hybrid concept, but it addresses the problem of comparing two OCL expressions to determine if they are equivalent. The concept does not prescribe any mechanism to compare the expressions, but the implementations may decide to use simple approaches (e.g., comparing string serializations) or more complex ones (e.g., clone detection). The only requirement is that it must be reliable, in the sense that it cannot be heuristic.

5 Catalogue

This section describes through examples the most relevant simplifications currently implemented in the catalogue. The catalogue has been created based on the author's experience building anATLyzer, but it can be easily extended as new needs arise from other tools.

5.1 Literal simplifications

This set of simplifications replaces operations over primitive values by their results. For instance, an operation like $1 < 0$ is replaced by $false$ by applying a constant folding simplification . This category also covers the simplification of collection expressions like Set { Set { 1 } }→flatten() ⇒ Set{1}.

It is worth noting that this kind of expressions will be rarely written by a developer, but are likely to appear in synthesized OCL code, hence the need for the catalogue in this setting.

5.2 Iterators

This set of simplifications deals with iterator expressions which can be removed or whose result can be computed at compile time. The following listing shows the three simplifications implemented up to now. The simplifications for select also apply to reject just by swapping the behaviour of *True select* and *False select*.

Original

```
-- Unnecessary collect
Place.allInstances()→collect(p | p)→select(p | ...)
-- True select
Place.allInstances()→select(p | true)→collect(p | ...)
-- True forAll
Place.allInstances()→forAll(p | true)
```

Simplified

```
-- Unnecessary collect
Place.allInstances()→select(p | ...)
-- True select
Place.allInstances()→collect(p | ...)
-- True forAll
true
```

5.3 Noisy let expressions

Let expressions are useful when a large expression is used many times, otherwise it tends to introduce unnecessary noise. This simplification takes into account the size of the assigned expression and the number of usages in order to remove such let expressions. The following listing shows an example.

Original

```
let src = arc.source in
let tgt = arc.target in
  src.oclIsKindOf(PNML!Place) and
  tgt.oclIsKindOf(PNML!Transition)
```

Simplified

```
arc.source.oclIsKindOf(PNML!Place) and
arc.target.oclIsKindOf(PNML!Transition)
```

The main concern with this simplification is that it may break well-crafted code, when the developer intended to organize a set of logical steps into let variables. Thus, the simplification should only be applied for synthesized code which is known to generate repetitive let expressions.

5.4 Type comparison simplifications

These simplifications are aimed at removing unnecessary type comparisons using *oclIsKindOf* / *oclIsTypeOf* or to simplify a complex chain of type comparisons into a simpler one.

Remove if-then type comparison. An example of this simplification has been shown in Fig. 2 and Listing 3. If the condition *cond* of an if expression is a single

type comparison in the form *expr.oclIsKindOf(T)* we check if *typeOf(expr) = T*. In such case, we can safely replace the whole expression with *true*, which may enable other simplifications (see for instance *if-else elimination* below).

Full subclass checking to supertype. This simplification takes a chain of *or* expressions in which each subexpression checks the type over the same variable and tries to simplify it to a unique type check over a common supertype.

For example, the listing below (left) is intended to rule out arcs from the contents reference. This simplification recognizes that all subtypes of NetContentElement are checked, and the simpler n.oclIsKindOf(PNML!NetContentElement) can be used instead.

Original

```
aPetriNet.contents→select(n |
  n.oclIsKindOf(PNML!Place) or
  n.oclIsKindOf(PNML!Transition))
```

Simplified

```
aPetriNet.contents→select(n |
    n.oclIsKindOf(PNML!NetContentElement))
```

The application condition of this simplification is relatively complex to implement, hence the advantage of implementing it in a reusable module. A binary operator must be composed by only "or" sub-expressions and each subexpression must apply an oclIsKindOf operator to the same source expression. Then, we extract the set of types used as arguments of the oclIsKindOf operations (*types*). From this set we obtain the most general common supertype (*sup*) of all of these classes (if any), with the constraint that all subclasses of such supertype are "covered" by the classes in *types*, that is the following OCL constraint must be satisfied: sup.allSubclasses→forAll(sub | types→forAll(c | c = sub or c.superTypes→includes(sub)))

In the example, the most general supertype satisfying this constraint is NetContentElement. This is so because its set of subtypes is completely covered by Transition and Place. In contrast, NetContent is not a valid result because Arc is not in the set of types compared by the expression. One concern with this simplification is that for some cases explicitly checking the subtypes could be more readable than the simplified code, since it evokes more clearly the vocabulary of the transformation meta-model. An alternative is to parameterize the simplification with a threshold indicating the minimum number of "oclIsKindOf checks" that need to exists in the original code to trigger it.

5.5 Unshort-circuiting.

OCL does not have short circuit for boolean expressions. Thus, automatic synthesis procedures need to take special care to produce safe boolean expressions. For instance, the expression arc.source.oclIsKindOf(PNML!Place) and arc.source.tokens is unsafe because the tokens feature will be accessed regardless of the result of the first type comparison (i.e., if arc.source is a Transition). Hence, a runtime error will be raised. The usual solution is to write nested ifs, one for each boolean sub-expression, which typically leads to unreadable code. In the case of synthesized code the situation is exacerbated since it is likely that the synthesizer implementation always generate nested ifs to stay on the safe side.

For example, the following listing (left) shows a piece of code in which short circuit evaluation is not actually necessary because the name feature is defined in a superclass of Place. Therefore, it can be simplified as shown in the right part of the listing.

Original

```
if arc.source.oclIsKindOf(PNML!Place) then
  if arc.source.name <> OclUndefined then
    'plc' + arc.source.name
  else
    'no−name'
  endif
else
  'no−name'
endif
```

Simplified

```
if arc.source.oclIsKindOf(PNML!Place) and
   arc.source.name <> OclUndefined then
   'plc' + arc.source.name
else
  'no−name'
endif
```

Please note that this simplification makes use of the Comparison concept to be able reason more accurately about what can be simplified and the Typing concept to check typing correctness. Another variatns of this simplification can also be implemented, for instance for checking OclUndefined conditions.

5.6 Conditionals

Remove dead if/else branch. This is the most basic simplification of conditionals. Given a true or a false literal in the condition, the corresponding then or else parts are used to replace the conditional in the AST. For instance, if true then 'a' else 'b' endif can be simplified to 'a'.

Remove equals condition and then expression. A simple but useful simplification is recognizing that the condition and the "then" branch (or else branch) of an if expression are the same, and thus they always yield the same result.

Original

```
if place.tokens→size() = 1 then
  place.tokens→size() = 1
else
  false
endif
```

Simplified

```
place.tokens→size() = 1
−− If the else branch of the original expression
−− is true, then whole expression can be
−− replaced by true
```

If fusion. This simplification takes a binary operation between the results of two if expressions whose conditions are the same. In this case, it is safe to inline the then and else branches of the second expressions in the first one, as in the following example:

Original

```
if elem.oclIsKindOf(PN!Place) then
  elem.tokens→size() > 1
else false endif
and
if elem.oclIsKindOf(PN!Place) then
  not elem.name.oclIsUndefined()
else true endif
```

Simplified

```
if elem.oclIsKindOf(PN!Place) then
  elem.tokens→size() > 1 and
  not elem.name.oclIsUndefined()
else
  false and true
endif
```

The simplified version is more concise, and at the same time enables more simplification opportunities.

6 Evaluation

This section reports the evaluation of our approach. We have evaluated whether the simplifications are able to reduce the complexity of expressions synthesized automatically (usefulness) and to what extent it is possible to reuse the catalogue (reusability) for different OCL dialects, reflecting on the advantages and limitations of the approach.

6.1 Usefulness

We have applied the simplifications of the catalogue to two different kinds of automatically generated OCL constraints, both for the ATL variant of OCL. The first experiment consisted on simplifying OCL preconditions generated from target invariants of model transformations as described in [14]. We simplified 24 constraints coming from invariants defined in three transformations used by existing literature HSM2FSM, ER2REL and Factories2PetriNets. The second experiment applied the simplifications to the quick fixes generated by anATLyzer for the 100 transformations of the ATL Zoo, focussing on those quick fixes which generate rule filters, binding filters or pre-conditions since they are the most interesting in terms of complexity of the generated expressions. Table 1 summarizes the results of the experiments. The complete data, and the scripts and instructions to reproduce the experiments are available at the following URL: `http://sanchezcuadrado.es/exp/beautyocl-ecmfa18`.

	Pre-conditions				Quick fixes			
	#Simp.	**%**	**Avg.**	**Median**	**#Simp.**	**%**	**Avg.**	**Median**
Literals	6	1.4%	-	-	2397	36.5%	-	-
Iterators	6	1.4%	-	-	712	10.6%	-	-
Noisy let	0	0.0%	-	-	177	2.7%	-	-
Type comparison	38	8.6%	-	-	31	0.5%	-	-
Unshort-circuiting	362	82.3%	-	-	63	0.9%	-	-
Conditionals	28	6.6%	-	-	3182	48.5%	-	-
Total simplifications	404	100%	18.3	5	6562	100%	3.8	2
% of nodes removed by simplifications			19.3%	16.8%			33.5%	15.8%

Table 1: Summary of the results of the experiments.

For the preconditions, a total of 440 simplifications were applied to 24 expressions. In average, 18.3 simplifications were applied for each expression, however the median was 5 simplifications. This is because some expressions were particularly large and involved more simplifications. For instance, two of the expressions had more than 3000 nodes, which enabled the application of more than 150 simplifications for each one. In the quick fixes experiment a total of 6562 simplifications were applied to 1729 expressions. We express the simplification power of the catalogue (shown in the "% nodes removed by simplifications" row) by counting the number of nodes of the AST before and after the simplifications. In both experiments the obtained reduction is similar, around 20% in average and 16% in the median.

Regarding which simplification categories are more useful, the results are disparate. Some simplifications occur much more often in one experiment than in

the other. For instance, simplifications for literals and conditionals are very useful for quick fixes, whereas unshort-circuiting is more useful for pre-conditions. This suggests that simplifications are to some extent specific to the kind of generated code and the method used to generate such code.

At first glance some of the simplifications are quite simple, others are most complex (e.g., those based on the typing and comparison concepts). Combining all of them, the user gets a much better experience. For instance, the following listing shows the situation before and after the use of BeautyOCL. The simplifications applied has been the following: (1) replacing the oclIsKindOf operation by true, then (2) replacing the if expression by its condition and finally (3) simplifying the remaining `expr and expr` by `expr` where `expr = not i.hasLiteralValue.oclIsUndefined()`. As can be observed the result is much more readable. In other evaluated expressions the results are not so "beauty", but the user would expect an even simpler expression. Nevertheless, the results are very promising, and it is expected to have very good results as the catalogue grows.

Original

```
if not i.hasLiteralValue.oclIsUndefined() then -- #2
  i.hasLiteralValue.oclIsKindOf(RDM!Literal) -- #1
else
  false
endif
and not i.hasLiteralValue.oclIsUndefined() -- #3
```

Simplified

```
not i.hasLiteralValue.oclIsUndefined()
```

The catalogue instantiated for ATL has been integrated into AnATLyzer through a dedicated extension point, so that the generated quick fixes are automatically simplified. Moreover, a quick assist to let the user simplify a piece of expression on demand is also available. A screencast demonstrating this feature in more detail is available at `https://anatlyzer.github.io/screencasts/`.

Regarding threats to validity, the main threat to the internal validity of these experiments is that we have only used code synthesized by AnATLyzer. The main reason is the lack of availability of similar tools for other OCL variants. Another issue is that we use the number of nodes to measure the improvement of an expression after simplifications. This metric can be misleading sometimes. For instance, the removal of let expressions generates a simpler expression, but it can introduce a few more nodes. A controlled experiment with final users is required to effectively assess this question. A threat to the external validity is the number of OCL variants reused. Variants like Epsilon or USE are not considered due to not using Ecore meta-models. This is so because our simplifications work at the abstract syntax level, specified with Ecore. Please note that many of them are complex transformations (e.g., use type information or compare sub-expressions) which cannot be addressed with text-based transformations.

6.2 Reusability

The catalogue of simplifications has been designed with reusability in mind in order to be able to easily instantiate the catalogue for a specific OCL variant.

To assess to what extent this is possible we have instantiated the component for ATL/OCL, EMF/OCL and SimpleOCL.

The catalogue was first tested and debugged by writing a binding to ATL. The binding was relatively straightforward. The binding for EMF/OCL was also simple except for one important issue. The designed concept expects that an OperatorExp has a name to identify the concrete operator. However, in EMF/OCL an operation is identified by a pointer to an EOperation defined in the standard OCL meta-model. Our binding for the target model (i.e., to support the creation of operator expressions) is not powerful enough to handle this natively. The solution to overcome this has been to extend the typing concept with a "setOperation" so that it is possible to programmatically find and assign the proper EOperation if needed. For SimpleOCL the main limitation is that it does not compute any typing information, and thus we could not reuse those simplifications making use of the typing concept. This means that the instantiated catalogue for SimpleOCL needs to be smaller.

Regarding the size of the implementations, the ATL transformation templates consists of 791 SLOCs, whereas the bindings for ATL, EMF/OCL and SimpleOCL are 38, 49 and 48 SLOCs respectively. The bindings are relatively simple mappings specifications. These figures provide some evidence of the advantage of building transformations as reusable components.

Altogether, the catalogue has proved useful to optimise OCL expressions in terms of their size, thus having simpler and perhaps more beautiful expressions. The effort invested in the creation of the catalogue is amortized by allowing multiple instantiations. Moreover, this work is also non-trivial case study of the application of genericity techniques to model transformations, which can be a baseline to improve these techniques.

7 Conclusions

In this paper we have presented a catalogue of OCL simplifications for OCL expressions, which targets code which has been automatically generated. This catalogue has been implemented as a generic transformation component, with the aim of making it applicable to any OCL variant based on Ecore. The current implementation fully supports ATL and has also been partially instantiated for EMF/OCL and SimpleOCL. The evaluation shows that the proposed simplifications are useful and they can generally reduce the size of the expressions around 30%. As future works we plan to add new simplifications to the catalogue in order to be able to reduce generated expressions by anATLyzer even more. Also, we would like to extend bentō to allow using rewriting languages like Stratego [4] to develop the transformation templates for the generic transformation components. Another line of work is to reflect on how to optimise other kinds of MDE artefacts generated automatically, like models or meta-models.

Acknowledgements. Work funded by the Spanish MINECO TIN2015-73968-JIN (AEI/FEDER/UE).

References

1. J. Ackermann and K. Turowski. A library of ocl specification patterns for behavioral specification of software components. In *International Conference on Advanced Information Systems Engineering*, pages 255–269. Springer, 2006.
2. T. Arendt, E. Biermann, S. Jurack, C. Krause, and G. Taentzer. Henshin: advanced concepts and tools for in-place EMF model transformations. In *International Conference on Model Driven Engineering Languages and Systems*, pages 121–135. Springer, 2010.
3. G. Bracha and D. Ungar. Oopsla 2004: Mirrors: Design principles for meta-level facilities of object-oriented programming languages. *ACM SIGPLAN Notices*, 50(8):35–48, 2015.
4. M. Bravenboer, K. T. Kalleberg, R. Vermaas, and E. Visser. Stratego/xt 0.17. a language and toolset for program transformation. *Science of computer programming*, 72(1-2):52–70, 2008.
5. J. Cabot and E. Teniente. Transformation techniques for ocl constraints. *Science of Computer Programming*, 68(3):179–195, 2007.
6. A. Correa and C. Werner. Refactoring object constraint language specifications. *Software & Systems Modeling*, 6(2):113–138, 2007.
7. A. Correa, C. Werner, and M. Barros. An empirical study of the impact of ocl smells and refactorings on the understandability of ocl specifications. In *International Conference on Model Driven Engineering Languages and Systems*, pages 76–90. Springer, 2007.
8. J. S. Cuadrado, E. Guerra, and J. de Lara. A component model for model transformations. *IEEE Transactions on Software Engineering*, 40(11):1042–1060, 2014.
9. J. S. Cuadrado, E. Guerra, and J. de Lara. Reverse engineering of model transformations for reusability. In *International Conference on Theory and Practice of Model Transformations*, pages 186–201. Springer, 2014.
10. J. S. Cuadrado, E. Guerra, and J. de Lara. Reusable model transformation components with bentō. In *International Conference on Theory and Practice of Model Transformations*, pages 59–65. Springer, 2015.
11. J. S. Cuadrado, E. Guerra, and J. de Lara. Quick fixing atl transformations with speculative analysis. *Software & Systems Modeling*, pages 1–35, 2016.
12. J. S. Cuadrado, E. Guerra, and J. de Lara. Static analysis of model transformations. *IEEE Transactions on Software Engineering*, 2016.
13. J. S. Cuadrado, E. Guerra, and J. de Lara. Anatlyzer: An advanced ide for atl model transformations. In *40th International Conference on Software Engineering (ICSE)*. ACM/IEEE, 2018.
14. J. S. Cuadrado, E. Guerra, J. de Lara, R. Clarisó, and J. Cabot. Translating target to source constraints in model-to-model transformations. In *2017 ACM/IEEE 20th International Conference on Model Driven Engineering Languages and Systems (MODELS)*, pages 12–22. IEEE, 2017.
15. J. S. Cuadrado, F. Jouault, J. G. Molina, and J. Bézivin. Optimization patterns for ocl-based model transformations. In *International Conference on Model Driven Engineering Languages and Systems*, pages 273–284. Springer, 2008.
16. J. de Lara and E. Guerra. From types to type requirements: genericity for model-driven engineering. *Software & Systems Modeling*, 12(3):453–474, 2013.
17. Eclipse Modelling Framework. `https://www.eclipse.org/modeling/emf/`.
18. Epsilon. `http://www.eclipse.org/gmt/epsilon`.

19. M. Giese and D. Larsson. Simplifying transformations of ocl constraints. In *International Conference on Model Driven Engineering Languages and Systems*, pages 309–323. Springer, 2005.
20. K. Hassam, S. Sadou, V. Le Gloahec, and R. Fleurquin. Assistance system for ocl constraints adaptation during metamodel evolution. In *Software Maintenance and Reengineering (CSMR), 2011 15th European Conference on*, pages 151–160. IEEE, 2011.
21. F. Jouault, F. Allilaire, J. Bézivin, and I. Kurtev. ATL: A model transformation tool. *Science of Computer Programming*, 72(1-2):31 – 39, 2008. See also `http://www.emn.fr/z-info/atlanmod/index.php/Main_Page`. Last accessed: Nov. 2010.
22. D. B. Loveman. Program improvement by source-to-source transformation. *Journal of the ACM (JACM)*, 24(1):121–145, 1977.
23. M. Monperrus. A critical review of automatic patch generation learned from human-written patches: essay on the problem statement and the evaluation of automatic software repair. In *Proceedings of the 36th International Conference on Software Engineering*, pages 234–242. ACM, 2014.
24. J.-M. Mottu, S. S. Simula, J. Cadavid, and B. Baudry. Discovering model transformation pre-conditions using automatically generated test models. In *Software Reliability Engineering (ISSRE), 2015 IEEE 26th International Symposium on*, pages 88–99. IEEE, 2015.
25. OMG. Object Constraint Language (OCL), 2014. http://www.omg.org/spec/OCL/2.4/PDF.
26. J. Reimann, C. Wilke, B. Demuth, M. Muck, and U. Aßmann. Tool supported OCL refactoring catalogue. In *Proceedings of the 12th Workshop on OCL and Textual Modelling, Innsbruck, Austria, September 30, 2012*, pages 7–12, 2012.
27. L. Rose, E. Guerra, J. De Lara, A. Etien, D. Kolovos, and R. Paige. Genericity for model management operations. *Software & Systems Modeling*, 12(1):201–219, 2013.
28. A. Solar-Lezama, L. Tancau, R. Bodik, S. Seshia, and V. Saraswat. Combinatorial sketching for finite programs. *ACM Sigplan Notices*, 41(11):404–415, 2006.
29. C. Tibermacine, S. Sadou, C. Dony, and L. Fabresse. Component-based specification of software architecture constraints. In *Proceedings of the 14th international ACM Sigsoft symposium on Component based software engineering*, pages 31–40. ACM, 2011.
30. D. Varró, G. Bergmann, Á. Hegedüs, Á. Horváth, I. Ráth, and Z. Ujhelyi. Road to a reactive and incremental model transformation platform: three generations of the viatra framework. *Software & Systems Modeling*, 15(3):609–629, 2016.
31. D. Wagelaar. Simpleocl. `https://github.com/dwagelaar/simpleocl`.
32. M. Wimmer, S. M. Perez, F. Jouault, and J. Cabot. A catalogue of refactorings for model-to-model transformations. *Journal of Object Technology*, 11(2):2–1, 2012.