Software & Systems Modeling (2020) 19:1139–1161
https://doi.org/10.1007/s10270-019-00740-1

SPECIAL SECTION PAPER

A verified catalogue of OCL optimisations

Jesús Sánchez Cuadrado[1]

Received: 7 February 2019 / Revised: 13 June 2019 / Accepted: 20 June 2019 / Published online: 2 July 2019

Abstract
OCL is widely used by model-driven engineering tools with different purposes like writing integrity constraints for meta-models, as a navigation language in model transformation languages or to define transformation specifications. Another scenario is the automatic generation of OCL code 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 simple and resemble manually written code as much as possible when a developer must manually maintain it. There exists refactorings approaches for manually written OCL code, but there is no tool targeted to the optimisation of OCL expressions which have been automatically synthesised. Moreover, there is no available catalogue of OCL refactorings which can be integrated seamlessly into a tool. 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 catalogue using bentō, a transformation reuse tool for ATL. This makes it possible to specialise the catalogue for any OCL variant based on Ecore. Moreover, we propose a method to verify the correctness of the implemented catalogue based on translation validation and model finding. We describe the design and implementation of the catalogue and evaluate it by optimising a large amount of OCL expressions and proving the correctness of each optimisation execution. We also derive working implementations of the catalogue for ATL, EMF/OCL and SimpleOCL made available in a tool called BeautyOCL.

Keywords Model transformations · OCL · Refactoring · Verification

1 Introduction

OCL [39] is used in model-driven engineering (MDE) in a wide range of scenarios, such as the definition of integrity constraints for meta-models and UML models, to navigate models 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. A completely 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 [44]) whose holes are filled by automatic procedures. There are scenarios in which the generated code needs to be maintained manually by a developer, like IDE autocompletion facilities and automatic program repair. In these cases, a developer is likely to expect code which is simple and looks like manually written.

The problem faced by the implementor of a non-trivial OCL synthesiser is to address the trade-off between the maintainability of the synthesiser and the quality of the generated code. Thus, the implementor could try to design the synthesiser in a way that favours the generation of concise and simple OCL expressions, but this introduces additional complexity at the core of the synthesiser which can be hard to manage. An alternative is to generate OCL code in the easiest way from the synthesiser's implementation point of view, and then have a separate simplifying process to handle the

Communicated by Prof. A. Pierantonio, A. Anjorin, S. Trujillo, and H. Espinoza.

Work funded by project RECOM (Spanish MINECO, TIN2015-73968-JIN, AEI/FEDER/UE) and a Ramón y Cajal 2017 grant (RYC-2017-237) co-funded by MINECO (Spain) and the European Social Fund.

✉ Jesús Sánchez Cuadrado
jesusc@um.es

[1] Universidad de Murcia, Murcia , Spain

task of generating simple and readable code. Our experience developing tools that generate OCL code [11,12] indicates that the second alternative is better, because it allows us to decouple two different concerns, namely the synthesis procedure and the optimisation of the generated OCL expressions in this case. Moreover, this enable the reuse of the optimiser for another applications.

In this work, we propose a catalogue of optimisations for OCL expressions, especially targeted to OCL code generated automatically. The catalogue is implemented as a generic transformation component using bentō [19], and it has been specialised for ATL, EMF/OCL and SimpleOCL in order to show its reusability. We have evaluated the catalogue by applying it to a large amount of OCL expressions, showing its usefulness to reduce the complexity of automatically generated expressions. Moreover, the transformations in the catalogue have been verified to be correct with an approach based on translation validation [41] and SAT solving. This is, to best of the author's knowledge, the first set of ATL transformations for which a practical correctness verification procedure exists. Finally, there is a working implementation of the catalogue (BeautyOCL), which has been integrated in anATLyzer[1], our IDE for ATL model transformations. The catalogue can be easily extended with new simplifications and specialisations by submitting pull requests to the available GitHub repository.[2]

This work is based on [18], which has been extended as follows:

- An extended version of the catalogue. The new catalogue increases the number of applicable optimisations by 100% in the experiments.
- Detailed explanations, including details about the custom-made transformation engine used to execute the optimisations.
- A practical correctness verification method based on translation validation. Using this method, a number of bugs in the catalogue were identified and fixed.
- A new evaluation of the catalogue, now also including the verification of the correctness of the applied optimisations and a performance evaluation.

Organisation

Section 2 motivates this work through a running example. Section 3 describes the catalogue of optimisations, and Sect. 4 introduces the framework used to develop the catalogue as a reusable component. Section 5 presents the method used to verify the catalogue. The work is evaluated in Sect. 6, and Sect. 7 presents related work. Finally, conclusions are presented in Sect. 8.

[1] http://anatlyzer.github.io.

[2] http://github.com/anatlyzer/beautyocl.

2 Motivation and running example

The context of this work is program synthesis tools for model-driven engineering which may generate sub-optimal OCL code. An instance of this kind of tools is anATLyzer, the IDE [13] for ATL transformations that we have implemented. It features a static analyser powered by constraint solving which has a high degree of precision [11], plus a quick fix facility which allows users to automatically generate and integrate pieces of OCL code which fix non-trivial problems in their transformations [14]. anATLyzer can also verify that a transformation satisfies a set of OCL invariants by generating source OCL pre-conditions [12]. From the usability point view, the main concern of our tool was that the generated OCL expressions were 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, making their understanding difficult. This problem is not exclusive to our approach, but it is acknowledged in other works [25,35]. Thus, the results of this work are directly applicable to any OCL-based synthesis tool. This includes tools for model transformation by example [50], repairing of MDE artefacts [14] or the generation of pre-conditions [36]. Besides, with the emergence of search-based software engineering approaches, the applications of this work are expected to increase [23].

In the following we present the running example which will be used throughout the paper. We will use an excerpt of the PNML2PetriNet transformation, from the *Grafcet to PetriNet* scenario in the ATL Zoo[3], slightly modified to show interesting cases. Figure 1 shows the source and target metamodels of the transformation, and Listing 1 shows an excerpt of the transformation.

```
rule PetriNet {
  from n : PNML!NetElement
  to p : PetriNet!PetriNet (
    elements ← n.contents,
    arcs ← n.contents→select(e | e.oclIsKindOf(PNML!Arc))
  )
}

rule Place {
  from n : PNML!Place
  to   p : PetriNet!Place ( ... )
}

rule Transition {
  from n : PNML!Transition
  to   p : PetriNet!Transition ( ... )
}

rule PlaceToTransition {
  from n : PNML!Arc (
    n.source.oclIsKindOf(PNML!Place) and n.target.oclIsKindOf(PNML!
       Transition)
  )
  to p : PetriNet!PlaceToTransition  (
    "from" ← n.source,
```

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

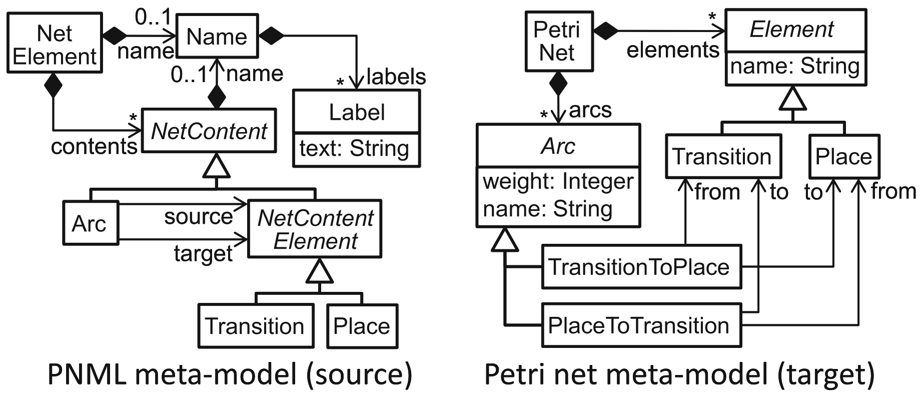


Fig. 1 Source and target meta-models of the running example

```
25        "to" ← n.target
26      )
27    }
28
29    rule TransitionToPlace {
30      from n : PNML!Arc (
31        -- The developer forgets to add n.source.oclIsKindOf(PNML!Transition)
32        n.target.oclIsKindOf(PNML!Place)
33      )
34      to p : PetriNet!TransitionToPlace (
35        "from" ← n.source, -- Problem here, n.source could be a Place
36        "to" ← n.target
37      )
38    }
```

Listing 1 Excerpt of the PNML2PetriNet ATL transformation.

Consider the bug introduced in line 31 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 35) 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 to places only or to transitions only). The implementation of quick fixes in anATLyzer will generate a constraint like the one shown in Listing 2. The constraint is generated without taking into account the possible optimisations that could be made. This means that the code generator may generate verbose code because its job is just to produce a piece of code that it is semantically correct. In the example, the conditionals checking v1.oclIsKindOf(Arc) are superfluous, but this is the general schema of the quick fix that produces correct code in all cases.

```
Arc.allInstances()→forAll(v1 |
  if v1.oclIsKindOf(Arc) then
    v1.source.oclIsKindOf(Transition) and v1.target.oclIsKindOf(Place)
  else
    false
  endif or
  if v1.oclIsKindOf(Arc) then
    v1.source.oclIsKindOf(Place) and v1.target.oclIsKindOf(Transition)
  else
    false
  endif
```

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

In general, a synthesiser uses a template and tries to fill the holes using some automated procedure. Figure 2 shows the schema to generate pre-conditions used in anATLyzer. To generate a constraint like the previous one, 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. 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 kinds of expressions into more idiomatic code, as shown in the Listing 3.

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

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

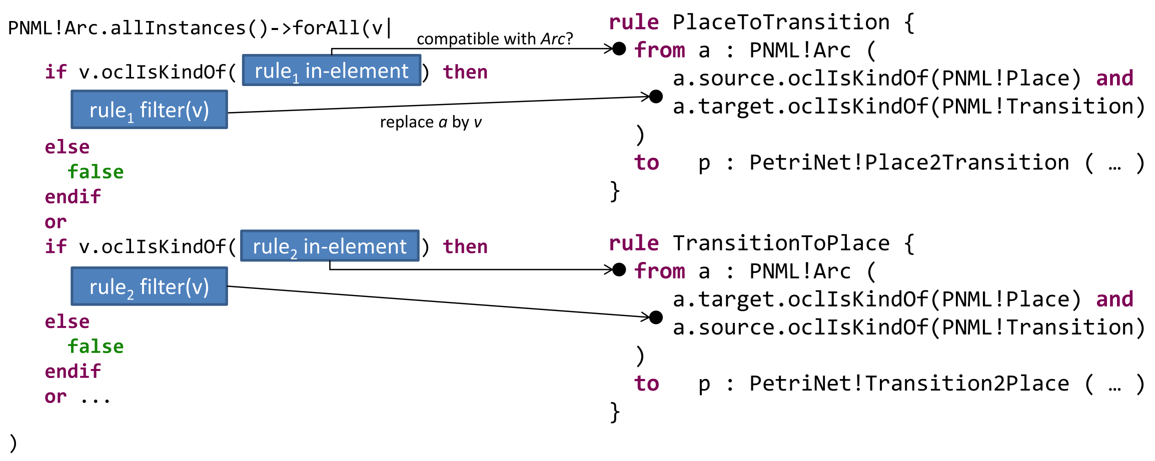
In the rest of this paper, we present our approach to optimise OCL code, in particular targeting automatically synthesised OCL code. As we will see, it is expected that such code has unnecessary complexity and contains repetitive expressions and it is many times difficult to read. The next section presents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

3 Catalogue of OCL optimisations

This section describes through examples the most relevant optimisations currently implemented in the catalogue. The examples are presented using excerpts of standard OCL code, independent of concrete OCL implementations.

3.1 Types of optimisations

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. The optimisations are targeted



```
PNML!Arc.allInstances()->forAll(v|
    if v.oclIsKindOf( rule1 in-element ) then
        rule1 filter(v)
    else
        false
    endif
    or
    if v.oclIsKindOf( rule2 in-element ) then
        rule2 filter(v)
    else
        false
    endif
    or ...

)
```

```
rule PlaceToTransition {
  from a : PNML!Arc (
       a.source.oclIsKindOf(PNML!Place) and
       a.target.oclIsKindOf(PNML!Transition)
  )
  to   p : PetriNet!Place2Transition ( … )
}

rule TransitionToPlace {
  from a : PNML!Arc (
       a.target.oclIsKindOf(PNML!Place) and
       a.source.oclIsKindOf(PNML!Transition)
  )
  to   p : PetriNet!Transition2Place ( … )
}
```

Fig. 2 Schema for constraint generation

towards OCL code which has been generated automatically, and their effect on it can be of three types, which are non-exclusive.

- *Simplifications*. This category corresponds to optimisations which reduce the size of the OCL expressions. Most of the optimisations fall into this category.
- *Idiomatic*. This corresponds to optimisations that generate code which looks more idiomatic, as if it was written by a human. The first example in the previous section falls into this category, since it is more idiomatic to write n.source.oclIsKindOf(Transition) than not n.source.oclIsKindOf(Place).
- *Performance*. Some optimisations may have a positive effect on performance (even if it is modest), whereas others may have a negative effect.

Table 1 summarises the optimisations in the catalogue indicating their categories. Each category may have one or more sub-categories, and for each one, there may be several variants implemented as transformation rules. Each sub-category is associated with one or more of the effects mentioned above. The rest of the section presents these categories. In the explanation, we assume that it is possible to access the following information:

- Type of an expression. Given an OCL expression, there is an operation $typeOf(expr)$ which returns its type.
- Superclasses of a type. Given a meta-class, this operation returns its superclasses.
- Nullness of an expression. Given an OCL expression, there is an operation $isNonUndefined(expr)$ which returns $true$ when the expression never yields an OclUndefined value. This operation needs to be conservative in the sense that if must always return $false$ is there is the possibility that $expr$ is OclUndefined.
- Property access type. Given a property access, it returns the type of the accessed property.
- Comparison of two expressions. This is a $compare(expr1, expr2)$ operation which returns $true$ if both expressions are exactly the same.

In Sect. 4, we explain how this operations are made available to the actual transformations which implement the optimisations.

3.2 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}.

Another kind of simplification is to apply rules for the identity element of each arithmetic or logical operation. For instance, tokens + 0 would be simplified to tokens.

It is worth noting that this kind of expressions will be rarely written by a developer, but are likely to appear in synthesised OCL code.

3.3 Operators

This set of optimisations intends to exploit the semantics of operators to remove unnecessary operands or to make the expression more idiomatic and readable.

Table 1 Summary of the different categories of optimisations

	Optimisation	Effect	Action
Literals			
1	Operations with literals	S	Constant-folding
2	Identity element	S	Apply rules of identity elements
Operators			
3	Same operand	S	Remove unneeded operand
4	Sequence of equalities	SIP	Generate Set key1, ..., keyN→includes(value)
Built-in operations			
5	Built-in	SP	Evaluate operations over literals
Iterators			
6	Literal body	SP	Remove the iterator
7	Unused iterator variable	SP	Replace iterator by its body
Let expressions			
8	Noisy let expression	IP_n	Inline let expressions
Type comparisons			
9	oclIsKindOf	S	Replace oclIsKindOf by true
10	Extract common supertype	SI	Replace sequence of oclIsKindOf by a common supertype
Conditionals			
11	Unshort-circuiting	SP_n	Replace oclIsKindOf by true
12	Dead if/else branch	S	Remove if expressions
13	Replace by condition	S	Replace a conditional by its condition
14	Replace by if/then	S	Replace a conditional by then or else branch
15	Equal cond. and then exp.	SP	Replace a conditional by its condition
16	If fusion	S	Merge the *then* (or *else*) branches of two conditionals
17	Complementary conditions	S	Identifies and remove complementary conditions in nested conditionals
18	Intro. operation into cond.	I	Concatenate an operation to both branches of a conditional

S Simplification, *I* idiomatic, *P* performance (positive), P_n negative impact in performance

Same operand Given a boolean expression, it uses the compare operation to check if both operands are the same in order to remove one of them. The following listing shows examples for the and and or operators.

Original

```
place.name = 'green' or  place.name = 'green'
place.name = 'green' and place.name = 'green'
```

Simplified

```
place.name = 'green'
place.name = 'green'
```

Sequence of equalities This optimisation deals with the scenario in which a sequence of the same logical operation checks the equality (or inequality) of the same sub-expression multiple times. The following example shows the optimisation for *and* and *or* operators.

Original

```
–– Sequence of or–equalities with similar 'key'
place.name = 'green' or place.name = 'red' or place.
        name = 'yellow'
–– Sequence of or–inequalities with similar 'key'
place.name <> 'green' or place.name <> 'red' or
        place.name <> 'yellow'

–– Sequence of and–equalities with similar 'key'
place.name = 'green' and place.name = 'red' and
        place.name = 'yellow'
–– Sequence of and–inequalities with similar 'key'
place.name <> 'green' and place.name <> 'red' and
        place.name <> 'yellow'
```

Simplified

```
–– More idiomatic and readable version
Set { 'green', 'red', 'yellow' }→includes(place.name)
–– More idiomatic and readable version
not Set { 'green', 'red', 'yellow' }→includes(place.
        name)

–– The operations can only yield false
false
–– More idiomatic and readable version
Set { 'green', 'red', 'yellow' }→excludes(place.name)
```

The generated code is more readable and can even be more efficient if the transformation engine caches the literal collection. Moreover, the literal collection could be factorised into a constant or a helper by another refactoring triggered by the user to give it a human name (e.g. TrafficLights in the case of the example).

3.4 Built-in operations

This set of optimisations are related to the semantics of built-in operations of the OCL library, in particular when used with empty collections and default values. The following listing shows a few examples.

Original

```
places→size() >= 0
Sequence { }→isEmpty() / notEmpty()
Sequence { }→sum()
```

Simplified

```
true
true / false
0
```

In the current implementation, we have considered mainly sequence operations, but the set of rules is easily extensible with new ones which cover additional operations. For instance, it would be relatively easy to implement a simplification rule which takes an expression like ''.size() and generates 0.

3.5 Iterators

This set of simplifications deals with iterator expressions which can be removed or whose result can be computed at compile time.

Literal body The following listing shows the three optimisations 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
```

Unused iterator variable An iterator expression applies its body to every element of a collection. Thus, it is expected that the iterator variable is used within its body. In the case of boolean iterator expressions like forAll and exists, the body can replace the whole iteration expression because the result does not depend on the collection's values (except the case in which the collection is empty which has to be treated independently). In the case of non-boolean iterator expressions like select, a conditional can be introduced. The following listing shows examples of the optimisations.

Original

```
NetContent.allInstances()→forAll(c |
  Place.allInstances()→forAll(p |
    not c.name.oclIsUndefined());

NetContent.allInstances()→forAll(c |
  Place.allInstances()→exists(p |
    not c.name.oclIsUndefined());

Arc.allInstances()→collect(a |
  Place.allInstances()
    →select(p |
       not c.name.oclIsUndefined())
    →select(p| a.source = p );
```

Simplified

```
NetContent.allInstances()→forAll(c |
   Place.allInstances()→isEmpty() or
   not c.name.oclIsUndefined());

NetContent.allInstances()→forAll(c |
   Place.allInstances()→notEmpty() and
   not c.name.oclIsUndefined());

Arc.allInstances()→collect(a |
  if not c.name.oclIsUndefined() then
     Place.allInstances()→select(p | a.source = p )
  else
     Set { }
  endif);
```

This optimisation may improve execution performance since the collection whose iterator variable is not used does not need to be traversed anymore. It may also improve readability and comprehension because the developer does not need to understand that the iterator is actually useless.

3.6 Noisy let expressions

Let expressions are useful when a large expression is used many times; otherwise, it tends to introduce unnecessary noise. This optimisation 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 optimisation is that it may break well-crafted code, when the developer intended to organise a set of logical steps into let variables. Thus, the optimisation should only be applied for synthesised code which is known to generate repetitive let expressions, and not to optimise handwritten code.

3.7 Type comparison optimisations

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

Remove oclIsKindOf expression An example of this simplification is shown in Fig. 2 and Listing 3, and it is also repeated below with a simpler expression.

Original

```
Arc.allInstances()→forAll(a |
  a.oclIsKindOf(Arc))
```

Simplified

```
Arc.allInstances()→forAll(a | true)
```

The application condition of this simplification is as follows: if an expression *expr* is a single type comparison in the form *expr.oclIsKindOf(T)*, we check whether *typeOf(expr)* = *T* or whether *T* <: *typeOf(expr)* (i.e. *T* is a supertype of *typeOf(expr)*), in which case we can replace the whole expression with $true$. There is, however, an additional check to be done to safely apply the optimisation. We use the isNonUndefined operation to ensure that *expr* cannot be OclUndefined, because OclUndefined.oclKindOf(T) = false. This issue was detected using the verification procedure explained in Sect. 5.

This optimisation is not applicable to oclIsTypeOf because it is only at runtime that we can be sure that an element has exactly the given type. For instance, Arc does not have any subclass in the meta-model, but it might be possible to evaluate the OCL expression using a model which conforms to an extension of the PNML meta-model for which we have defined a subclass of Arc.

Replace subclass checking with superclass This simplification takes a chain of *or* expressions in which each sub-expression checks the type over the same sub-expression 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 recognises 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 sub-expression 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 $types$. This constraint is expressed with the following recursive operation, which checks if a meta-class (self) is included in $types$ (the set of types used in the oclIsKindOf operations) or all its subclasses must appear in such a set (the recursive call).

```
context Class
def : isCoveredBy(types : Set(Class)) =
  if types→includes(self) then
    true
  else
    self.subclasses→notEmpty() and
    self.subclasses→forAll(c | c.isCoveredBy(types))
  endif;
```

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 parameterise the optimisation with a threshold indicating the minimum number of "oclIsKindOf checks" that needs to exists in the original code to trigger it.

3.8 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'.

Replace conditional by its condition The conditional can be replaced by its condition when the *then* and *else* branches are *true* and *false*, respectively.

Original

```
if elem.oclIsKindOf(PN!Place) then
  true
else
  false
endif

if not elem.oclIsKindOf(PN!Place) then
  false
else
  true
endif
```

Simplified

```
elem.oclIsKindOf(PN!Place)
elem.oclIsKindOf(PN!Place)
```

Replace conditional by then/else branches The conditional can be replaced by its *then* or *else* branches when both are the same, and thus, the condition is irrelevant.

Original

```
if place.name = 'red' then
  place.tokens→size()
else
  place.tokens→size()
endif
```

Simplified

```
place.tokens→size()
```

Equal condition and then expression. A simple but useful simplification is recognising 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 the whole expression can be
−− replaced by true
```

If fusion This optimisation applies when there is 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 expression 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 optimised version is more concise, and at the same time enables more simplification opportunities (e.g. false and true can now be simplified).

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 conditionals, one for each boolean sub-expression, which typically leads to unreadable code. In the case of synthesised code, the situation is exacerbated since it is likely that the synthesiser implementation always generates nested conditionals to stay on the safe side.

In its simplest version, the optimisation checks whether the condition is independent of the *then* or *else* branches. This means that we need to ensure that both expressions can be evaluated at the same time. This is exemplified in the following listing, in which the *condition* and the *then* branch are independent since they access different properties of the arc element.

Original

```
if arc.source.oclIsKindOf(Place) then
  arc.target.oclIsKindOf(Transition)
else
  false −− could also be true
endif);
```

Simplified

```
−− When the else branch is false
arc.source.oclIsKindOf(Place) and
arc.target.oclIsKindOf(Transition)

−− When the else branch is true
arc.source.oclIsKindOf(Place) implies
arc.target.oclIsKindOf(Transition)
```

A more complex scenario is exemplified in the following listing (left), which 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, and also recognises that there are nested conditions which can be merged into one. 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
```

This simplification makes use of the expression comparison operation (i.e. the compare operation described at the beginning of the section) to be able reason more accurately about what can be simplified, and also uses type-related operations like typeOf and featureAccess in order to reason about property accesses. Another variant of this optimisation can also be implemented, for instance for checking OclUndefined conditions using the *isNonUndefined* operation.

A caveat is that the optimisation may modify code that has been legitimately crafted to use a conditional for establishing an evaluation order in which simpler conditions are checked first, in order to improve performance. To address this situation, our engine can be configured with a set of elements which can (or cannot) be simplified. In this way, an smart synthesiser can protect the original code against unwanted changes. For instance, in ANATLYZER a quick fix modifies the ATL transformation in-place but we are interested in applying the optimiser only to the elements modified by the quick fix. Hence, the BeautyOCL API allows a "match scope" to be defined. In the case of a quick fix, the scope is represented by the root element that is generated or modified in-place, so that only this element and its children are matched and optimised.

Complementary conditions In a pair of nested conditionals, the inner conditional may not be necessary when its condition is just the complement of the outer conditional. In the following example, it can be seen that the inner conditional is always taken because its condition is always $true$ (e.g. > 10 and $<= 10$ covers the whole spectrum of integer values). Moreover, this would enable the application of the optimisation which will replace the inner conditional by 1.

Original

```
if place.tokens > 10 then
 10
else
 if place.tokens <= 10 then
  1
 else
  10
 endif
endif
```

Simplified

```
if place.tokens > 10 then
 10
else
 if true then
  1
 else
  10
 endif
endif
```

Introduce operation into conditional Given an binary operator or an operation call with one argument, in which the source or the argument is a conditional, it is typically more idiomatic to apply the operation to both branches of the conditional. For instance, in the following example the optimisation introduces $>$ 10 into the conditional, which is more readable (notably when there are more conditionals involved) and also enables more optimisations.

Original

```
if place.tokens = 0 then
 1
else
 place.tokens
endif > 10
```

Simplified

```
if place.tokens = 0 then
 1 > 10
else
 place.tokens > 10
endif
```

This optimisation can be parameterised with the maximum complexity allowed on the embedded operation. In this example, it is clear that the literal 0 is small enough. However, if the expression is too large, applying this rule will lead to less unreadable code.

4 Framework

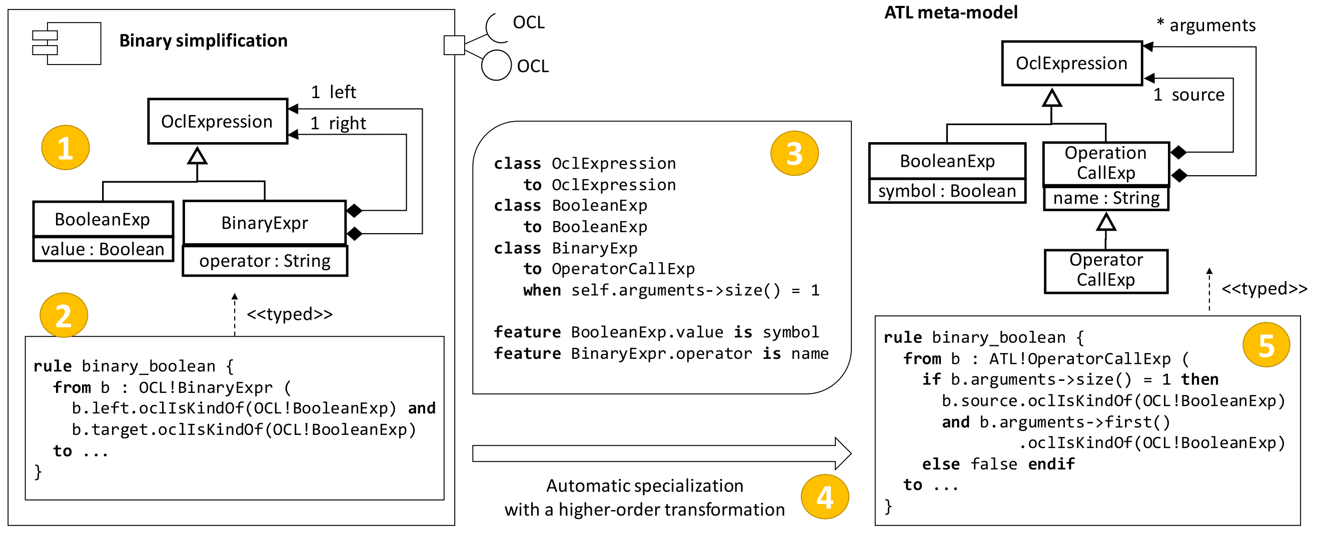
This section describes the design of the reusable component to optimise OCL expressions. We have designed the catalogue of optimisations as a set of reusable transformations using the notion of concept-based transformation components [19]. Our aim is to deal with the fact that there are several implementations of OCL which could be benefited from automatic optimisations. In the EMF ecosystem, we can find the standard OCL distribution (EMF/OCL) [21], SimpleOCL [49], OCL embedded in the ATL language [28], the OCL variant of Epsilon [22], 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. However, all of them have the same semantic underpinnings, and we aim at bridging the structural differences among them in order to amortise the effort of developing the catalogue of OCL optimisations.

4.1 Overview

Our framework is based on the notion of *concept* and generic transformation component [19]. It has been technically implemented using the facilities provided by bentō to develop reusable transformation components [15].

A transformation component encapsulates one or more reusable transformations, which need to be specialised for their use with a concrete meta-model. Figure 3 shows a simple component intended to simplify OCL expressions, for example `true and true` → `true`. The interface of a component is its concept (label 1). 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). In the example, the concept only contains the classes to represent two kinds of OclExpression, boolean literals and binary expressions, since this is the only information that the generic transformation requires to perform its work. The reusable transformation is expressed as a transformation template (label 2), written in ATL in the case of bentō. A template is a regular transformation developed against a concept. To instantiate the component for a specific meta-model (label 3), 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. In this simple example, the component is being adapted to work simplifying one kind of ATL expression (logical expressions). Hence, the result of the specialisation procedure is a new ATL transformation whose transformation rules match and transform ATL elements (label 5). For instance, the adaptation procedure has changed BinaryExpr to OperatorCallExpr modifying the rule filter to make it work properly, according to the binding specification.

The catalogue that we have developed consists of several transformations which target the refactoring of OCL expressions. Figure 4 shows the architecture of the solution. In this design, the optimisation component has a set of small transformations, each one targeting only one kind of optimisation, which have been described in the previous section. In comparison with previous approaches which assumed one concept per transformation [19,43], in this work all transformations share a common OCL-based concept plus two additional concepts to enable parameterised access to type information and expression comparison facilities (see Section 4.3). The output is a set of rewriting commands, which will be interpreted by a custom in-place engine (described below). Given a specific OCL implementation for which we want to reuse the optimisation component, we must implement a binding between the concrete OCL meta-model



```
rule binary_boolean {
  from b : OCL!BinaryExpr (
    b.left.oclIsKindOf(OCL!BooleanExp) and
    b.target.oclIsKindOf(OCL!BooleanExp)
  to ...
}
```

```
class OclExpression
   to OclExpression
class BooleanExp
   to BooleanExp
class BinaryExp
   to OperatorCallExp
   when self.arguments->size() = 1

feature BooleanExp.value is symbol
feature BinaryExpr.operator is name
```

```
rule binary_boolean {
  from b : ATL!OperatorCallExp (
    if b.arguments->size() = 1 then
      b.source.oclIsKindOf(OCL!BooleanExp)
      and b.arguments->first()
              .oclIsKindOf(OCL!BooleanExp)
    else false endif
  to ...
}
```

Fig. 3 A simple component

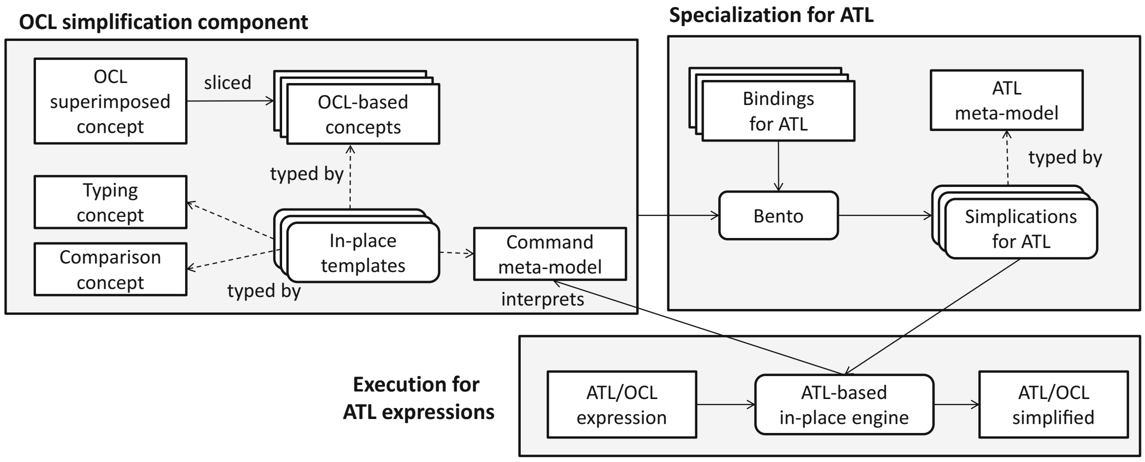
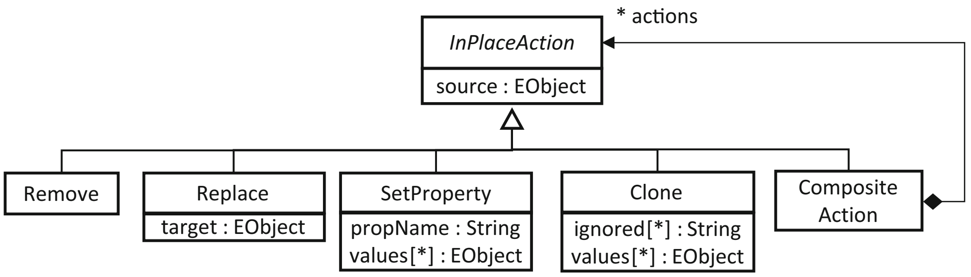


Fig. 4 Architecture of the generic component and its application to optimise ATL/OCL expressions

Fig. 5 Actions meta-model



(ATL/OCL in the figure) and the OCL concept meta-model. Given the binding and the component the bentō tool derives a new optimisation component specialised for ATL. This is fed into the in-place engine to apply the optimisations to concrete ATL expressions.

4.2 Transformation templates

To develop the different transformation templates of the catalogue, we use ATL as our implementation language since this is the language supported by bentō. However, the in-place mode of ATL is quite limited, and it is not adequate to perform the rewritings required to implement the catalogue. Hence, 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 the transformation templates. Secondly, given the motivation of integrating the optimisations within ANATLYZER, it seems logical to use ATL to avoid extra dependencies. Finally, Henshin [3] 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. This made the Henshin specifications too verbose. Other languages like Viatra [48] or EOL [22] have action languages which are imperative. This complicates its integration with bentō because it is difficult to write a higher-order transformations (HOT) for non-declarative languages, which is the main mechanism used by bentō to adapt transformations.

Listing 4 shows a simplification rule written in ATL and our command language. A conditional 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).

```
helper context OCL!OclExpression def: isTrue() : Boolean = false;
helper context OCL!BooleanExp   def: isTrue() : Boolean = self.
        booleanSymbol;

rule removeIf {
  from o : OCL!IfExp ( o.condition.isTrue() )
  to a : ACT!Replace
  do {
    a.source ← o;
    a.target ← o.thenExpression;
  }
}
```

Listing 4 Simplification rule

Actions language. Figure 5 shows the meta-model representing the actions supported by our in-place engine. The are five kind of actions:

- *Remove*. Removes an element (pointed by the *source* property). The element will also be removed from all references pointing to it.
- *Replace*. Replaces an element (pointed by the *source* property) with the element pointed by the *target* property. The target element may be an existing element or a newly created element.
- *SetProperty*. Establishes the value of a given property of the *source* object.
- *Clone*. It clones a *source* object, but leaving unassigned the features listed in the *ignored* property, or setting such features to the objects given in *values* (if *values* is not empty).
- *Composite action*. Represents a sequence of actions which must be executed together in a certain order.

These five kinds of actions are enough for our implementation needs. Simple transformation rules typically consist of a Replace action plus the creation of an OCL element which is used to replace the matched element. Complex rules consist of a CompositeAction which normally contains cloning of some objects, replacements and property set actions.

In-place engine. The execution of an in-place transformation specified using the action language outputs a target model which contains both action elements and new OCL elements. Our in-place transformation engine interprets and applies replacement actions over the source model. Algorithm 1 shows the transformation algorithm. It works by executing transformations and evaluating commands using an iterative, as-long-as-possible algorithm (line 2). Each transformation of the catalogue is tried in a pre-defined order (line 5–6). If one or more matches are found for a transformation, we try to execute all compatible matches. There are two compatibility checks. Firstly, the matched element must be in scope (line 9). This is so because we want to avoid the situation in which we transform elements which should not be optimised, like manually written code. Secondly, we check that a given match is not incompatible with a previous rewriting (line 11), which will happen when a matched element is an ancestor of an already transformed element. If these conditions are met, the rewriting actions are applied (line 13). The transformed element is then added to the transformed list for the next compatibility check. Finally, the tryTransform flag must be set in order to make sure that we check all transformations again the next round.

In this way, for a given optimisation all possible matches are computed, and we apply as many optimisations as possible in one transformation round, before trying other optimisation. In addition, all the transformations are executed in a pre-defined order, and termination has to be guaranteed by ensuring that the generated commands only reduce the given expression, and for those that do not reduce an expression (like "If fusion") we need to make sure that there are no other transformations which undo their work. In this sense, it is possible to extend anATLyzer to enforce this property statically to some extent, which is part of our future work.

```
Input: An expression to be optimised, referred to as ($exp$)
Input: A set of transformations, referred to as ($catalogue$)
Input: An AST element which acts as scope for the transformation ($scope$)
Output: The rewritten input expression
1  def execute (exp, catalogue, scope):
2      tryTransform = true
3      while tryTransform do
4          tryTransform = false
5          foreach trafo in catalogue do
6              matches = execute(trafo, exp)
7              transformed = empty list
8              foreach match in matches do
9                  if not match is within scope then
10                     continue
11                 if transformed has an ancestor of match.source then
12                     continue
13                 apply actions of match // this actually rewrites the expression
14                 add match.source to transformed
15                 tryTransform = true
16             end
17         end
18     end
19 end
```

Algorithm 1: Transformation algorithm used by the in-place engine.

4.3 Concept design

A transformation template is a regular transformation, developed in some transformation language, which is typed against a *concept*. A concept is akin to a meta-model, but it contains the minimum number of elements for the transformation to work. To reuse the template with a concrete meta-model, one needs to provide a binding between the concept elements and the concrete meta-model elements. In our implementation, transformation templates are developed in ATL, 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 6. The *OCL concept* represents the elements of the OCL language which will be subject to optimisations. 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 [20], since they provide hook methods which will be implemented by each specialisation. Essentially, these hybrid concepts allow us to implement the operations described at the beginning of Sect. 3 for each concrete OCL technology.

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 unnecessary complexity from the transformation

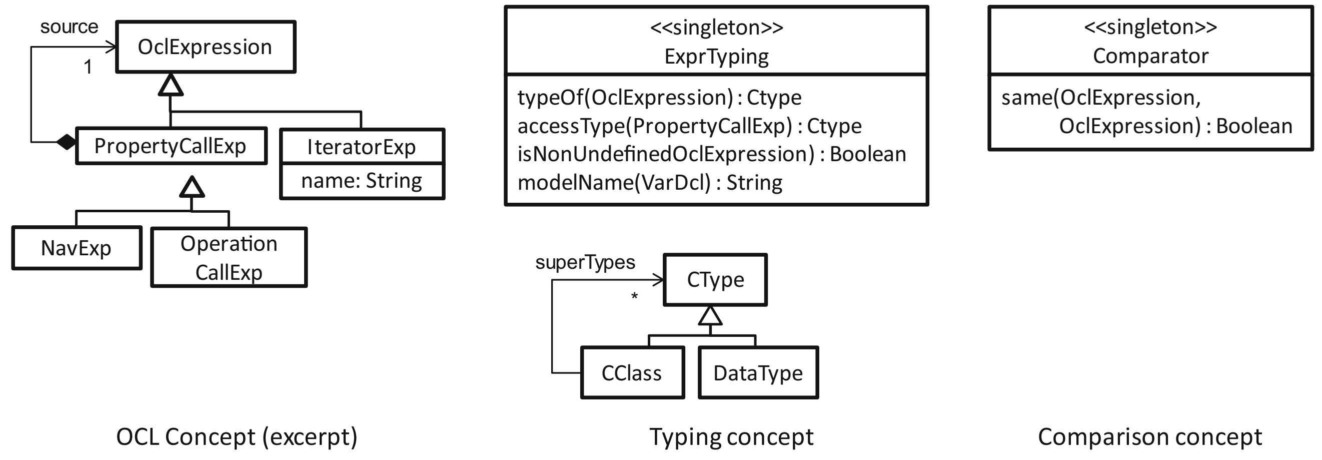


Fig. 6 Concepts used in the simplification component

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 us to have as many bindings as reused transformations. Therefore, we have designed a superimposed concept which contains all the elements required by the transformations of the catalogue. From this concept, we can extract automatically the minimal concept of each transformation using the approach described in [16], so that each individual rewriting could be used in isolation if needed. The superimposed OCL concept (i.e. it merges all the concepts used by the individual rewritings) has the property that it does 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 currently implemented contains only 27 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. Nevertheless, the more optimisations are implemented, the larger the OCL concept will become. For instance, for the extended version of the catalogue we had to include seven more classes.

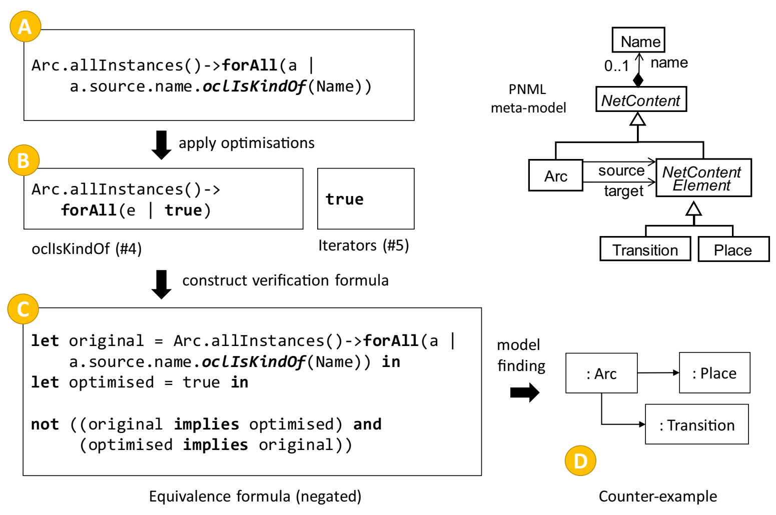
Typing concept There are a number of transformations in the catalogue which require access to the types of the abstract syntax of the OCL expression. One alternative would be to extend the OCL concept with elements to represent typing information. However, this approach is not flexible enough since it 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.

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 serialisations) 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 Verifying the catalogue correctness

The catalogue of transformations must be correct for external tools to integrate it in a confident and reliable manner. In this context, correctness means to satisfy the semantics preservation property, since an optimisation should not change the functional behaviour of the original OCL expression. For example, in anATLyzer our aim is to use the catalogue to generate simplified code for quick fixes. Should one of the transformations in the catalogue be incorrect, an anATLyzer user could introduce a bug in her code inadvertently. Hence, a practical procedure to verify the correctness of the catalogue is needed. One approach would be to implement a verified transformation (akin to a verified compiler [31,32]), but it would require an operational semantics of OCL, specified in a formal language like Coq, as well as developing the transformations in the same formal language. This is not currently a scalable approach since its implementation cost is very high, although recent developments like CoqTL [46] may make it possible. The alternative used in this work has been to resort to a form of translation validation [38,41], which is explained in the following.

Fig. 7 Translation validation example. In this case it uncovers an error in the first optimisation



```
Arc.allInstances()->forAll(a |
    a.source.name.oclIsKindOf(Name))
```

```
Arc.allInstances()->
   forAll(e | true)
```

```
true
```

```
let original = Arc.allInstances()->forAll(a |
    a.source.name.oclIsKindOf(Name)) in
let optimised = true in

not ((original implies optimised) and
     (optimised implies original))
```

Translation validation is an approach to verify compilers in which *rather than proving in advance that the compiler always produces a target code which correctly implements the source code (compiler verification), each individual translation (i.e. a run of the compiler) is followed by a validation phase which verifies that the target code produced on this run correctly implements the submitted source program* [41]. A translation validation approach requires an equivalence criterion (or "correct implementation" relation) between the original program and the compiled program. There are several ways to specify such criterion, which are discussed in more detail in Sect. 7. In our case, we use an off-the-shelf OCL-based model finder, USE Validator [30], to ask for a counter-example of the equivalence between the original and the optimised OCL expressions. Hence, USE Validator is our "semantic oracle" to verify each rewriting of an OCL expression.

Our approach is based on the following steps, which are illustrated by means of an example in Fig. 7.

1. Given a source OCL expression, written in some specific technology, apply the catalogue of optimisations specialised for such technology. Figure 7 (label A) is a piece of OCL expression in ATL which is going to be subject to optimisations. In this case, two optimisations are applied (label B), one that replaces the usage of oclIsKindOf by true, and a subsequent optimisation which removes the iterator expression.
2. Extract a source expression (*original*) and a rewritten expression (*optimised*). This step is straightforward when, for all variable usages in the original expression, the corresponding variable declarations appear in the expression to be rewritten. In the example, the only variable usage, a, is also declared in the expression, Arc.allInstances()⟶forAll(a | a.source.name.oclIsKindOf(Name)). However, if the expression would have been a.source.name.oclIsKindOf(Name), the declaration of a would not appear in the expression. Since we need to feed the model finder with a complete, well-formed OCL expression, we would add a top level expression like Arc.allInstances()⟶forAll(a | <original-expr>). The general algorithm is based on identifying all unbound variable references, and construct a nested sequence of "T.allInstances⟶forAll" expressions according to the type T of each variable usage.
3. To prove that both expressions are equivalent, construct the following equivalence formula (label C), as suggested in [26]:

   $$\text{not}\ \ ((\text{source} \implies \text{target})\ \text{and}\ (\text{target} \implies \text{source}))$$

   We resort to USE Validator to verify the satisfiability of the formula[4].
4. If the model finder outputs a counter-example, this means that both expressions are not equivalent and there is a bug in one of the applied optimisations (label D). If no counter-example is found, then both expressions are equivalent (within the bounds given to the model finder), and it proves that the applied optimisations preserve the semantics of the original expression.

This method provides several advantages. First of all, it enables the practical and pragmatic verification of the catalogue of optimisations, which can be implemented with a modest amount of resources, compared to other verification methods. Another advantage, in this case compared to testing, is that OCL is decidable within a given bounds (i.e. a search scope consisting of the minimum and maximum number of allowed objects of each type and similar bounds for primitive values). This means that the model finder will always

[4] ANATLYZER provides a convenient access to USE Validator which enables seamless analysis of ATL/OCL and EMF/OCL expressions.

find a counter-example within such bounds if it exists, which would signal the existence of a bug in one of the optimisation transformations, and we can use the counter-example as input model for debugging.

Section 6.2 evaluates the application of this verification method to the catalogue. As will be shown, it is generally possible to verify an optimisation in a few milliseconds. It is thus possible to integrate the verification method at runtime in any tool in which the catalogue is integrated. In particular, this can be used as an option in anATLyzer, so that optimisations are verified at runtime and a rollback is applied whenever the verification step fails.

The example of Fig. 7 shows an actual error found in the catalogue using this procedure. At first glance, the optimisation looks correct since the original expression a.source.name.oclIsKindOf(Name) will always be *true* because the type of name is Name and therefore, the oclIsKindOf call will always return *true*. However, the witness model obtained with the verification procedure indicates that the optimisation is invalid when we have a model which includes a Place without a name. This is because OclUndefined.oclIsKindOf(Name) $\Longrightarrow$ false, and the original implementation neglected this case. To fix this, we introduced the isNonUndefined operation in the typing concept, in order to restrict the cases in which the implementation is applicable.

Finally, this approach also opens an interesting possibility, which is to have “optimistic” optimisations, in the sense of dropping some of the application conditions, letting the decision about whether this was a good decision or not to the verification procedure. For instance, in a expression like the following:

```
Arc.allInstances()→select(a | a.source.name <> OclUndefined)
            →select(a | a.source.name.oclIsKindOf(Name))
```

It can be difficult to determine statically that a.source.name will never be undefined, and thus, our optimisation is applicable[5]. If we aim at maximising the amount of applicable optimisations, we can drop the isNonUndefined check in the optimisation implementation and rely on the verification procedure to determine if the optimisation was properly applied. We aim at studying the possibilities of this approach in future work.

6 Evaluation

This section reports the evaluation of our reusable catalogue of optimisations and the associated verification method. The evaluation aims to answer three research questions:

- RQ1: Are the optimisations proposed in the catalogue useful to deal with automatically generated OCL code?
- RQ2: Are the implemented optimisations in the catalogue correct?
- RQ3: To what extent is the catalogue reusable?

To answer RQ1 and RQ2, we have used ATL/OCL expressions automatically generated by anATLyzer as a testbed to apply the catalogue and verify its correctness. For RQ3 we have specialised the catalogue for three OCL implementations.

6.1 RQ1: Usefulness

In this experiment, we evaluate the usefulness of our catalogue of optimisations to deal with realistic, automatically generated OCL expressions by analysing whether the optimisations are able to reduce its complexity.

We have applied the optimisations 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 pre-conditions generated from target invariants of model transformations as described in [12]. We optimised 24 constraints coming from invariants defined in three transformations used by existing literature HSM2FSM, ER2REL and Factories2PetriNets. The second experiment applied the optimisations 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 2 summarises 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-sosym19.

For the pre-conditions, a total of 1415 optimisations were applied to 24 expressions. In average, 58.96 simplifications were applied for each expression; however, the median was 9 simplifications. This is because some expressions were particularly large and involved more simplifications. For instance, two of the expressions had more than 3500 abstract syntax elements, which enabled the application of more than 300 simplifications for each one. In the quick fixes experiment, a total of 9510 simplifications were applied to 1731 expressions. We express the simplification power of the catalogue (shown in the “% nodes removed by simplifications” row) by counting the number of AST nodes before and after the simplifications. The reduction is 18% for the pre-conditions dataset and 36% for the quick fixes dataset (Table 3).

Regarding which optimisation categories are more useful, the results are disparate. Some optimisations occur much more often in one experiment than in the other. For instance,

[5] anATLyzer does provide this analysis, but most OCL tools do not. For these cases, an optimistic approach would enable more aggressive optimisations.

Table 2 Summary of the results for the experiment with the *pre-conditions* dataset

	Pre-conditions			
	#Simp.	%	Avg.	Median
Literals	8	0.57	–	–
Identity element	8	0.57	–	–
Same operand	0	0.00	–	–
Sequence of equalities	0	0.00	–	–
Built-in operations	26	1.84	–	–
Iterators	8	0.57	–	–
Noisy let	0	0.0	–	–
Remove oclIsKindOf expression	22	1.55	–	–
Replace subclass checking with superclass	0	0.00	–	-
Remove dead if/else branch	22	1.55	–	–
Replace conditional by its condition	0	0.00	–	–
Replace conditional by then/else branches	0	0.00	–	-
Equal condition and then expression	0	0.00	–	–
If fusion	267	18.87	–	–
Unshort-circuiting	168	11.87	–	–
Complementary conditions	4	0.28	–	–
Introduce operation into conditional	882	62.33	–	-
Total simplifications	1415	100	58.96	9
% of nodes removed by simplifications			18.02%	2.54%

Table 3 Summary of the results for the experiment with the *quick fixes* dataset

	Quixk fixes			
	#Simp.	%	Avg.	Median
Literals	862	9.06	–	–
Identity element	74	0.78	–	–
Same operand	0	0.00	–	–
Sequence of equalities	152	1.60	–	–
Built-in operations	0	0.00	–	–
Iterators	706	7.42	–	–
Noisy let	153	1.61	–	–
Remove oclIsKindOf expression	1795	18.87	–	-
Replace subclass checking with superclass	0	0.00	–	-
Remove dead if/else branch	1795	18.87	–	–
Replace conditional by its condition	11	0.12	–	–
Replace conditional by then/else branches	0	0.00	–	-
Equal condition and then expression	706	7.42	–	-
If fusion	863	9.07	–	–
Unshort-circuiting	2376	24.98	–	–
Complementary conditions	0	0.00	–	–
Introduce operation into conditional	17	0.18	–	–
Total simplifications	9510	100	5.49	2
of nodes removed by simplifications			35.56%	20.75%

simplifications for literals and conditionals are very useful for quick fixes, but less useful for pre-conditions. This suggests that optimisations 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 more 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 for a real quick fix.

Original

```
RDM!Antecedent.allInstances()→select(i | i.oclIsTypeOf(RDM!
        Antecedent))→
 forAll(i | i.containsAtom→forAll(v |
  if v.oclIsKindOf(RDM!Atom) then
    v.name = 'IndividualPropertyAtom'
  else
    false
  endif or if v.oclIsKindOf(RDM!Atom) then
    v.name = 'ClassAtom'
  else
    false
  endif or if v.oclIsKindOf(RDM!Atom) then
    v.name = 'DataRangeAtom'
  else
    false
  endif or if v.oclIsKindOf(RDM!Atom) then
    v.name = 'DataValuedPropertyAtom'
  else
    false
  endif or if v.oclIsKindOf(RDM!Atom) then
    v.name = 'SameIndividualAtom'
  else
    false
  endif or if v.oclIsKindOf(RDM!Atom) then
    v.name = 'DifferentIndividualAtom'
  else
    false
  endif or if v.oclIsKindOf(RDM!Atom) then
    v.name = 'BulitinAtom'
  else
    false
  endif));
```

Simplified

```
RDM!Antecedent.allInstances()→select(i |i.oclIsTypeOf(RDM!
        Antecedent))→
 forAll(i | i.containsAtom→forAll(v |
   Set { 'BulitinAtom', 'DifferentIndividualAtom', 'SameIndividualAtom',
        'DataValuedPropertyAtom', 'DataRangeAtom', 'ClassAtom',
        'IndividualPropertyAtom' }→includes(v.name)));
```

The optimisations applied have been the following: (1) to replace the oclIsKindOf operation by true (7 times), then (2) to replace the if expression by its then expression (7 times) and finally (3) to simplify the sequence of *or* expression checking equalities by a `Set{...}→contains(v.name)` operation. As can be observed, the result is much more readable. In other evaluated expressions, the results are not so "beautiful", but nevertheless the expression is much more reduced which facilitates its manual modification. It is expected that the catalogue grows as new needs for optimisations are found, which would improve the overall results even more.

The catalogue instantiated for ATL has been integrated into ANATLYZER through a dedicated extension point, so that the generated quick fixes are automatically optimised. 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 synthesised 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. Another threat is that some optimisations are never applied, and we cannot determine whether this is because they are not really applicable or because of a bug in the application condition. To minimise this threat, we have manually searched the original expressions to find places in which an optimisation is not applied because of a bug in the application condition, and we have also constructed a manual test suite. 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. Moreover, many of them are complex transformations (e.g. use type information or compare sub-expressions) which could not be addressed with simple text-based transformations.

6.2 RQ2: Correctness

To validate the correctness of the catalogue, we have used the translation validation approach explained in Sect. 5. This section reports the results of applying the verification method to the optimisations applied for RQ1.

For each OCL expression generated in the previous experiment, we filter out those expressions for which no optimisation has been applied. Then, we apply the method explained in Sect. 5 to verify the correctness of the optimised expression with respect to the original expression. To increment the amount of expressions that can be verified successfully, we also compute the set of ATL helpers that are directly or indirectly used by the expression and add them to the formula as part of operations in the meta-model. The results are summarised in Table 4. To run the verification procedure, the bounds of the model finder were set to a minimum of 0 objects and a maximum of 5 objects per meta-class. The verification procedure was run several times during the development of this work, and it allowed us to discover and fix several errors, which will be briefly discussed later. In the following, we present the results of the last run.

For the *pre-conditions* data set, there were only 20 optimised expressions (out of 24). No time out was set. We could only verify eight expressions. The other ones (reported as *Failure*) could not be evaluated because there is no support in

Table 4 Results of the correctness evaluation, using 5 as object bounds for the model finder

Correctness	Pre-conditions	textbfQuick fixes
#Optimisations	20	1411
#Correct	8	908
#Invalid	0	0
#Timeout	0	12
#Failure	12	491
Solving times (s)		
Average	4.272	0.532
Mean	11.953	0.014
Maximum	21.641	14.55

USE Validator for tuple types. We also measured the solving times of the expressions which were successfully evaluated by USE Validator. However, the obtained times are very disparate, since some of the expressions are much larger than others. Unfortunately, this dataset is too small to reach any conclusions.

For the *quick fix* data set, there were a total of 1411 optimised expressions. In this case we set up a time out of 15 seconds. We could verify 908 optimisations (64%), all of which were deemed correct by the decision procedure. There were only 12 timeouts. We inspected manually the expressions which were not evaluable, and they fall in one of three categories: (a) expressions which contain features not supported by USE Validator (e.g. maps, sequence operations, etc.), (b) expressions which include target elements of a model-to-model transformation and cannot be processed correctly by our translation to USE Validator and (c) expressions which contain errors because the original transformation also contained errors. Thus, improvements in the internal translation from ATL to USE Validator may increase the number of verifiable expressions to a certain extent, but it is unlikely that all of the expressions can be verified in practice.

With respect to the performance of the verification, we have also measured the solving times of the expressions which were successfully evaluated by USE Validator, taking into account only those which did not timeout. The average is 0.532 seconds and the maximum is 14.55 seconds. However, the mean is very low 0.14, signifying that the verification of, at least, half of the applied optimisations is very fast. These results are further explored next.

6.2.1 Performance of the verification procedure

Our verification method is a form of runtime verification since we need to apply it for each run of the catalogue on a given expression. In some scenarios, we may need certain guarantees that the verification will be fast enough to avoid disturbing the user. One such scenario is the application of a quick fix, since the user expects a rapid response from the IDE. On the other hand, a model finder is normally highly dependent on the given object bounds: if the bounds are small, the response is likely to be fast, but it may be unreliable. However, as the bounds are increased, the response time might grow in an exponential manner.

In order to understand what could be a good configuration for the anATLyzer quick fix facility, we have evaluated how an increase of the bounds affects the correctness results and the corresponding solving times. We wanted to know if an increase in the object bounds will result in the detection of an invalid optimisation which may not be catched with lower bounds. To this end, we have run the same experiment explained above but increasing the object bounds from 2 to 8. Table 5 shows how many optimisations have been deemed correct, incorrect or have time out, as the object bounds are increased. As can be observed, no invalid optimisations are found with greater bounds. However, with smaller bounds there are less timeouts. This suggests that for this dataset, small bounds are sufficient and do not compromise reliability.

Figure 8 shows the evolution of the solving time as the object bounds are increased. In this case, we set the timeout to 5 seconds. As can be observed, the average solving time quickly grows when the object bounds are greater than 5. Interestingly, the mean remains very low in all cases, signifying that this growth in the average is probably due to a few expressions which are too complex to evaluate (i.e. those which time out).

In general, the model finder exhibits good performance for this scenario. This is probably due to two reasons. First of all, because the expressions generated by the quick fixes are generally not very complex. Secondly, because our mapping to USE Validator automatically performs meta-model slicing to reduce the search space (i.e. remove unneeded meta-classes and features), which is an important factor to reduce the solving time.

Altogether, this experiment shows that it is possible to integrate our verification method in an IDE like anATLyzer using a reasonable maximum object bound of three or four objects per meta-class, and a timeout of about two seconds.

6.2.2 Discovering and fixing bugs

With the help of the correctness approach, we found several bugs in the implementation. These bugs were located not only in the implementation catalogue but also in other involved technologies, notably anATLyzer and ATL. In the process of tracing back the root case of a bug, the fact that the verification process outputs an actionable witness model (i.e. a regular EMF model) was very valuable. In the following we describe some of the discovered bugs.

Table 5 Correctness results as object bounds are increased. Timeout is set to 5 seconds

Bounds	#Optimisations	#Correct	#Invalid	#Timeout	#Failure
2	1411	916	0	4	491
3	1411	908	0	12	491
4	1411	908	0	12	491
5	1411	869	0	51	491
6	1411	837	0	83	491
7	1411	836	0	84	491
8	1411	835	0	85	491

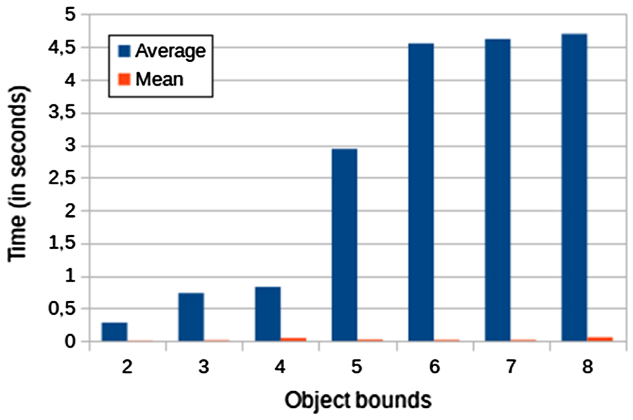


Fig. 8 Evolution of solving time w.r.t. object bounds

– We found a bug in the *Remove oclIsKindOf expression* optimisation which was described before. We missed the proper treatment of OclUndefined values.
– A bug in the *Replace subclass checking with superclass* optimisation was uncovered by transformation which used a meta-model with multiple inheritance (XHTML2XML transformation). Our implementation did not properly find the common supertype. Fixing this bug makes the optimisation not to be applicable anymore to any of the expressions generated in the evaluation.
– In the implementation of *Unused iterator variable*, there was another semantic bug because the additional expression to check whether the collection is empty or not was missing.
– We found implementation bugs in *Sequence of equalities* and *Unshort-circuiting*, which were solved as well.
– We detected two errors in anATLyzer. First, some of the quick fixes do not generate faithful type information along with the generated OCL code, which made the optimisation rules take incorrect decisions. Second, there was a bug in an internal routine which was in charge of copying pieces of abstract syntax.
– We also found an issue with ATL. The ATL parser treats the and/or operators with the same precedence, which is different from the normal behaviour in which *and* has higher precedence. We needed to update our infrastructure to deal with this behaviour.

Altogether, the fact that we found bugs in the catalogue provides working evidence of the usefulness of the verification method, and the need of similar mechanisms to increase the reliability of published model transformations.

6.3 RQ3: Reusability

The catalogue of optimisations has been designed with reusability in mind in order to allow its specialisation 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. In this section we report to what extent it is possible to reuse the catalogue for different OCL dialects, reflecting on the advantages and limitations of the approach.

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 consist of 1593 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.

This experiment shows that it is possible to automatically derive specialisations of the catalogue for several OCL variants. However, we have only shown the correctness of the ATL specialisation, and only simple, manual test cases have been carried out to show that the OCL/EMF and SimpleOCL specialisations also work. Therefore, without further experiments we cannot fully claim that the catalogue is reusable.

6.4 Assessment

The evaluation shows that our catalogue is useful to reduce the size of the expressions generated by two different methods implemented in anATLyzer. However, the best results are obtained when the catalogue provides optimisations targeted to the specific patterns of code generated by a particular synthesiser. This suggests that catalogues of optimisations created for other languages need to be easily extensible with new transformations.

We have also shown that it is possible to prove the correctness of the catalogue, to a certain extent. This is a step forward in the model transformation technology since model transformations are typically not verified. One of the reasons is the difficulty to apply compiler verification techniques to model transformations. The scope of a compiler is larger than a model transformation, so it does not pay off to apply techniques which are costly and heavyweight. Moreover, any technique which aims at checking semantic preservation requires the specification of the semantics of the involved languages, which is also costly to construct. The alternative put forward in this paper is to use translation validation taking advantage of an OCL-based finder as its semantic anchor. The main limitation is that this correctness procedure guarantees that a given execution of a transformation is correct, within a bound, but cannot discover bugs in transformations which have not been executed yet. That is, it is a form of runtime verification. On the other hand, the main difficulty to generalise this verification method to other transformations is the need of having some semantic oracle to check for equivalence, as we do with USE Validator.

Regarding the reusability of the catalogue, our approach is based on rewriting the original transformation to generate a new transformation which conforms to a different meta-model. This tights our implementation to bentō, which provides this capability. An alternative which does not require the bentō machinery would be to make our concept behave as a pivot meta-model and write transformations (instead of binding specifications) from each OCL variant to the pivot meta-model. The catalogue would then be applied over the pivot meta-model. However, the optimisations need to modify the expressions in-place, and this is problematic for the pivot meta-model approach since we would need to implement a synchronising transformation to translate the changes in the pivot model back to the original model. Instead, the rewriting process performed by bentō addresses this issue seamlessly since there is no intermediate transformation process, but a new complete catalogue is derived for the selected OCL variant.

At the implementation level, we have used ATL to perform rewritings. The experience with this project shows that for simple rewritings it can be very effective and we have been able to profit from the existing infrastructure. However, for complex rewritings the main bottleneck is the difficulty to navigate the AST and to match complex patterns using OCL.

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 amortised 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 Related work

The closest work to ours was proposed by Giese and Larsson [25]. 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 means of examples. More complex cases including conditionals, let expressions and the treatment 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 [9], finding that refactored expressions are more understandable. The experiments were carried out on handwritten expressions; thus, it is likely that refactorings for expressions generated automatically have an even bigger impact on understandability. In [51], 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 [10]. Some of them are of interest for our case and we also implement variants of them, particularly refactorings for verbose expressions, while others are particularly useful for handwritten OCL expressions and we do not implement them. A complementary work with additional refactorings is presented in [42]. Cabot and Teniente [7] 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 optimisations patterns to improve the performance of OCL expressions in ATL programs is presented in [17]. Altogether, this work complements existing works with new optimisations targeted specially to automatically generated code, and contributes a working and extensible implementation.

The notion of optimisation has been used in many other areas, including compilers (in which they are used pervasively), program transformation, program repair and mutation testing. In each case the criteria to evaluate the usefulness of the optimisation vary. For instance, one source of related works is expression simplification rules developed with program transformation systems. In [33] a catalogue of optimisations is presented for imperative languages. In the program repair field there is a similar to need to obtain human readable patches. In [24] the maintainability of automatically generated patches is studied, considering features like number of conditionals, descriptive variable names, etc. However, the work was not conclusive about which elements affect maintainability more. DirectFix is program repair system which seeks to generate patches which are syntactically minimal [34]. To evaluate it the authors compare against the human-written patch and also compare AST nodes. In [4] the metrics used to measure repair maintainability are the size of the repair in terms of changed lines of code and the distribution of the modifications. For mutation testing, optimisations produced by a compiler have been used to detect equivalent mutants [29] in C and Java programs. Hence, our catalogue can also be useful to detect equivalent mutants in OCL-based programs.

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 [2,25,45]. Other works modify OCL expressions as a response to meta-model evolution [27]. 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 such approaches tend to generate "alien code" [35] 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 [14] and the generation of pre-conditions [11,12,36]. Hence, we believe that this work will be also valuable to complement OCL synthesis tools likely to appear in the future.

The design of the typing and comparison concepts is, to some extent, similar to the idea of Mirrors [5] which also aims to separate reflective operations from its particular implementation. In our case, we do not want to tight the component to the implementation based on providing type information as annotations over the abstract syntax tree. The typing concept introduces this intermediate layer for flexibility.

Another set of related works concerns transformation verification. The problem of proving that a given transformation preserves the semantics of the source language is akin to the problem of verifying a compiler [32]. The most common approach is based on writing the compiler using some theorem prover like Coq [32] or Isabelle/HOL [31]. In the MDE setting, CoqTL [46] could be used to develop transformations amenable to compiler verification techniques. Other approaches verify compilers and transformations indirectly using a proof-carrying code approach (translation validation), that is, to verify each transformation run independently [37]. In [1] code generators are extended to generate semantic annotations and assertions to guide a model checker that checks that the generated code meets the UML semantics. In [8] translation validation is used to validate the SimpleGT compiler (a variant of ATL for in-place transformations). Equivalence is proved by a translation of each piece of generated EMFTVM bytecode (the output of SimpleGT) to the Boogie verification system, in which a formalisation of the EMFTVM semantics has been implemented. In [40] a translator from an asynchronous language, SIGNAL, is compiled to C. The equivalence criterion is based on checking whether transition system specified by the original program is equivalent to the transition system represented by the generated code. In [47] a transformation from StateCharts to PetriNets is verified by a pair of transformations to a common transition system. In general, translation validation requires a different equivalence criterion depending on the nature of source and target language. Notably, our translation validation method is based on using model finding with USE Validator as a way to implement the equivalence criterion.

8 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 35%. More importantly, we have implemented a practical verification procedure based on translation validation which has allowed us to prove correct about 64% of the applied optimisations. We have also shown that the performance of the verification method is generally good enough to use it as part of editing environments. As future work, we plan to investigate the possibility of applying "optimistic optimisations" in order to widen the application conditions of the current optimisations. Also, we would like to extend bentō to allow using rewriting languages like Stratego [6] 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 is funded by project RECOM (Spanish MINECO, TIN2015-73968-JIN, AEI/FEDER/UE) and a Ramón y Cajal 2017 grant (RYC-2017-237) funded by MINECO (Spain) and co-funded by the European Social Fund. I would like to thank the reviewers for their accurate and useful comments, and Javier Bermúdez Ruiz for proofreading the first version of the manuscript.

References

1. Ab Rahim, L., Whittle, J.: Verifying semantic conformance of state machine-to-Java code generators. In: International Conference on Model Driven Engineering Languages and Systems, pp. 166–180. Springer (2010)
2. Ackermann, J., Turowski, K.: A library of OCL specification patterns for behavioral specification of software components. In: International Conference on Advanced Information Systems Engineering, pp. 255–269. Springer (2006)
3. Arendt, T., Biermann, E., Jurack, S., Krause, C., Taentzer, G.: Henshin: advanced concepts and tools for in-place EMF model transformations. In: International Conference on Model Driven Engineering Languages and Systems, pp. 121–135. Springer (2010)
4. Assiri, F.Y., Bieman, J.M.: An assessment of the quality of automated program operator repair. In: 2014 IEEE Seventh International Conference on Software Testing, Verification and Validation, pp. 273–282. IEEE (2014)
5. Bracha, G., Ungar, D.: Mirrors: Design principles for meta-level facilities of object-oriented programming languages. OOPSLA'04, ACM SIGPLAN Notices **50**(8), 35–48 (2015)
6. Bravenboer, M., Kalleberg, K.T., Vermaas, R., Visser, E.: Stratego/xt 0.17. a language and toolset for program transformation. Sci. Comput. Program. **72**(1–2), 52–70 (2008)
7. Cabot, J., Teniente, E.: Transformation techniques for ocl constraints. Sci. Comput. Program. **68**(3), 179–195 (2007)
8. Cheng, Z., Monahan, R., Power, J.F.: Formalised emftvm bytecode language for sound verification of model transformations. Softw. Syst. Model. **17**(4), 1197–1225 (2018)
9. Correa, A., Werner, C., Barros, M.: 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, pp. 76–90. Springer (2007)
10. Correa, A., Werner, C.: Refactoring object constraint language specifications. Softw. Syst. Model. **6**(2), 113–138 (2007)
11. Cuadrado, J.S., Guerra, E., de Lara, J.: Static analysis of model transformations. IEEE Trans. Softw. Eng. **17**(3), 779–813
12. Cuadrado, J.S., Guerra, E., de Lara, J., Clarisó, R., Cabot, J.: 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), pp. 12–22. IEEE (2017)
13. Cuadrado, J.S., Guerra, E., de Lara, J.: Anatlyzer: An advanced IDE for ATL model transformations. In: 40th International Conference on Software Engineering (ICSE). ACM/IEEE (2018)
14. Cuadrado, J.S., Guerra, E., de Lara, J.: Quick fixing ATL transformations with speculative analysis. Softw. Syst. Model. pp. 1–35 (2016)
15. Cuadrado, J.S., Guerra, E., de Lara, J.: Reusable model transformation components with bentō. In: International Conference on Theory and Practice of Model Transformations, pp. 59–65. Springer (2015)
16. Cuadrado, J.S., Guerra, E., de Lara, J.: Reverse engineering of model transformations for reusability. In: International Conference on Theory and Practice of Model Transformations, pp. 186–201. Springer (2014)
17. Cuadrado, J.S., Jouault, F., Molina, J.G., Bézivin, J.: Optimization patterns for OCL-based model transformations. In: International Conference on Model Driven Engineering Languages and Systems, pp. 273–284. Springer (2008)
18. Cuadrado, J.S.: Optimising OCL synthesized code. In: European Conference on Modelling Foundations and Applications, pp. 28–45. Springer (2018)
19. Cuadrado, J.S., Guerra, E., de Lara, J.: A component model for model transformations. IEEE Trans. Softw. Eng. **40**(11), 1042–1060 (2014)
20. de Lara, J., Guerra, E.: From types to type requirements: genericity for model-driven engineering. Softw. Syst. Model. **12**(3), 453–474 (2013)
21. Eclipse Modelling Framework. https://www.eclipse.org/modeling/emf/
22. Epsilon. http://www.eclipse.org/gmt/epsilon
23. Fleck, M., Troya, J., Kessentini, M., Wimmer, M., Alkhazi, B.: Model transformation modularization as a many-objective optimization problem. IEEE Trans. Softw. Eng **43**(11), 1009–1032 (2017)
24. Fry, Z.P., Landau, B., Weimer, W.: A human study of patch maintainability. In: Proceedings of the 2012 International Symposium on Software Testing and Analysis, pp. 177–187. ACM (2012)
25. Giese, M., Larsson, D.: Simplifying transformations of OCL constraints. In: International Conference on Model Driven Engineering Languages and Systems, pp. 309–323. Springer (2005)
26. Gogolla, M., Hilken, F., Doan, K.H.: Achieving model quality through model validation, verification and exploration. Comput. Lang. Syst. Struct. **54**, 474–511 (2018)
27. Hassam, K., Sadou, S., Le Gloahec, V., Fleurquin, R.: Assistance system for OCL constraints adaptation during metamodel evolution. In: Software Maintenance and Reengineering (CSMR), 2011 15th European Conference on, pp. 151–160. IEEE (2011)
28. Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I.: 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
29. Kintis, M., Papadakis, M., Jia, Y., Malevris, N., Le Traon, Y., Harman, M.: Detecting trivial mutant equivalences via compiler optimisations. IEEE Trans. Softw. Eng. **44**(4), 308–333 (2017)
30. Kuhlmann, M., Hamann, L., Gogolla, M.: Extensive validation of OCL models by integrating sat solving into use. In: International Conference on Modelling Techniques and Tools for Computer Performance Evaluation, pp. 290–306. Springer (2011)

31. Kumar, R., Myreen, M.O., Norrish, M., Owens, S.: Cakeml: a verified implementation of ml. In: ACM SIGPLAN Notices, vol. 49, pp. 179–191. ACM (2014)
32. Leroy, X.: Formal verification of a realistic compiler. Commun. ACM **52**(7), 107–115 (2009)
33. Loveman, D.B.: Program improvement by source-to-source transformation. J. ACM **24**(1), 121–145 (1977)
34. Mechtaev, S., Yi, J., Roychoudhury, A.: Directfix: Looking for simple program repairs. In: Proceedings of the 37th International Conference on Software Engineering-Volume 1, pp. 448–458. IEEE Press (2015)
35. Monperrus, M.: 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, pp. 234–242. ACM (2014)
36. Mottu, J.M., Simula, S.S., Cadavid, J., Baudry, B.: Discovering model transformation pre-conditions using automatically generated test models. In: 2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE), pp. 88–99. IEEE (2015)
37. Necula, G.C.: Proof-carrying code. design and implementation. In: Proof and system-reliability, pp. 261–288. Springer (2002)
38. Necula, G.C.: Translation validation for an optimizing compiler. In: ACM sigplan notices, vol. 35, pp. 83–94. ACM (2000)
39. OMG: Object Constraint Language (OCL) (2014). http://www.omg.org/spec/OCL/2.4/PDF
40. Pnueli, A., Shtrichman, O., Siegel, M.: Translation validation: from signal to C. In: Olderog, E.R., Steffen, B. (eds.) Correct System Design. Lecture Notes in Computer Science, vol. 1710, pp. 231–255. Springer, Berlin, Heidelberg (1999)
41. Pnueli, A., Siegel, M., Singerman, E.: Translation validation. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems, pp. 151–166. Springer (1998)
42. Reimann, J., Wilke, C., Demuth, B., Muck, M., Aßmann, U.: Tool supported OCL refactoring catalogue. In: Proceedings of the 12th Workshop on OCL and Textual Modelling, Innsbruck, Austria, September 30, 2012, pp. 7–12 (2012). https://doi.org/10.1145/2428516.2428518
43. Rose, L., Guerra, E., De Lara, J., Etien, A., Kolovos, D., Paige, R.: Genericity for model management operations. Softw. Syst. Model. **12**(1), 201–219 (2013)
44. Solar-Lezama, A., Tancau, L., Bodik, R., Seshia, S., Saraswat, V.: Combinatorial sketching for finite programs. ACM Sigplan Notices **41**(11), 404–415 (2006)
45. Tibermacine, C., Sadou, S., Dony, C., Fabresse, L.: Component-based specification of software architecture constraints. In: Proceedings of the 14th International ACM Sigsoft Symposium on Component Based Software Engineering, pp. 31–40. ACM (2011)
46. Tisi, M., Cheng, Z.: Coqtl: An internal DSL for model transformation in COQ. In: International Conference on Theory and Practice of Model Transformations, pp. 142–156. Springer (2018)
47. Varró, D., Pataricza, A.: Automated formal verification of model transformations. CSDUML pp. 63–78 (2003)
48. Varró, D., Bergmann, G., Hegedüs, Á., Horváth, Á., Ráth, I., Ujhelyi, Z.: Road to a reactive and incremental model transformation platform: three generations of the viatra framework. Softw. Syst. Model. **15**(3), 609–629 (2016)
49. Wagelaar, D.: Simpleocl. https://github.com/dwagelaar/simpleocl
50. Wimmer, M., Strommer, M., Kargl, H., Kramler, G.: Towards model transformation generation by-example. In: System Sciences, 2007. HICSS 2007. 40th Annual Hawaii International Conference on, pp. 285b–285b. IEEE (2007)
51. Wimmer, M., Perez, S.M., Jouault, F., Cabot, J.: A catalogue of refactorings for model-to-model transformations. J. Object Technol. **11**(2), 2–1 (2012)

Jesús Sánchez Cuadrado is a researcher at the Languages and Systems Department of the University of Murcia. His research is focused on model driven engineering (MDE) topics, notably model transformation languages, meta-modelling and domain specific languages. On these topics, he has published several articles in journals and peer-reviewed conferences, and developed several open source tools. His web-page is http://sanchezcuadrado.es

.