Tool Supported OCL Refactoring Catalogue

Jan Reimann, Claas Wilke, Birgit Demuth, Michael Muck, Uwe Aßmann
Technische Universität Dresden
Institut für Software- und Multimediatechnik
D-01062, Dresden, Germany
firstname.lastname@tu-dresden.de | michaelmuck@gmx.de

ABSTRACT

The Object Constraint Language (OCL) as the primary constraint language in model-driven software development is heavily used to specify static semantics of arbitrary languages and models. Models and constraints are therefore interconnected and depend on each other. On the one hand, daily work with models enjoys a good tool support, whereas, on the other hand, mature OCL tools are not widely spread but a niche. Unfortunately, during their life-time, the complexity of models rises and so do their OCL constraints. Thus, the gap between conventional modelling and OCL tools becomes obvious. This fact demands for OCL tool support to cope with the complexity. To bridge this gap, *refactoring* is well-suited and mighty. In this paper we discuss existing work, present a revised catalogue of OCL-exclusive refactorings and provide an implementation. We do not consider co-refactorings of OCL constraints and their constrained models.

Categories and Subject Descriptors

D.2.3 [**Coding Tools and Techniques**]: Program editors; D.2.7 [**Distribution, Maintenance, and Enhancement**]: Restructuring, reverse engineering, and reengineering

Keywords

OCL, model refactoring, OCL-exclusive refactoring, refactoring catalogue

1. INTRODUCTION

In recent years, OCL has become a popular constraint language used to specify constraints on top of several Meta Object Facility (MOF)-based metamodels, as well as Unified Modeling Language (UML) models, and models from other domains such as XML. However—as in conventional programming languages—when specifications become complex, qualities such as reusability, readability, and understandability must be maintained to improve comprehensibility for modellers working with these constraints. Listing 1 shows an example for a constraint from the Nomos XML validation service demo. As can be seen, the constraint uses the association call `dbtr.pstlAdr` as well as the idiom `sourceExp->size()=0` several times. Thus, *refactorings* [5] can be used to extract the association call `dbtr.pstlAdr` into a variable and the idiom into a helper method, or replacing it with a more appropriate expression, such as `sourceExp=null` (cf. Listing 2). Altogether, the XML validation demo consists of 79 constraints having similar structure and requiring similar refactorings to improve their readability and maintainability.[1]

```
1 context DirectDebitTransactionInformation1
2 inv EPC_AOS_DrctDbtTxInfDbtr:
3    dbtr.pstlAdr.adrTp->size() = 0 and
4    dbtr.pstlAdr.strtNm->size() = 0 and
5    dbtr.pstlAdr.bldgNb->size() = 0 and
6    dbtr.pstlAdr.pstCd->size() = 0 and
7    dbtr.pstlAdr.twnNm->size() = 0 and
8    dbtr.pstlAdr.ctrySubDvsn->size() = 0 and
9    dbtr.ctryOfRes->size() = 0
```

Listing 1: OCL example from Nomos Software's XML validation demo.

```
1 context DirectDebitTransactionInformation1
2 inv EPC_AOS_DrctDbtTxInfDbtr:
3    let adr = dbtr.pstlAdr
4    in
5       adr.adrTp = null and
6       adr.strtNm = null and
7       adr.bldgNb = null and
8       adr.pstCd = null and
9       adr.twnNm = null and
10      adr.ctrySubDvsn = null and
11      dbtr.ctryOfRes = null
```

Listing 2: OCL example with applied refactorings.

Today, OCL refactorings are usually done manually. This process of restructuring constraints is very error-prone since their modification must not change the meaning of the constraints. Thus, refactoring is one of the most wanted features for OCL tooling, which was recently underpinned by a survey conducted among the OCL community [2]. To the best

[1]All constraints of the example as well as their model (an XML schema file) are online available as an example shipped together with Dresden OCL.

of the authors' knowledge, currently only catalogues of OCL refactorings exist, but no publicly available tool supports OCL refactoring yet. Furthermore, existing catalogues lack details (e.g., incomplete pre- or postconditions). For this reason, in this paper we contribute (1) an analysis of existing approaches of OCL refactorings, (2) an extended catalogue, and (3) a refactoring tool for Dresden OCL[2] implemented with the generic model refactoring framework Refactory [9].[3]

The remainder of this paper is structured as follows: In Sect. 2 we present related work. Afterwards, in Sect. 3 we present and discuss our extended catalogue. Sect. 4 shortly discusses our refactoring implementation for Dresden OCL. Finally, Sect. 5 concludes our work.

2. REFACTORING FOR OCL REVISITED

Several authors have already focused on OCL refactorings. Their work can be divided into two categories: (1) OCL refactorings as co-refactorings and (2) OCL-exclusive refactorings. *Exclusive* refactorings are directly intended to modify OCL expressions, whereas *co-refactorings* modify OCL expressions and their constrained model in parallel (e.g., modifications in a UML class model where classes referenced in OCL constraints are renamed).

In the area of UML/OCL co-refactoring important research was done by Marković and Baar [8] or by Hassam et al. [7]. These authors developed approaches for synchronising OCL constraints specified for evolving UML models. Since we focus on OCL-exclusive refactorings, we do not discuss these papers further. In the following, related work for OCL-exclusive refactorings is elaborated.

In [6] Giese and Larsson investigate possibilities for the automatic simplification of OCL constraints. Based on the idea of design patterns, not only implementations for UML model elements may be generated, but also the corresponding OCL constraints. In such a scenario automatically or semi-automatically generated OCL constraints often contain redundancies. To simplify constraints suffering from this problem, they propose the repeated application of small, simple rules, as e.g. simplifying `(false and e)` to `(false)`.

The most important work on OCL-exclusive refactorings was published by Correa and Werner [3, 4]. They presented a unique collection of OCL smells which are used as a basis for a catalogue of refactorings that can be used to enhance understandability of OCL constraints. Table 1 shows their OCL smells and the according refactorings. Critical refactorings from their findings are discussed in the following. Refactoring (2) is used to split a large boolean expression into separate expressions being connected by **and** operators. Due to the greater operator binding strength of the **and** operator in comparison to that of the **implies** operator, it is necessary for this refactoring to put the intermediary results of the type `a implies b` into a set of brackets before linking them by **and**. Otherwise, the refactoring would change the semantics of the expression, a fact that has not been mentioned by Correa and Werner. Refactoring (3) splits constraints consisting of boolean expressions connected by **and**

[2] `http://www.dresden-ocl.org/`
[3] `http://www.modelrefactoring.org/`

OCL refactoring	Associated smell
(1) Replace implies chain by a single implication	Implies chain
(2) Split conditional rules	Non-atomic rule
(3) Split AND chain	AND chain
(4) Replace forAll chain by navigations	forAll chain
(5) Remove redundant expression	Redundancy
(6) Simplify operation calls	Duplication
(7) Change context	Down-casting
(8) Change initial navigation	Verbose expression
(9) Add variable definition	Magic literal
(10) Replace expression by variable	Magic literal
(11) Add operation definition	Long Journey
(12) Replace expression by operation call	Long Journey
(13) Add property definition/replace expression by property call	Magic literal
(14) Introduce polymorphism	Type-related conditionals
(15) Inline attribute/inline operation definition	
(16) Rename attribute/rename operation	

Table 1: Refactorings and related OCL code smells proposed in [3].

operators into several constraints for the same context. The authors did not discuss that this refactoring can be applied on invariants, pre- and postconditions only, as for definitions and body constraints the semantics can not be split into several expressions. Refactorings (5) and (6) are formulated very vaguely (e.g., the authors propose to assure that the target expressions must not change the semantics, a statement that holds for each refactoring). In addition, refactoring (7) also is very imprecise because it is only said that a context change should be performed in case of long unreadable association paths. Fortunately, this problem was discussed by Cabot and Teniente [1]. They formalised their approach as a path problem over a graph which enables them to determine all alternatives. Refactoring (8) could be solved similar to (7) but it cannot be claimed that it is always semantically correct to replace an association path with a shorter one. It may occur that one path has another intent than another one and does not have the same meaning. Refactorings both (9–10) and (11–12) in combination are used to foster the reuse of expressions. But it remains unclear why Correa and Werner separated atomic extract refactorings into the extraction and the replacement of the occurrences. Opposed to that refactoring (13) is examined in combination. The refactorings combined under (15) are the inverses of (12–13) but Correa and Werner do not go far enough. One can say that every extraction refactoring can be inverted by its inlining. The according smell might be *single use* which occurs in case when the extract is only referred to once.

3. OCL REFACTORING CATALOGUE

In this section we categorise the refactorings identified in related work as well as further refactorings identified by our group. We do not assume our catalogue as being complete. However, we argue that our catalogue contains several of the most required refactorings for OCL constraint specification and maintenance. We grouped the 28 identified refactorings into four categories, namely, (1) *renamings*, (2) *removals* and their inverse materialisation, (3) *extractions* and their inverse inlinings, and (4) *separations* and their inverse

RENAMING
Preconditions: The target name η of the element ε to be renamed and any other element in its scope must be disjunct.
Steps: (1) Locate the declaration of the element ε to be renamed. (2) Compute the scope σ of ε. (3) Verify that σ does not include any element of the same kind having name η. (4) If no match was found, rename ε and all its references. Otherwise cancel the refactoring.

Table 2: Renamings

merges. Below, the individual refactorings are presented and discussed based on the following pattern: first, the motivation or necessity for each refactoring is discussed. Where necessary, examples illustrate the further meaning of individual refactorings. Besides, the preconditions as well as the necessary steps to perform a refactoring are summarised as enumerations in a table. We denote the selected element(s) which is/are intended to be refactored with α.

3.1 Renamings

In general, renaming elements can be considered as the most common refactoring (Table 2). In the context of OCL, everything that has a name can be renamed which includes constraints, variables (including parameter names in method declarations), as well as operations and properties, if they are defined by an OCL definition (operations and properties from the constrained model cannot be renamed without affecting a co-refactoring of the constrained model). An important prerequisite for this refactoring is that a respective element of the new name must not already exist within the scope of the renamed element. The scope of a constraint includes all other constraints defined on the same model. The scope of a variable is the area in which it is visible in. For methods and attributes the scope includes all other methods or attributes defined on the same class and its superclasses.

3.2 Removals

Removal refactorings (Table 3) remove unnecessary, redundant elements from OCL expressions. In some cases, their opposite materialisations are useful refactorings as well.

Remove Unused Elements. During the evolution of an OCL expression sometimes variables are no longer required as all their uses have been removed from the constraint. Removing these unused declarations improves readability as well as the execution performance of OCL expressions, as unnecessary variables consume memory for their allocation as well as computation time for their initialisation. The same holds for unused properties or operations defined as helper elements using OCL **def** declarations.

Remove/Add Redundant Brackets. While reading OCL expressions, an often found smell is redundant brackets. Considering an expression `(a+b)+c`, the brackets surrounding `a+b` can be removed without affecting the readability of the affected expression. However, in other cases, such as `a+b*c` explicit brackets improve readability as natural reading order does not represent the arithmetic precedence of the expression (which is `a+(b*c)`). Thus, adding redundant brackets is considered as a useful refactoring as well.

REMOVE UNUSED ELEMENTS
Preconditions: α should contain at least one **let** expression, property or operation definition, respectively.
Steps: (1) Find all variable/property/operation declarations Δ within α. (2) $\forall\delta\epsilon\Delta$: (2-1) Search for any references to δ. (2-2) If no reference is found, remove δ. (2-3) If for a removed variable declaration no variable remains in the same **let** expression λ, remove λ.
REMOVE/ADD REDUNDANT BRACKETS
Steps: (1) If brackets shall be removed search for brackets within α which do not alter the precedence rules of α and remove them. (2) If brackets shall be added, surround α with a pair of brackets.
REMOVE/MATERIALIZE self
Steps: (1) Locate all references to `self`. (2) If explicit references shall be removed, remove all explicit references to `self`. (3) If explicit references shall be materialised, search for all implicit references to `self` and make them explicit.
REMOVE/MATERIALIZE TYPE DECLARATIONS
Steps: (1) Let E denote the set of all expressions within α (also nested expressions) that can have an explicit type declaration. (2) If type declarations shall be removed: $\forall\varepsilon\epsilon E$: if ε's type declaration δ can be inferred, remove δ. (3) If type declarations shall be materialised: $\forall\varepsilon\epsilon E$: if ε misses an explicit type declaration δ, add δ.
REMOVE IMPLICIT asSet
Preconditions: α should contain at least one implicit collection conversion.
Steps: (1) Be C the set of all implicit collection conversions in α. (2) $\forall\varepsilon\epsilon C$: in ε = `exp->op()` replace ε with an explicit collection conversion ε' = `exp.asSet()->op()`.
REMOVE IMPLICIT collect
Preconditions: α should contain at least one implicit `collect()` iterator.
Steps: (1) Be C the set of all implicit `collect()` iterators in α. (2) $\forall\ \varepsilon\epsilon C$: in ε = `sourceExp.targetExp` (where `sourceExp` is an expression resulting in a collection of values) replace ε with ε' = `sourceExp->collect(targetExp)`.
REMOVE DEPRECATED null CHECK
Preconditions: α should contain at least one **null** check η of the pattern `a->size()=0`, `a->size()> 0`, `a->size()= 1`, or `a->size()<> 0` where `a` denotes an expression resulting in a non-collection value.
Steps: (1) Be N the set of all deprecated null checks in α. (2) $\forall\eta\epsilon\alpha$: (2-1) If η = `a->size()=0` replace η with η' = `a=`**null**. (2-2) Otherwise replace η with η' = `a<>`**null**.

Table 3: Removals

Remove/Materialise Self. Since explicit usage of the `self` reference for property and operation calls is optional, some calls show this reference while others do not. This refactoring is intended to unify the selected constraints according to the individual preference of the user.

Remove/Materialise Type Declaration. OCL does not require to explicitly specify the type of expressions, if the type can be inferred from the expression. Sometimes, ex-

plicit type declarations make code more unreadable (e.g., if a collection type is specified although the expression uniquely identifies the result type). However, in other cases, explicit type declarations may be more appropriate.

Remove implicit `asSet`. In OCL operation calls are separated into calls on collection and non-collection operations. The former are denoted using an arrow (->), the latter using a dot (.) notation. When the arrow notation is used on a non-collection expression, its value is implicitly converted into a collection before invoking the operation call (e.g., in Listing 1, all property call results are converted into collections before invoking the `size()` operation). Although this can be usable in some cases, it can lead to unexpected semantics [10]. Thus, we propose a refactoring to convert implicit collection conversions into explicit ones.

Remove implicit collect. A counterpart of implicit collection conversions is the implicit `collect()` iterator that applies an expression onto all elements contained in a collection (e.g., an expression `company.employees.salary*12` results in a collection containing the salaries of all employees of a company multiplied by 12). Although this seems to be quite useful in some cases, the implicit collect iterator may lead to confusing results as one might expect the example expression resulting in a single integer value. Thus, refactoring the expression into the explicit form `company.employees->collect(salary*12)` improves its comprehensibility.

Remove deprecated null check. In former versions, OCL did not allow the **`null`** literal, which caused the unavailability of **`null`** checks. A common idiom to check for **`null`** values was the expression `a->size()=0` which implicitly converts a non-collection value into a collection and checks the collection for its size. Since OCL 2.3, the **`null`** literal can be used for explicit checks on **`null`** values. Thus, we propose the *remove deprecated null check* refactoring to replace the idiom with an appropriate **`null`** check. This refactoring could be applied to improve the readability and comprehensibility of the constraint shown in Listing 1 as shown in Listing 2.

3.3 Extractions

Extraction refactorings (Table 4) extract expressions into variables, properties, or operations to reuse them from several calls within the same or even several OCL constraints. There counterparts are the inverse inlining refactorings.

Extract Variable. The extraction of a variable is related to the two refactorings *add variable definition* and *replace expression by variable* from Correa and Werner [3]. The refactoring can be applied to the example given in Listing 1, where multiple occurrences of the property call `dbtr.pstlAdr` are replaced by a variable `adr` (cf. Listing 2) which improves both readability and evaluation performance. If a user wants to extract a single literal (e.g., an integer or boolean value into a variable), it is impossible to automatically decide if other literals having the same value in the scope of the newly introduced variable shall be replaced with a variable call as well or not. Thus, for literal values, this refactoring only extracts the one selected by the user. Note that not only variables defined in a **`let`** expression have to be checked for name collisions, but also all iterator variables and operation parameters of the constraint's **`context`** declaration as well.

EXTRACT VARIABLE
Preconditions:
If another variable in the scope of the variable to be introduced has the same name, its type and value must match the selected expression α to be extracted.
Steps:
(1) Compute the scope of α. (2) If a variable ν having the same name already exists, continue if its initialisation expression matches α, else abort. (3) If no variable having this name exists: (3-1) Add a declaration of a new variable ν to the beginning of the body expression containing α and initialise it with α. (3-2) If α is a literal replace α with a call to ν. (3-3) If α is no literal replace all occurrences of α in the scope of the extracted variable with a call to ν.

INLINE VARIABLE
Preconditions:
References to the variable ν have to be present within α.
Steps:
(1) Check if α is a variable declaration or an expression containing variable references. (2) If α contains references: (2-1) If the variable ν to inline has a primitive type, encapsulate its initialisation in brackets to ensure correct precedence rules. (2-2) Replace the references with ν's initialisation expression. (3) If α is a variable declaration: (3-1) Find all references to the variable ν. (3-2) If ν has a primitive type, encapsulate its initialisation expression in brackets. (3-3) Replace all found references with the encapsulation. (4) Perform *remove unused variables* to remove unused variables.

EXTRACT PROPERTY/OPERATION
Preconditions:
If the names of any property/operation and the one to be extracted on a class or superclass are not disjunct, its type and expression must match α. If a property shall be extracted, α must not contain any references to externally defined variables or parameters.
Steps:
(1) Check if a property/operation ϕ of the given name is already defined on the same class or a superclass. (2) If this is not the case, add a new property/operation ϕ and initialise it with α. (2-1) If ϕ is an operation, collect the set R of external references to variables or parameters not being part of α. (2-2) $\forall \gamma \epsilon R$: add a parameter ρ to the operation declaration having the same name and type as γ. (3) Locate all occurrences of α and replace them with a call to ϕ. (4) If ϕ is an operation, add the corresponding arguments for each external reference from α to variables or parameters to the operation call in a way they match the signature and semantics of ϕ's dedclaration.

INLINE PROPERTY/OPERATION
Preconditions:
References to the property/operation ϕ have to be present within α. ϕ must not be declared recursively.
Steps:
(1) Check if α is a reference to ϕ or its declaration. (2) If α is a property/operation call: (2-1) If ϕ's expression ε to inline results in a primitive type, $\varepsilon' = (\varepsilon)$ to ensure precedence rules. Else $\varepsilon' = \varepsilon$. (2-2) Replace the selected call with ε'. (3) If α is ϕ's declaration, let R denote all references to ϕ: (3-1) $\forall \gamma \epsilon R$: Replace γ with ε'. (3-2) If γ is an operation call, replace the names of all parameter calls in the inlined ε', with their counterparts from γ. (4) Perform *remove unused properties/operations*.

Table 4: Extractions

```
1 context DirectDebitTransactionInformation1
2 def: adr: PostalAddress1 = dbtr.pstlAdr
3
4 context DirectDebitTransactionInformation1
5 def: getAdrTp(adr PostalAddress1): AdressType2Code =
6       adr.adrTp
```

Listing 3: Extracted properties and operations.

Inline Variable. Inline variable is the inverse refactoring to *extract variable*. There is either the possibility to replace one selected call to the variable, or all occurrences of the variable at once. If the user selected a reference to the variable, only this selected reference will be replaced by the variable initialization. But if the user selection is the variable declaration, all references to the selected variable will be computed and replaced by the variable initialisation.

Extract Property/Operation. Besides the extraction of variables, it is also possible to extract a property or operation. For example, instead of using a **let** expression, the expression `dbtr.pstlAdr` could be extracted from the example constraint in Listing 1, resulting in a newly defined property `adr` (cf. Listing 3, lines 1–2). Now, we can reuse this property definition to ease the example constraint as well as all other constraints defined on the same class. In some cases it is also sensible to extract an operation instead of a property. This is necessary if the expression we want to extract includes references to parameters or variables within the constraint we want to extract the expression from. In this case, we can extract an operation having all this external references as parameters. Listing 3, lines 4–6 shows an example for such an extracted operation from Listing 2. We extracted the expression `adr.adrTp` which is a property call having `adr` as its source expression. As `adr` is a locally defined variable, we have to pass it as a parameter to the operation.

Inline Property/Operation. Inline a property or operation is the inverse operation to the above *extract property/operation* refactoring. There is either the possibility to replace one selected call to the property/operation, or all occurrences of the property/operation at once. If the user selected a reference to the property/operation, only this selected reference will be replaced by the property's/operation's expression. But if on the other hand the user selection is the property/operation definition, all references to the selected property/definition will be computed and replaced by the respective expression. For operation inlining, existing parameter calls of the operation's expression must be replaced with the names of their counterparts from the operation call, as they may have other names in the inlined context.

3.4 Separations

Separation refactorings (Table 5) are refactorings splitting complex expressions into several, typically more readable refactorings. Their counterparts are merge refactorings that, in some cases, can improve readability as well.

Merge Chained Let Expressions. The OCL specification allows for an arbitrary number of **let** expressions to be present in one OCL constraint. Often, they are forming a chain at the beginning of the constraint's body expression which can be combined to one single, often more readable **let** expression.

MERGE CHAINED let EXPRESSIONS
Preconditions:
α should contain chained let expressions.
Steps:
(1) Find the first **let** expression ε whose child λ is a **let** expression.
(2) Move all variable declarations from λ to its parent ε maintaining their order.
(3) In ε replace λ with its child expression.
(4) Repeat the refactoring until no further chained **let** expression remain.
SPLIT BOOLEAN EXPRESSIONS
Preconditions:
α must conform to one of the three expression types (1–3).
Steps:
(1) Verify that the top level expression of α is a boolean-typed expression β of one of the three types.
(1-1) If β is of type (1): $\forall x_i$ *in* β: create an ε_i = (`x`$_i$ **implies** `y`).
(1-2) If β is of type (2): $\forall y_i$ *in* β: create an ε_i = (**x implies** `y`$_i$).
(1-3) If β is of type (3): $\forall b_i$ *in* β: create an ε_i = (**a implies** `b`$_i$), $\forall c_i$ *in* β: create a γ_i = (**(not a) implies** `c`$_i$).
(2) α' = the connection of all ε_i (and all γ_i) with **and**.
(3) Replace α with α'.
SPLIT/MERGE CONSTRAINT
Preconditions:
(a) If a constraint shall be split, α must be an invariant, pre-, or postcondition and its body expression must be a conjunction γ of expressions.
(b) If several constraints shall be merged, α must contain only invariants, only pre-, or only postconditions. Each constraint δ in α must be defined on identical context declarations. If a δ is pre- or postcondition, all variable names of the constraints in α must be disjunct.
Steps:
(1) If a constraint δ shall be split up:
(1-1) Check if there is a conjunction at the top level of δ. If not, end the refactoring.
(1-2) Remove the **and** operator and replace it with its first child expression.
(1-3) Create a new constraint declaration δ' according to δ within the same context and place the second child expression from the conjunction into δ'.
(1-4) Repeat from step (1-1) for both δ and δ'.
(2) If constraints $\kappa \epsilon \alpha$ shall be merged:
(2-1) Let Δ denote the set of all **context** declarations in α.
(2-2) $\forall \delta_i \epsilon \Delta$, i > 1: adapt the names of all parameter's ρ_j in δ_i to the name of the corresponding ρ_j in δ_1.
(2-2) Surround the expressions to be merged with brackets to ensure precedence rules.
(2-3) Merge the expression using conjunctions.
SPLIT/MERGE CONTEXT DECLARATIONS
Preconditions:
For splitting a **context** declaration δ, more than one OCL constraint must belong to δ. For merging **context** declarations (set Δ), the expressions to be merged must have identical contexts.
Steps:
(1) If δ shall be split up, locate each constraint κ declared within δ. $\forall \kappa_i \epsilon \Delta, i > 1$: Create a new **context** declaration δ' with the same **context** class or element and move κ_i to δ'.
(2) If Δ shall be merged, verify that the contexts of each declaration $\gamma_i \epsilon \Delta$ are identical. $\forall \kappa_i \epsilon \bigcup_{j=2..n} \delta_j$: Move constraint κ_i from γ_j to δ_1. Afterwards remove the remaining empty declarations $\delta_i \epsilon \Delta, i > 1$.

Table 5: Separations

Split boolean expressions. This refactoring to split a boolean expression was proposed by Correa and Werner [3]. The motivation for applying this refactoring is to split a large expression into several expressions connected by boolean **and** operators. While this alone may not seem worthwhile, the result allows using the *split constraint* refactoring explained below for further splitting. The refactoring can be applied to modify three types of boolean expressions:

$$\frac{(x_1 \ \ldots \ \textbf{or} \ x_n) \ \textbf{implies} \ y}{(x_1 \ \textbf{implies} \ y) \ \ldots \ \textbf{and} \ (x_n \ \textbf{implies} \ y)} \quad (1)$$

$$\frac{(x \ \textbf{implies} \ (y_1 \ \ldots \ \textbf{and} \ y_n))}{(x \ \textbf{implies} \ y_1) \ \ldots \ \textbf{and} \ (x \ \textbf{implies} \ y_n)} \quad (2)$$

$$\frac{\begin{array}{c}\textbf{if} \ a \ \textbf{then} \ (b_1 \ \ldots \ \textbf{and} \ b_m) \\ \textbf{else} \ (c_1 \ \ldots \ \textbf{and} \ c_n) \ \textbf{endif}\end{array}}{\begin{array}{c}(a \ \textbf{implies} \ b_1) \ \ldots \ \textbf{and} \ (a \ \textbf{implies} \ b_m) \\ \textbf{and} \ ((\textbf{not} \ a) \ \textbf{implies} \ c_1) \ \ldots \ \textbf{and} \ ((\textbf{not} \ a) \ \ \textbf{implies} \ c_n)\end{array}} \quad (3)$$

Split/merge Constraint. Splitting an OCL constraint into multiple constraints is a refactoring related to the *split and chain* refactoring introduced by Correa and Werner [3]. This refactoring is designed for complex OCL constraints that consist of several boolean conditions bound together by logical **and** operators. For example, the invariant shown in Listing 1 could be split into several invariants checking the non-initialisation of several properties individually. As such a restructuring is only applicable for constraints used for consistency checks and not for rules specifying semantics for operations and properties, this refactoring can only be applied on invariants, pre- and postconditions. The opposite refactoring *merge constraint* can be applied to merge several constraints defined on the same context into one constraint.

Split/merge Context Declarations. The refactorings *merge* and *split context declaration* unify multiple constraints under one or split them to multiple **context** declarations (e.g., multiple invariants defined on the same class can share a common **context** declaration or can use individual ones). These refactorings are motivated by the fact that, especially in models with a large number of OCL expressions, placing each individual expression within its own context classifier provides the user with a clearer separation of the individual expressions. However, in some cases using individual **context** declarations may improve readability, especially if the individual constraint's expressions require many lines of codes leading to situations where the **context** declaration of following constraints would not be visible on the screen together with its body expression.

4. REALISATION

For providing an extensible tool where refactorings can be added, a framework is required. Therefore, we implemented the catalogue above with the generic model refactoring framework Refactory [9] for the Eclipse-based tool Dresden OCL.

Dresden OCL provides a collection of tools for parsing, evaluating and editing OCL expressions for languages based on the Eclipse Modeling Framework (EMF), as well as XML and Java. Since the constraint editor from Dresden OCL itself is EMF-based, Refactory could be used easily to provide OCL-refactorings because it also supports EMF languages. Our tool has been tested on several models and metamodels, among them various UML models, the Modelica metamodel and the UML.

Detailed information on our publicly available tool can be found under `http://www.dresden-ocl.org/refactoring`.

5. CONCLUSION

In this paper we analysed the OCL-exclusive refactorings published in [3]. We showed that they have deficiencies and provided a new extended catalogue which corrects existing OCL refactorings and encloses additional ones. Finally, we shortly presented a tool which provides refactoring support for Dresden OCL realised with the tool Refactory. It is the first refactoring tool for OCL being freely available.

In future work we will investigate OCL co-refactoring for maintaining OCL constraints and constrained models in parallel as well as further OCL refactorings not discussed in this catalogue.

Acknowledgements

This research has been co-funded by the European Social Fund and the Federal State of Saxony within the project ZESSY QualiTune #0809518061. We also like to thank Nomos Software for providing their OCL constraints.

6. REFERENCES

[1] J. Cabot and E. Teniente. Transforming OCL constraints: a context change approach. In *ACM symposium on Applied computing*, 2006.

[2] J. Chimiak-Opoka, B. Demuth, A. Awenius, D. Chiorean, S. Gabel, L. Hamann, and E. Willink. OCL Tools Report based on the IDE4OCL Feature Model. In *OCL and Textual Modelling*, 2011.

[3] A. Correa and C. Werner. Refactoring object constraint language specifications. *Software and Systems Modeling*, 2007.

[4] A. Correa, C. Werner, and M. Barros. Refactoring to improve the understandability of specifications written in object constraint language. *Software, IET*, 3(2):69–90, 2009.

[5] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[6] M. Giese and D. Larsson. Simplifying Transformations of OCL Constraints. In *Model Driven Engineering Languages and Systems*. Springer, 2005.

[7] K. Hassam, S. Sadou, V. Gloahec, and R. Fleurquin. Assistance System for OCL Constraints Adaptation during Metamodel Evolution. In *CSMR2011*, 2011.

[8] S. Marković and T. Baar. Refactoring OCL Annotated UML Class Diagrams. *Software and Systems Modeling*, 2008.

[9] J. Reimann, M. Seifert, and U. Aßmann. On the reuse and recommendation of model refactoring specifications. *Software and Systems Modeling*, 2012.

[10] C. Wilke and B. Demuth. UML is still inconsistent! How to improve OCL Constraints in the UML 2.3 Superstructure. *EASST*, 2011.