Specification and design refactorings for sustainable agile model-driven engineering

Kevin Lano[1], Shekoufeh Rahimi[2] and Zishan Rahman[1]

[1]King's College London, London, UK
[2]University of Roehampton, London, UK

Abstract

Refactoring is an essential technique in agile development, aiming to improve the quality of delivered code, and to manage the technical debt (TD) levels of software over system lifetimes. For agile model-driven engineering (MDE), refactorings should be applicable at the software modelling level. In addition, due to concerns about climate change, refactorings should also reduce software energy use.

In this paper we describe refactorings of software specifications and designs expressed in UML/OCL models, and evaluate the effectiveness of these refactorings for reducing software energy-use.

Keywords

Model-driven Engineering, Agile Development, Refactoring, Sustainable Software

1. Introduction

Model-driven Engineering (MDE) advocates the development of software based on the use of software models, such as UML and OCL models. Agile MDE aims to combine MDE techniques and processes with those of agile methods such as Extreme Programming (XP) and Scrum. A key technique of agile methods is the *refactoring* [1] of code to (i) correct quality defects that may have arisen due to the agile emphasis on rapid code production, prior to system delivery; (ii) to remove quality flaws that remain in delivered code – also referred to as the *technical debt* (TD) of the code – in order to reduce its ongoing maintenance costs [2].

Although the focus of refactoring has been on code quality improvement in the context of manual code production, software developed using MDE and automated code generation may also have TD issues [3], and MDE artefacts such as software models and model transformations can also have quality flaws [4], which may lead to flaws in code generated from models. Thus it is important to have available refactorings for software models and model transformations, which can address quality flaws and technical debt as part of an Agile MDE process.

In response to increasing concern about the contribution of computing technologies (ICT) to climate change, the concept of *sustainable software* has been defined [5]. An important aspect of software sustainability is the reduction of energy use (and hence greenhouse gas emissions) due to software operation, and we will investigate to what extent software model refactorings can assist in achieving this goal.

In Section 2 we survey related work, in Sections 3, 4 we describe selected refactorings for energy-use reduction and quality improvement, and in Section 5 carry out evaluation of the effect of some example refactorings.

2. Background

The paradigm of technical debt and the detection of TD through the identification of 'code smells' [2, 6] is now widely-adopted in mainstream software engineering, and implemented via tools such as

Joint Proceedings of the STAF 2025 Workshops: OCL, OOPSLE, LLM4SE, ICMM, AgileMDE, AI4DPS, and TTC. Koblenz, Germany, June 10–13, 2025.

kevin.lano@kcl.ac.uk (K. Lano); shekoufeh.rahimi@roehampton.ac.uk (S. Rahimi); zishan.rahman@kcl.ac.uk (Z. Rahman)
0000-0002-9706-1410 (K. Lano)

published 2025-12-10

SonarQube [7]. Although the use of a rigorous model-driven engineering approach with automated code-generation should reduce the frequency of code flaws, in principle, there remains the problem of flaws in MDE artefacts such as models, metamodels and model transformations.

From the early years of MDE there was awareness of the issue of quality flaws in MDE artefacts and the need for transformations (refactorings) to correct these [8]. Potential metamodel and model flaws include concrete superclasses, excessive depth of inheritance hierarchies, excessive numbers of subclasses, or multiple inheritance [9, 10]. Model transformation flaws include the existence of cloned rule elements, excessively complex transformation rules and excessive dependencies between rules [4, 11]. OCL constraints may be poorly and unclearly expressed, and may result in inefficient execution and high energy use if translated directly into program code. The papers [12, 13] considered refactoring of OCL specifications for quality improvement and increased efficiency, whilst [14] defined a catalogue of 24 refactorings for model transformations. Some model transformation design patterns [15] can also be regarded as defining refactorings, replacing transformation coding without the pattern by an improved version using the pattern.

In [16] we introduced the concept of the *computational cost* of an expression or statement d, defined as $cost(d) = \Sigma_{a \in \mathcal{C}(d)} cost(a)$, where $\mathcal{C}(d)$ is the collection of basic actions needed to compute d. The $cost(d)$ can be used as a model-level estimator to compare the relative energy uses of alternative designs d.

The use of program refactorings to reduce energy use was investigated by [17]. They found that some of the well-known refactorings for quality improvement had an adverse effect on software energy use. In particular, converting a local variable to a field, extracting a method, inlining methods, introducing indirection and introducing a parameter object could all increase energy use when tested across a range of applications. Only extracting a local variable showed a consistent energy-use reduction. In [16] we used computational cost analysis to show that this refactoring should reduce energy use, under certain assumptions. This refactoring could also be used for model transformations, via the use of OCL *let* expressions or mechanisms such as the *using* clause of ATL rules, thus adding a further transformation refactoring to the catalogue of [14].

3. Specification refactorings

The AgileUML toolset for Agile MDE [18] uses the OCL notations of the OMG standard [19], extended by additional data structures and types (functions, maps, sorted sets and sorted maps), and additional operators such as lambda expressions and regular expression operators on strings [20, 21].

While the OCL refactorings defined in [12, 13, 14] still apply to this extended AgileUML OCL, there are also new refactorings for the new operators. Table 1 lists OCL optimisations from [12, 13, 14], and additional refactorings, which should reduce computational cost and hence energy usage. This holds for the first two groups of cases because evaluation of $s \rightarrow select(P)$, $s \rightarrow reject(P)$ or $s \rightarrow collect(x \mid e)$ requires iteration over all elements of s, whilst evaluation of an expression $s \rightarrow forAll(P)$, $s \rightarrow exists(P)$ or $s \rightarrow any(P)$ may be terminated as soon as a counter-example (for $\rightarrow forAll$) or an example is found (for $\rightarrow exists$, $\rightarrow any$). Thus $\mathcal{C}(r)$ of the refactored version r will always be a subset of $\mathcal{C}(d)$ of the original version d, and can be a strict subset. Likewise, $s \rightarrow count(x)$ will always iterate over all elements of s (in the case of sequences and bags), so formulations using this operator may be more energy-expensive than logically equivalent versions using $\rightarrow includes$, $\rightarrow excludes$, etc. An inexperienced OCL specifier could make the mistake of using the non-optimal version, without appreciating the implications of this choice for implementations of the constraint. In most cases the refactoring also reduces the size and complexity of the constraint (measured as the count $c(e)$ of the number of identifier and operator occurrences in e), and hence improves readability and quality of the constraint. $vars(expr)$ refers to the set of local variable, attribute/reference and parameter names in expression *expr*.

Similarly, we can define energy-use reductions which simplify expressions to avoid unnecessary computations. Tables 2, 3 give some example simplification refactorings of this kind, for the $\rightarrow size()$ and $\rightarrow last()$ operators. Similar refactorings can be defined for other standard OCL operators. The

Table 1
OCL refactorings for energy-use reduction

Original expression	*From*	*Condition*	*Refactored expression*
$col \rightarrow reject(P) \rightarrow size() = 0$		No side-effects in P	$col \rightarrow forAll(P)$
$col \rightarrow reject(P) \rightarrow isEmpty()$		No side-effects in P	
$col \rightarrow select(P) \rightarrow size() = 0$	[13]	No side-effects in P	$col \rightarrow forAll(not(P))$
$col \rightarrow select(P) \rightarrow isEmpty()$	[13]	No side-effects in P	
$s \rightarrow select(P) \rightarrow size() > 0$	[13]	No side-effects in P	$s \rightarrow exists(P)$
$s \rightarrow select(P) \rightarrow size() \geq 1$	[13]	No side-effects in P	
$s \rightarrow select(P) \rightarrow notEmpty()$	[13]	No side-effects in P	
$s \rightarrow reject(P) \rightarrow size() > 0$		No side-effects in P	$s \rightarrow exists(not(P))$
$s \rightarrow reject(P) \rightarrow size() \geq 1$		No side-effects in P	
$s \rightarrow reject(P) \rightarrow notEmpty()$		No side-effects in P	
$s \rightarrow select(P) \rightarrow size() = 1$	[13]	No side-effects in P	$s \rightarrow exists1(P)$
$s \rightarrow reject(P) \rightarrow size() = 1$		No side-effects in P	$s \rightarrow exists1(not(P))$
$s \rightarrow select(P) \rightarrow exists(Q)$		No side-effects in P, Q	$s \rightarrow exists(P \ \& \ Q)$
$s \rightarrow select(P) \rightarrow forAll(Q)$	[13]	No side-effects in P, Q	$s \rightarrow forAll(P \ \Rightarrow \ Q)$
$col \rightarrow select(P) \rightarrow any()$	[14]	No side-effects in P	$col \rightarrow any(P)$
$col \rightarrow select(P) \rightarrow first()$	[14]	No side-effects in P	
$col \rightarrow collect(x \mid e) \rightarrow includes(y)$		$x \notin vars(y)$	$col \rightarrow exists(x \mid y = e)$
$col \rightarrow collect(x \mid e) \rightarrow excludes(y)$		$x \notin vars(y)$	$col \rightarrow forAll(x \mid y \neq e)$
$col \rightarrow collect(x \mid e) \rightarrow sum()$		$x \notin vars(e)$	$e * (col \rightarrow size())$
$col \rightarrow collect(x \mid e) \rightarrow prd()$		$x \notin vars(e)$	$e \rightarrow pow(col \rightarrow size())$
$col \rightarrow count(x) > 0$	[12]	No side-effects in x	$col \rightarrow includes(x)$
$col \rightarrow count(x) = 0$	[12]	No side-effects in x	$col \rightarrow excludes(x)$
$col \rightarrow size() = 0$	[12]		$col \rightarrow isEmpty()$
$col \rightarrow size() > 0$	[12]		$col \rightarrow notEmpty()$
$col \rightarrow at(col \rightarrow size())$	[12]	No side-effects in col	$col \rightarrow last()$

reason for the final case in Table 3 is that *sortedBy* involves $O(n * log\ n)$ comparison and list addition actions, whilst *selectMaximals* involves $O(n)$.

Table 2
OCL simplification refactorings for $\rightarrow size()$

Original expression	*Condition*	*Refactored expression*
$Sequence\{1..n\} \rightarrow size()$		n
$Sequence\{a..b\} \rightarrow size()$	Numeric a, b	$(b - (a) + 1)$
$Sequence\{x_1, ..., x_n\} \rightarrow size()$		n
$Bag\{x_1, ..., x_n\} \rightarrow size()$		n
$s \rightarrow collect(P) \rightarrow size()$		$s \rightarrow size()$
$s \rightarrow including(x) \rightarrow size()$	s sequence or bag	$(s \rightarrow size() + 1)$
$s1 \rightarrow union(s2) \rightarrow size()$	$s1$, $s2$ sequences or bags	$(s1 \rightarrow size() + s2 \rightarrow size())$

AgileUML OCL includes a map type, together with specialised operators for maps, similar to the map type and operators of the Eclipse OCL extension to OCL [22]. Function types and lambda expressions are also included. Table 4 shows some map and lambda expression refactorings for energy-use reduction, where *mp*, *m* are maps of the same type. The first four refactorings also apply to sets, ordered sets, sorted sets and sorted maps. In addition, converting a frequently-called operation to a cached version should generally improve performance and reduce energy-use, without affecting readability (refactoring 17 of [14]).

A specification can also be refactored by replacing a suboptimal data structure by a more efficient or more appropriate data structure (Table 5). Coding to enforce unique membership in a set can be removed, as can the application of $\rightarrow sort()$ to a sorted map or set.

The cases in Table 6 either reduce the number of iterations, or remove duplicated expressions. Thus they should reduce energy use, or leave it unchanged, in addition to improving quality. Short-circuit

Table 3
OCL simplification refactorings for $\rightarrow last()$

Original expression	*Condition*	*Refactored expression*
$sq \rightarrow tail() \rightarrow last()$	Sequence, OrderedSet sq	$sq \rightarrow last()$
$sq \rightarrow front() \rightarrow last()$	Sequence, OrderedSet sq	$sq \rightarrow at(sq \rightarrow size() - 1)$
$sq \rightarrow reverse() \rightarrow last()$	Sequence, OrderedSet sq	$sq \rightarrow first()$
$sq \rightarrow including(x) \rightarrow last()$	Sequence sq	x
$sq \rightarrow append(x) \rightarrow last()$	Sequence sq	x
$sq \rightarrow collect(x \mid e) \rightarrow last()$	Sequence, OrderedSet sq No side-effects in e	$let\ x = sq \rightarrow last()\ in\ e$
$col \rightarrow sortedBy(x \mid e) \rightarrow last()$	Collection col No side-effects in e	$col \rightarrow selectMaximals(x \mid e) \rightarrow any()$

Table 4
OCL simplification refactorings for maps and functions

Original expression	*Condition*	*Refactored expression*
$mp \rightarrow union(m) \rightarrow union(m)$	No side-effects in m	$mp \rightarrow union(m)$
$mp \rightarrow union(mp)$	No side-effects in mp	mp
$mp \rightarrow intersection(m) \rightarrow intersection(m)$	No side-effects in m	$mp \rightarrow intersection(m)$
$mp \rightarrow intersection(mp)$	No side-effects in mp	mp
$mp \rightarrow keys() \rightarrow size()$		$mp \rightarrow size()$
$mp \rightarrow keys() \rightarrow includes(x)$		$mp \rightarrow includesKey(x)$
$mp \rightarrow keys() \rightarrow excludes(x)$		$mp \rightarrow excludesKey(x)$
$mp \rightarrow values() \rightarrow includes(x)$		$mp \rightarrow includesValue(x)$
$mp \rightarrow values() \rightarrow excludes(x)$		$mp \rightarrow excludesValue(x)$
(lambda $x : T$ in $expr) \rightarrow apply(e)$		let $x : T = e$ in $expr$
let $x : T = e$ in $expr$	$x \notin vars(expr)$, e has no side-effects	$expr$
(let $x : T = e$ in A) $\&$ (let $x : T = e$ in B)	e has no side-effects	let $x : T = e$ in $(A \ \& \ B)$

Table 5
OCL data structure refactorings

Original data type	*Replaced by*
Sequence with no use of indexing, *first*, *last*, *front*, *tail*	Replace by *Bag*
Sequence or *Bag* with coding to enforce unique elements	Replace by *OrderedSet* if indexing needed, or by *Set*, *SortedSet*
Set or *Map* with use of $\rightarrow sort()$	Replace with *SortedSet* or *SortedMap*

logical evaluation is recommended in [14] to optimise transformation performance (refactoring 24 of [14]). AgileUML uses short-circuit logical operators $\&$, *or*, $\Rightarrow$ to avoid unnecessary computations in evaluating logical formulae. A conjunction $A \ \& \ B$ with $cost(A) > cost(B)$ could be optimised by re-ordering the expression to $B \ \& \ A$.

Additionally, OCL flaws may be present which imply potential increased energy-use, but there is no single refactoring which can remove them. An example is long reference chains $e1.r1.r2$ and longer chained sequences of references [13], which in an implementation would require iterations and/or chained operation calls. These can be highlighted to the specifier as a quality flaw with potential for increasing energy use, and then manually corrected.

The main focus of the refactoring rules of [12, 13] is on improving the readability and clarity of OCL constraints, to support system comprehension and evolution. Some of these refactorings would appear to have no significant effect upon the energy use of implementations of the constraints. For example, rewriting an expression $P \Rightarrow (Q \Rightarrow R)$ as $(P \ \& \ Q) \Rightarrow R$ [13] would not be expected to affect the constraint energy use. Table 7 gives examples of quality improvement refactorings which should

Table 6
OCL quality and energy-use refactorings

Original expression	*Condition*	*Refactored expression*
$col \rightarrow select(P) \rightarrow select(Q)$		$col \rightarrow select(P \ \& \ Q)$ [13]
$col \rightarrow reject(P) \rightarrow reject(Q)$		$col \rightarrow reject(P \ or \ Q)$
$s \rightarrow forAll(P) \ \& \ s \rightarrow forAll(Q)$		$s \rightarrow forAll(P \ \& \ Q)$
$s \rightarrow exists(P) \ or \ s \rightarrow exists(Q)$		$s \rightarrow exists(P \ or \ Q)$
$(not(P) \ or \ Q) \ \& \ (P \ or \ R)$	P no side-effects	$if \ P \ then \ Q \ else \ R \ endif$ [12]
$(P \Rightarrow Q) \ \& \ (not(P) \Rightarrow R)$	P no side-effects	
$(P \ or \ Q) \ \& \ (not(P) \ or \ not(Q))$	P, Q no side-effects	$P \ xor \ Q$ [12]
$(P \ \& \ not(Q)) \ or \ (Q \ \& \ not(P))$	P, Q no side-effects	
$(P \ or \ Q) \ \& \ (P \ or \ R)$	P no side-effects	$P \ or \ (Q \ \& \ R)$ [12]
$(P \ \& \ Q) \ or \ (P \ \& \ R)$	P no side-effects	$P \ \& \ (Q \ or \ R)$ [12]
$s \rightarrow collect(e) \rightarrow sum() \ + \ s \rightarrow collect(f) \rightarrow sum()$		$s \rightarrow collect(e + f) \rightarrow sum()$ [13]
$s \rightarrow collect(e) \rightarrow prd() \ * \ s \rightarrow collect(f) \rightarrow prd()$		$s \rightarrow collect((e) * (f)) \rightarrow prd()$
$if \ e \ then \ a \& b \ else \ c \& b \ endif$		$(if \ e \ then \ a \ else \ c \ endif) \ \& \ b$

not adversely affect energy use.

Table 7
OCL quality refactorings

Original expression	*From*	*Condition*	*Refactored expression*
$A \Rightarrow (B \Rightarrow C)$	[13]	$(A \ \& \ B) \Rightarrow C$	
$col \rightarrow forAll(x \mid x.r \rightarrow forAll(y \mid P))$	[13]	$x \notin vars(P)$	$col.r \rightarrow forAll(y \mid P)$
$r \rightarrow forAll(x \mid p \rightarrow includes(x))$	[13]		$p \rightarrow includesAll(r)$
$col \rightarrow intersection(s) \rightarrow isEmpty()$			$col \rightarrow excludesAll(s)$
$col \rightarrow intersection(s) \rightarrow size() = 0$			
$col \rightarrow excluding(x) \rightarrow forAll(y \mid P)$		identifier x, $x \neq y$	$col \rightarrow forAll(y \mid y \neq x \Rightarrow P)$

Another flaw of this kind is the use of *oclIsKindOf* and *oclAsType* to explicitly test the specific class type of objects of a superclass and to downcast them to that type [13]. This flaw could occur in operation pre or post conditions:

```
operation op(x : C, ...)
pre:
  (x->oclIsKindOf(C1) => Pre_1) & ... & (x->oclIsKindOf(Cn) => Pre_n)
post:
  (x->oclIsKindOf(C1) => Post_1) & ... & (x->oclIsKindOf(Cn) => Post_n)
```

This can be refactored by defining overloaded versions of the operation:

```
operation op(x : C1, ...)
pre: Pre_1
post: Post_1

...

operation op(x : Cn, ...)
pre: Pre_n
post: Post_n
```

A common cause of inefficient processing in model transformations is the use of iterations over $C.allInstances()$ for a class C [12, 14]. This can be avoided by adding appropriate associations to a model, to directly obtain the relevant sets of C instances for a constraint, instead of searching through all instances (refactoring 23 of [14]). Substituting if-else chains with a map (refactoring 19 of [14]) should generally reduce energy use, provided that each if case is directly replaced by a map access, rather than by a helper call.

Finally, there are refactorings from [12, 13, 14] which improve some measure of quality, but may increase energy use, such as 'Split conditional rules' [13], which replace formulae of the form A *or* $B \Rightarrow C$ by $(A \Rightarrow C)$ & $(B \Rightarrow C)$. This transformation results in a duplicated evaluation of C. Likewise, replacing an expression by an operation call [13] or extracting a transformation helper/rule [14] corresponds to the classic 'Extract operation' refactoring [1], which may increase energy use [17]. Inlining helpers/rules, merging or splitting rules, and inheritance-related refactorings may also increase energy use (refactorings 4 to 16 of [14]).

Most of the energy-use flaws of Tables 1, 2, 3, 4 and 5 are identified automatically by AgileUML, and the corresponding refactorings of Tables 1, 2, 3, 4 are performed automatically (the 'Optimise OCL' refactoring).

4. Design refactorings

To define software designs, AgileUML uses a generalised statement language similar to the extended executable OCL of [23], [13], [24] or [25]. Table 8 summarises the main statement constructs of AgileUML. In the table T is a type, and *expr* is any OCL expression of the appropriate type – in loop and conditional statement tests *expr* must be boolean-valued. s, $s1$ and $s2$ are statements.

Table 8
AgileUML structured activities

Statement	*Meaning*
x := *expr*	Assignment
var *iden* : T := *expr*	Variable declaration
$s1$; $s2$	Sequencing
if *expr* then $s1$ else $s2$	Conditional
while *expr* do s	Unbounded *while* loop
repeat s until *expr*	Unbounded *repeat* loop
for *ident* : *expr* do s	Bounded loop
return	Return statement
return *expr*	Return value statement
(s)	Statement group
break	Break statement
continue	Continue statement
expr	Operation call, *expr* is *obj.op*(*pars*) or *ClassName.op*(*pars*)
execute *expr*	Attempt to establish *expr*
skip	No-op

A number of refactorings apply to this activity formalism, which rewrite or reduce activity statements into simpler and potentially more energy-efficient forms. Table 9 gives examples of these refactorings, which could also be considered as program *reductions* [26]. The rules of Table 2 can be used to simplify the expression $t \rightarrow size()$ in the final two rules.

The energy-use flaws of Table 9 are identified automatically by AgileUML, as are the corresponding refactorings.

5. Evaluation

In this section we evaluate the effect of selected OCL, design and transformation refactorings on energy use. Software energy use is computed in milli-Watt hours (mWh) using the calculator at `https://calculator.green-algorithms.org`, or in Joules (J) using `https://github.com/joular/joularjx`[1]. The evaluation platforms used are (i) Windows 10 OS,

[1] 1 mWh = 3.6 J.

Table 9
AgileUML program reductions

Original	*Conditions*	*Refactored*
$c1$; $c2$	$c1$ ends with break, exit, continue or return.	$c1$
if e then c else c	e has no side-effects	c
if true then $c1$ else $c2$		$c1$
if false then $c1$ else $c2$		$c2$
while false do c		skip
repeat c until true	c has no break, continue	c
for $i : s$ do $r := r + e$	$r \notin vars(e) \cup vars(s), r \neq i$	$r := r + s{\rightarrow}collect(i \mid e){\rightarrow}sum()$
for $i : s$ do $r := r - e$	$r \notin vars(e) \cup vars(s), r \neq i$	$r := r - s{\rightarrow}collect(i \mid e){\rightarrow}sum()$
for $i : s$ do $r := r * e$	$r \notin vars(e) \cup vars(s), r \neq i$	$r := r * (s{\rightarrow}collect(i \mid e){\rightarrow}prd())$
for $i : s$ do $r := r/e$	$r \notin vars(e) \cup vars(s), r \neq i$	$r := r/(s{\rightarrow}collect(i \mid e){\rightarrow}prd())$
for $i : s$ do $r := r \,\&\, e$	$r \notin vars(e) \cup vars(s), r \neq i$	$r := r \,\&\, s{\rightarrow}forAll(i \mid e)$
for $i : s$ do if $cond$ then $r := r \,\&\, e$ else skip	$r \notin vars(e) \cup vars(s), r \neq i$	$r := r \,\&\, s{\rightarrow}forAll(i \mid cond \Rightarrow e)$
for $i : s$ do $r := r$ *or* e	$r \notin vars(e) \cup vars(s), r \neq i$	$r := r$ *or* $s{\rightarrow}exists(i \mid e)$
for $i : s$ do if $cond$ then $r := r$ *or* e else skip	$r \notin vars(e) \cup vars(s), r \neq i$	$r := r$ *or* $s{\rightarrow}exists(i \mid cond \,\&\, e)$
for $x : t$ do $(i := i + 1;\ v := v + i)$	$v \notin vars(t), i \notin vars(t),$ $x \neq v, x \neq i, i \neq v,$ $n = t{\rightarrow}size()$	$v := v + n * i + n * (n + 1)/2;$ $i := i + n$
for $x : t$ do $(i := i + 1;\ v := v * i)$	$v \notin vars(t), i \notin vars(t),$ $x \neq v, x \neq i, i \neq v,$ $n = t{\rightarrow}size()$	$v := v * (n + i)!/i!;$ $i := i + n$

with JDK 8, on a laptop with a 4 core i5-9400 processor, 8GB available memory; (ii) Linux Mint 21.3, JDK 21, on a i7-1365U processor laptop with 10 cores, 16GB available memory.

The average of three energy-use computations is used for each result. JoularJX is used for Linux and GreenAlgorithms for Windows. The evaluation examples and results are available at Zenodo.org[2].

5.1. OCL optimisation

We evaluated the effect of replacing $col{\rightarrow}select(x \mid x * x > 50){\rightarrow}any()$ by $col{\rightarrow}any(x \mid x * x > 50)$ for integer collections *col* of different sizes (Table 10). These show consistent reduction in energy use for both platforms (Figure 1 shows the different versions for Windows 10).

Table 10
OCL expression energy use for “find any element" versions

Collection size	*Windows platform (mWh)*		*Linux platform (J)*	
	Unoptimised	*Optimised*	*Unoptimised*	*Optimised*
10000	0.269	0.008	14.5	10.1
50000	1.09	0.012	20.2	10.5
100000	2.04	0.011	28.3	10.2

Similarly, we evaluated the difference between alternative versions of a “find maximal elements" computation: $col{\rightarrow}sortedBy(x \mid x * x){\rightarrow}last()$ and $col{\rightarrow}selectMaximals(x \mid x * x){\rightarrow}any()$ for integer sequences *col* of varying sizes (Table 11). This shows that the refactoring reduces energy use on both platforms.

[2] zenodo.org/records/15387228

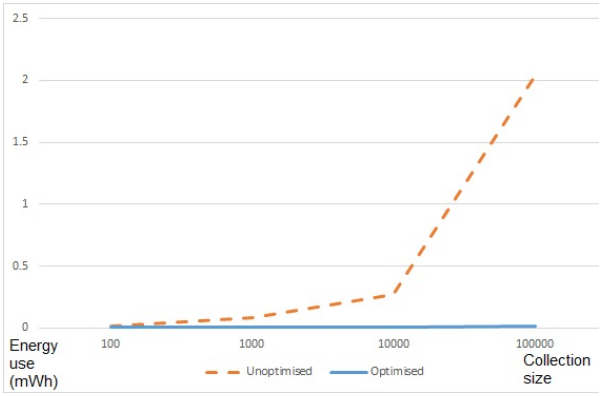

Figure 1: Energy use of OCL “find any element" versions on Windows 10

Table 11
OCL expression energy use for “find maximal elements" versions

Collection size	*Windows platform (mWh)*		*Linux platform (J)*	
	Unoptimised	*Optimised*	*Unoptimised*	*Optimised*
10000	0.168	0.1	11.7	6.3
50000	0.22	0.08	15.9	6.45
100000	0.285	0.124	20.4	7.2

5.2. Transformation optimisation

In [14], alternative transformation designs and refactorings are evaluated for their performance. In particular, it is identified that rules with multiple input elements can be inefficient, because they involve iteration over all instances of multiple class types in a model. Such rules should therefore also be avoided when considering energy use reduction, or optimised using a transformation design pattern such as ‘Restrict input ranges’ from [27]. Inefficient iterations over *allInstances* collections within the body of a rule can be removed by applying the ‘Replace *allInstances* with navigation’ refactoring of [14].

Here we compare the energy use of different versions of a UML-RSDS implementation of the SimpleUML2SimpleRDB transformation from the ATL Zoo. Table 12 gives the energy use in mWh of four versions of the transformation:

1. An unoptimised version.
2. The unoptimised version with the most energy-expensive rule (the final rule) omitted.
3. A version with the final rule optimised by using ‘Replace *allInstances* with navigation’.
4. The optimised version with different data structures: sets instead of sequences for collection-valued association ends.

Table 12
SimpleUML2SimpleRDB MT energy use (mWh)

Model size	*Unoptimised*	*Unoptimised, omit final rule*	*Optimised final rule*	*Optimised, using Sets*
100	0.002	0.003	0.003	0.004
1000	0.028	0.025	0.027	0.047
10000	5.37	4.45	6.57	1.86
100000	893.6	456.6	651.2	553.45

The first two versions and the final optimised version are shown graphically in Figure 2.

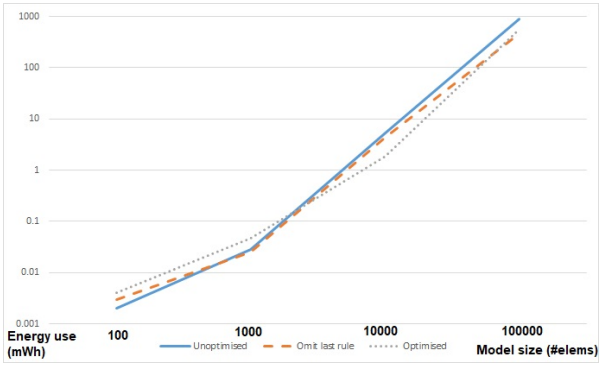

Figure 2: Energy use of SimpleUML2SimpleRDB versions

The energy-expensive rule in this transformation is:

```
UMLClass::
Table->exists( tab |
  tab.tableId = classId &
  tab.fkeys =
          FKey[Association.allInstances()->select( ast |
                    ast.src = self )->collect(associationId)])
```

This involves a double iteration over *UMLClass* instances and then over *Association* instances. The →*select* iteration here computes the inverse of the *src* role linking Association instances to UMLClass instances. Hence (at least) quadratic computational cost in terms of model size is expected. To avoid the double iteration, the 'Replace *allInstances* with navigation' refactoring of [14] can be used. An explicit inverse role *outgoing* of *src* is added to the metamodel, and the rule rewritten to:

```
UMLClass::
Table->exists( tab |
       tab.tableId = classId &
       tab.fkeys = FKey[outgoing.associationId])
```

This restricts the scope of the inner iteration to the *outgoing* associations of the UML class, instead of over all associations in the model. Further optimisation can be achieved by replacing unique-element sequences by sets. The result is a transformation with energy use similar to the reduced version omitting the final rule (Table 12). In the case of the largest model, there is a 38% reduction in the energy-use of the optimised transformation version compared to the original version.

6. Conclusions

In this paper we have surveyed proposed OCL and transformation refactorings, and also introduced further refactorings for the extended OCL notations of AgileUML. We have classified refactorings by their impact on quality and on the reduction of software energy-use, and performed quantitative evaluation to demonstrate the benefits of particular refactorings.

Declaration on Generative AI

The authors declare that no generative artificial intelligence tools were used to generate text, figures, code, data, analyses, or reviews for this submission.

References

[1] M. Fowler, K. Beck, Refactoring: improving the design of existing code, 2nd Edition, Pearson, 2019.

[2] R. Marinescu, Assessing technical debt by identifying design flaws in software systems, IBM Journal of Research and Development 56 (2012).
[3] X. He, P. Avgeriou, P. Liang, Z. Li, Technical debt in MDE: A case study on GMF/EMF-based projects, in: MODELS 2016, 2016.
[4] S. Rahimi, K. Lano, M. Sharbaf, M. Karimi, H. Alfraihi, A comparison of quality flaws and technical debt in model transformation specifications, JSS 169 (2020).
[5] S. Naumann, M. Dick, E. Kern, T. Johann, The GREENSOFT model: a reference model for green and sustainable software and its engineering, Sustainable Computing: Informatics and Systems 1 (2011) 294–304.
[6] M. Tufano, F. Palomba, G. Bavota, R. Oliveto, M. D. Penta, A. D. Lucia, D. Poshyvanyk, When and why your code starts to smell bad, in: ICSE, 2015.
[7] G. Campbell, P. Papapetrou, SonarQube in Action, Manning Publications Co, 2013.
[8] K. Lano, J. Bicarregui, UML refinement and abstraction transformations, in: ROOM 2 workshop, 1998.
[9] L. Bettini, D. D. Ruscio, L. Iovino, A. Pierantonio, Quality-driven detection and resolution of metamodel smells, IEEE Access (2019).
[10] M. Strittmatter, G. Hinkel, M. Langhammer, R. Jung, R. Heinrich, Challenges in the evolution of metamodels: Smells and anti-patterns of a historically-grown metamodel, in: 10th workshop on models and evolution (ME 2016), 2016, pp. 30–39.
[11] S. Kolahdouz-Rahimi, K. Lano, M. Karimi, Technical debt in procedural model transformation languages, Journal of Computer Languages 59 (2020). doi:https://doi.org/10.1016/j.cola.2020.100971.
[12] J. Cabot, E. Teniente, Transformation techniques for OCL constraints, Science of Computer Programming 68 (2007) 179–195.
[13] A. Correa, C. Werner, Refactoring OCL specifications, SoSyM 6 (2007) 113–138.
[14] M. Wimmer, S. Martinez, F. Jouault, J. Cabot, A catalogue of refactorings for model-to-model transformations, Journal of Object Technology 11 (2012) 1–40.
[15] K. Lano, S. K. Rahimi, Model transformation design patterns, IEEE TSE 40 (2014) 1224–1259.
[16] K. Lano, L. Alwakeel, Z. Rahman, Software modelling for sustainable software engineering, in: 2nd Agile MDE Workshop, STAF 2024, 2024.
[17] C. Sahin, L. Pollock, J. Clause, How do code refactorings affect energy usage?, in: ESEM '14, ACM, 2014.
[18] Eclipse, 2025. Agile UML project, https://projects.eclipse.org/projects/modeling.agileuml.
[19] Object Management Group, Object Constraint Language (OCL) 2.4 specification, 2014.
[20] K. Lano, S. Kolahdouz-Rahimi, Extending OCL with map and function types, in: FSEN 2021, 2021.
[21] K. Lano, Adding regular expression operators to OCL, in: OCL 2021, 2021.
[22] Eclipse, 2022. Eclipse OCL Version 6.4.0, https://projects.eclipse.org/projects/ modeling.mdt.ocl.
[23] F. Buttner, M. Gogolla, On OCL-based imperative languages, Science of Computer Programming 92 (2014) 162–178.
[24] Eclipse QVTo project, 2022. Imperative OCL, wiki.eclipse.org/QVTo.
[25] S. Motogna, et al., Extension of an OCL-based executable UML components action language, Informatica LIII (2008) 15–26.
[26] C. Sun, Y. Li, Q. Zhang, T. Gu, Z. Su, Perses: Syntax-guided program reduction, in: ICSE 2018, ACM, 2018.
[27] K. Lano, S. Kolahdouz-Rahimi, S. Yassipour-Tehrani, M. Sharbaf, A survey of model transformation design patterns in practice, JSS 140 (2018) 48–73.