JOT
JOURNAL OF
OBJECT TECHNOLOGY

Using MDE to support sustainable software re-engineering

Kevin Lano[*], **Shekoufeh Kolahdouz Rahimi**[†], **and Zishan Rahman**[*]
[*]King's College London, UK
[†]University of Roehampton, UK

ABSTRACT Re-engineering of legacy software systems is widely used to improve the maintainability of such systems, by migrating them to modernised platforms and environments. With increasing concern over the climate change impact of ICT, there is also a need to consider the energy use of legacy systems, and to identify and remove energy use flaws as part of a re-engineering process. In this paper we describe how energy use analysis and improvement can be carried out at the software model level within a model-driven re-engineering (MDRE) process. Our results show that significant improvements in the energy efficiency of re-engineered applications can be achieved. Additionally, we show that the energy efficiency of the MDRE process itself can be improved.

KEYWORDS Software energy use; Software sustainability; Model-driven engineering; Re-engineering

1. Introduction

Legacy software systems often perform the core business operations of organisations, and re-engineering efforts are therefore usually carried out with the aim of reducing the ongoing costs of maintaining such systems, and of ensuring their continued functioning for the long term (Marco et al. 2018; Khadka et al. 2014; Sneed & Jandrasics 1987; Sneed 2011). Model-driven reverse/re-engineering (MDRE) (Siala et al. 2024) and model-driven modernisation (MDM) (Perez-Castillo et al. 2011) aim to perform re-engineering by reverse-engineering legacy source code artefacts to higher levels of abstraction, i.e., to models in UML or other design/specification representations, and then performing forward-engineering from these models to produce code for the target platform.

An MDRE process facilitates substantial changes in the structure and architecture of legacy software systems, for example, to move from a monolithic architecture deployed on a mainframe, to a service-oriented architecture for cloud deployment (Fuhr et al. 2013; Krasteva et al. 2013). There is evidence that such MDRE projects can improve system quality and maintainability, however there has been a lack of investigation of the impact of MDRE on software sustainability (Naumann et al. 2011), in the sense of reducing the greenhouse gas (GHG) emissions due to the energy used by software execution.

In this paper we describe techniques by which energy use flaws in legacy code can be detected and corrected as part of an MDRE process. We also consider the energy efficiency of the MDRE process itself, and how this can be improved.

1.1. MDRE using AgileUML

The MDRE approach of AgileUML (Lano & Siala 2024a) uses the Concrete Grammar Transformation Language (CGTL) text-to-text transformation language to abstract program code to a semantic representation in UML/OCL (Lano, Haughton, et al. 2024), and then applies various code generators to perform forward engineering. The semantic representation uses extensions of the OCL 2.4 standard (Object Management Group 2014), in particular a procedural extension of OCL, similar to the SOIL formalism (Buttner & Gogolla 2014), is used to represent procedural statements. Libraries for common programming data types and facilities have also been defined (Lano et al. 2022), and map and function types were added to OCL (Lano & Kolahdouz-Rahimi 2021). Sorted collections have also been added to OCL. Details of the semantic representation are given in (Lano &

JOT reference format:
Kevin Lano, Shekoufeh Kolahdouz Rahimi, and Zishan Rahman. *Using MDE to support sustainable software re-engineering*. Journal of Object Technology. Vol. 24, No. 2, 2025. Licensed under Attribution 4.0 International (CC BY 4.0) http://dx.doi.org/10.5381/jot.2025.24.2.a15

Siala 2024b).

An example of VB to Python re-engineering is the following function from a VB option pricing application:

```
FUNCTION D1(S AS Double, K AS Double, Vol AS
    Double, R AS Double, Q AS Double, T AS Double
    ) AS Double
  D1 = (WorksheetFunction.Ln(S / K) +
       (R - Q + Vol * Vol / 2) * T) /
          (Vol * Sqr(T))
END FUNCTION
```

This is automatically abstracted by AgileUML to the corresponding OCL version:

```
operation D1(S : double,K : double,Vol : double,R
     : double,Q : double,T : double) : double
 pre: true post: true
 activity: var D1 : double;
   D1 := (((Excel.Ln((S /(1.0 * K))) +
      ((((R - Q) + ((Vol * Vol) /(1.0 * 2)))) * T
    ))) /
         (1.0 * ((Vol * (T)->sqrt()))));
   return D1;
```

Code generation from this model targeted at Python produces the following Python 3.10 code:

```
class FromVB :
  def D1(self, S, K, Vol, R, Q, T) :
    D1 = 0.0
    D1 = (Excel.Ln(S/(1.0 * K)) +
         (((R - Q) + (Vol * Vol)/(1.0 * 2)) * T))
            /(1.0 * (Vol * math.sqrt(T)))
    return D1
```

An important characteristic of AgileUML is that the code generation process produces executable code with the same structure as the design model procedural OCL. In particular, the cyclomatic complexity (measured as the number of basic logical control flow conditions plus 1) of expressions and code is usually preserved. The operations, parameters and local variables of the model are also retained in the generated code, as is the call graph of the model. Therefore analysis of quality or energy use flaws at the model level based on such structural features will remain valid for generated implementations.

1.2. Energy use analysis

Identification of potentially wasteful energy usage by a software application is critical to improving software sustainability (Pathania et al. 2023). Energy measurement techniques include external power meters, internal (on-chip) power sensors, or energy predictive models based on performance monitoring counters (Fahad et al. 2019; Noureddine 2022). Tools such as Green Algorithms (Lannelongue et al. 2021), mlco2 (Lacoste et al. 2019), Codecarbon[1] and Carbontracker (Anthony et al. 2020) provide estimates of software carbon footprints based on the usage of computational resources and on the location of such resources. However, these tools all measure the energy use of executable code, and for general utility in an MDRE process, it would be preferable to detect possible energy use flaws at the software modelling level. This would predict implementation-level energy use flaws based upon the software models.

[1] https://codecarbon.io

Model-based energy analysis is a new area of research, with only a few published investigations considering this issue (Alves et al. 2020; Brunschwig & Goaer 2024; Duarte et al. 2019; Lano, Alwakeel, & Rahman 2024b,a). In (Lano, Alwakeel, & Rahman 2024b) we introduced the concept of energy use analysis at the software modelling level in AgileUML, and described techniques for reducing software potential energy use by refactorings, choice of data types and optimisation of design and architectural patterns. In this paper we describe extension of the energy use flaw analysis and refactoring techniques of (Lano, Alwakeel, & Rahman 2024b) to consider a wider range of energy use flaws, and we detail how the detection and correction of energy use flaws at the model level is carried out in the AgileUML MDRE process.

1.3. Research questions and paper structure

We will investigate to what extent model-based energy use analysis and improvement can be carried out within an MDRE process, and if there are clear benefits observed from applying model-based energy use analysis and improvement. We also investigate the energy use of MDRE process steps themselves, and what energy use improvements and tradeoffs are involved.

Thus the research questions which we aim to answer are:

- *RQ*1 : Can model-based energy use analysis and optimisation be supported within an MDRE process?
- *RQ*2 : Can using model-based energy use analysis and optimisation in an MDRE approach significantly reduce the energy use of legacy systems?
- *RQ*3 : Can the energy use of the MDRE process itself be reduced, and what tradeoffs arise for such reduction?

Section 2 describes different categories of energy use flaw that can be encountered in legacy code, and how these can be detected in abstracted models. Section 3 describes refactorings that can be applied to models to improve energy efficiency prior to target code synthesis. Section 4 considers the energy use of MDRE processes themselves, and how these can be reduced. Section 5 gives a detailed evaluation of energy use reduction using MDRE using real-world re-engineering cases, Section 6 gives a comparison with related work, and Section 7 limitations and future work.

2. Energy use flaws in legacy code

Legacy code exists in many different programming languages, ranging from COBOL (Sammet 1978) and BASIC (Kurtz 1978) to Python[2]. COBOL is still in widespread use in legacy business applications (Kerner 2023). BASIC, in the form of Visual Basic (VB) and Visual Basic for Applications (VBA) is the main language for defining auxiliary code modules within MS applications such as Excel (Microsoft Corp. 2022). Python has become the principal language used to program machine learning (ML) applications, and it is also used for computationally-intensive numeric processing and data analysis in financial applications and other domains.

[2] python.org

Common problems with legacy systems are (i) unstructured code; (ii) redundant elements and processing; (iii) use of outdated platforms and software/hardware environments. These problems all have implications for the energy usage of re-engineered versions of the systems.

2.1. COBOL

COBOL originated in the late 1950's and early 1960's, before the principles of structured programming were widely accepted. Programs can feature heavy use of GO TO statements (unconditional jumps). While these may be efficient, problems arise in translating such coding into energy-efficient structured code in modern languages such as Java. A statement

```
GO TO P.
```

in COBOL source code would typically be abstracted to an operation call $P()$, representing the functionality encountered by a 'fall-through' to paragraph or section P, and this call is then translated to a method call in Java. Recursive P operations arise when backwards jumps are used to define loops in source code, for example in the COBOL code:

```
  MOVE 1 TO SUB.
SUM-AMTS.
  ADD AMOUNT(SUB) TO TOTAL.
  ADD 1 TO SUB.
  IF SUB LESS THAN 13
    GO TO SUM-AMTS.
  STOP RUN.
```

This idiom is commonly used with input file processing in COBOL. The abstraction of the above code in procedural OCL is:

```
  SUB := 1;
  self.SUM_AMTS();
```

where *SUM_AMTS* is a recursively defined operation:

```
operation SUM_AMTS()
activity:
  TOTAL := TOTAL + AMOUNT->at(SUB) ;
  SUB := SUB + 1 ;
  if SUB < 13 then self.SUM_AMTS() else skip;
  OclProcess.exit();
```

The translation of the operation to Java or another programming language would retain this algorithmic structure and hence would also be recursive. A similar strategy of translating GO TO statements to recursive operations is carried out by the approach of (Sneed 2011).

In general, recursively-defined operations can be energy-expensive, because computational effort is expended by processing each call to create an activation record (Gries 1971). There are also memory overheads, with a stack of activation records being maintained for the recursively-nested calls. Recursive and mutually-recursive operations can be detected at the model level by identification of loops within the call-graph of the system.

COBOL business systems typically use batch processing on sequential file data. Direct translation of such processing to operations on relational database representations of the data can result in poor and even infeasible performance (Marco et al. 2018). Instead, we adopt the approach of loading files into memory and operating directly upon these (represented as sequences of records). This strategy has been shown to reduce performance problems, but also limits the extent of concurrent processing of the batch files (Marco et al. 2018).

2.2. Visual Basic

Because VB6/VBA also has a GOTO statement, similar control flow translation problems can arise from VB6/VBA legacy code. In addition, due to system evolution, legacy VB code may also contain redundant operation parameters: parameters which are not actually used in the body of the operation. These can result in redundant computations – the copying of actual parameter values to formal parameters, and to additional memory use. Similarly, there can be unused local variables or operations. As with recursion, the presence of such flaws can be detected in the UML/OCL abstraction of the source code.

VBA is often used in financial spreadsheet applications to define application-specific functions (macros). Poor coding practices and energy use flaws can arise when these macros are written by end users who lack formal programming expertise (Mann 2023). Macro VBA code will also typically utilise calls to Excel worksheet functions (as in the example of Section 1.1), and this may result in inefficiencies compared to natively-coded functionality. The reverse engineering process of AgileUML replaces worksheet function calls by calls to an OCL library, *Excel*. In generated implementations, a natively-coded implementation of this library is provided.

2.3. Python

Python has a reputation as a language that can be successfully used by relatively inexperienced programmers. Unfortunately this ease of use may result in such programmers creating programs which are simple in structure and easy to understand, but which are quite inefficient, containing redundant or duplicated computations.

A typical case of redundant computation is where an operation returns a list or tuple, but only one item from the result is actually used:

```
op(pars)[i]
```

where i is a variable-free expression evaluating to an integer. The computation of the other elements of the $op(pars)$ result is therefore redundant. This situation can be detected in the UML/OCL abstraction of the code, since (in the case $i \geq 0$) the call will be abstracted to

```
op(apars)->at(i + 1)
```

in OCL, where *apars* abstracts *pars*.

Duplicated computations may also occur, where a complex expression *expr* occurs in two or more places within a function *op*, and with *expr* having the same value in each location, because its variables $vars(expr)$ are not in the write frame of *op*:

$$vars(expr) \cap wr(op) = \varnothing$$

Such a flaw in source code will be abstracted to the same flaw in UML/OCL, and hence will be detected by AgileUML.

Another case of duplicated expression evaluation occurs with loop-constant expressions: complex expressions *expr* in the body of a loop *s*, whose value is the same on each loop iteration:

$$vars(expr) \cap wr(s) = \varnothing$$

This can also be identified by energy-flaw analysis in the abstracted UML/OCL.

Inexperienced programmers may also use inappropriate datatypes for variables. A common flaw is to misuse lists as sets, by always checking membership of an element in a list before adding it, for example in the Python code:

```
if x in lst :
  pass
else :
  lst.append(x)
```

This leads to excessive computational effort compared to a version using a set. Flawed code of this form abstracts to code with the same structure and with the same flaw in OCL:

```
if lst->includes(x)
then skip else
  lst := lst->including(x)
```

Thus this flaw in the source code can also be detected in the abstracted model.

2.4. Case studies

As a COBOL case study, we use the first example of (Gandhi et al. 2024). As a VB6/VBA re-engineering case, we use a suite of routines for pricing financial products, which originated in the 1990's, but have subsequently been extended and are still used in an investment bank. These include option pricing and bond pricing routines. The example of Section 1.1 is an extract from the option pricing application. As a case study in Python re-engineering we also consider a numerical processing application, for bond analysis and pricing, from the finance domain.

3. Energy use analysis and improvement

Once a source application has been abstracted to UML/OCL, it can be analysed for specific energy use flaws and refactored or restructured to remove these flaws. Performing restructuring at the specification level means that the improved structure applies to every generated implementation across multiple different target platforms.

3.1. Model-based energy use analysis

AgileUML analyses UML/OCL specifications to detect specification quality flaws analogous to 'code smells' in the sense of (Fowler & Beck 2019; He et al. 2016). These can also be regarded as *technical debt* indicators (Marinescu 2012). Energy use flaws can also be identified using different measures and additional 'smells' which indicate potential energy use hotspots and errors of different levels of criticality (red or amber). Category 'red' flaws are those which are definitely sub-optimal and can be optimised by refactoring, whilst category 'amber' flaws are potentially non-optimal. Analogously to technical debt, these flaws can be considered to be indicators of *(environmental) sustainability debt* – i.e., of negative environmental impact embedded in the code (Betz et al. 2015).

The current version (2.4) of AgileUML incorporates energy use analysis and improvement at the specification and design model levels (Lano, Alwakeel, & Rahman 2024b). This involves the computation of relevant metrics which can serve as indicators of sustainability (energy use) debt, and the identification of specific 'bad smells' that indicate energy use flaws. Because AgileUML preserves code and expression structure from the design model level to generated code, these identified model-level flaws will also predict the existence of flaws in generated code in multiple target programming languages.

Relevant metrics and sustainability debt indicators are given in Table 1, adapted from the technical debt indicators of (Rahimi et al. 2020). The metric $c(expr)$ is the number of identifier occurrences plus operator occurrences in *expr*. The specific thresholds such as $c(expr) > 10$ for expression clone size can be varied as appropriate for particular systems.

Specification model elements that have energy flaws can be refactored for improved energy efficiency using the specific refactoring techniques given in Table 2. The criticality level of the flaw facilitates prioritisation of different refactoring actions.

EPL (excessive parameter length) above a threshold value (such as 5) can result in increased computational cost for invocation of the operation, in addition to poor comprehensibility. High CC (cyclomatic complexity) and MEL (maximum expression length) for an operation can result in high computational cost for execution of the operation, in addition to increasing testing and maintenance effort. EFO (excessive fan-out) over a threshold for an operation implies increased resource use by the operation execution, in addition to maintenance costs due to high dependencies. $\text{CBR}_2 > 0$ (cyclic dependency) implies the existence of self or mutual recursion between operations. LCE (loop constant expressions) refers to expressions such as $s{\rightarrow}max()$ in an OCL iterator

$$col{\rightarrow}select(p \mid p > s{\rightarrow}max())$$

which have the same value for each $p \in col$. These are an indicator of redundant computations. DEV > 0 or UVA > 0 also imply that there are redundant computations, and may also impair maintenance and comprehension. OES > 0 implies that there are expression computations with excessive energy use. Tables 3 and 4 give some examples of OES flaws.

Relevant OCL optimisation refactorings from (Cabot & Teniente 2007; Correa & Werner 2007; Wimmer et al. 2012) are given in Tables 3, 4. The motivation for replacing the left hand expression by the right in the cases of Table 3 is that $s{\rightarrow}select(P)$ or $s{\rightarrow}reject(P)$ always involves an iteration over all elements of *s*, whilst evaluation of an expression $s{\rightarrow}forAll(P)$, $s{\rightarrow}exists(P)$, $s{\rightarrow}one(P)$ or $s{\rightarrow}any(P)$ may be terminated as soon as a counter-example (for $\rightarrow forAll$, $\rightarrow one$) or an example of *P* is found (for $\rightarrow exists$, $\rightarrow any$).

The cases in Table 4 either reduce the number of iterations, or remove duplicated expressions. AgileUML uses short-circuit logical operators &, *or*, $\Rightarrow$ to avoid unnecessary computations in logical formulae.

Indicator	Description	Definition
EPL	Excessive parameter list	Number of operation parameters.
CC	High cyclomatic complexity	Number of basic control flow conditions in an operation + 1.
EFO	Excessive fan-out	Number of different operations called from one operation.
CBR_2	Cyclic dependencies	Number of operations with self-dependencies either directly or indirectly in call graph.
MEL	Maximum expression length	Maximum $c(expr)$ within an operation.
DEV	Duplicated expression evaluation	Number of cloned *expr* with $c(expr) > 10$ and with unchanged value across clones.
LCE	Loop constant expressions	Number of iterator body expressions *expr* with $c(expr) > 10$ and *expr* independent of iterator variables.
OES	OCL efficiency smell	Number of inefficient OCL expressions.
UVA	Unused parameters/local variables	Number of unused parameters/variables of operation.

Table 1 Sustainability debt indicators.

Flaw	Level	Refactoring
Self-recursive operation.	Red	Replace tail-recursion by iteration; or make operation $\ll cached \gg$.
Mutually-recursive operations.	Red	Replace calls by definition for one operation.
Using sq : *Sequence* with tests to enforce unique membership.	Red	Replace *Sequence* by *Set* or *SortedSet* if no sq indexing used, otherwise by *OrderedSet*.
DEV or LCE of complex expression.	Red	Replace by new local constant v and lookups of v.
Redundant result computation $op(pars) \rightarrow at(i)$	Red	Define specific operations for individual results.
OES flaw.	Red/ Amber	Replace by optimised version.
Unused operation parameter.	Amber	Remove parameter in declaration and calls.
Long chains of method calls.	Amber	Replace call of chain end operation by its definition.

Table 2 Specification energy use flaws and refactorings.

Non-optimal expression	Refactored expression
$col \rightarrow reject(P) \rightarrow size() = 0$	$col \rightarrow forAll(P)$
$col \rightarrow reject(P) \rightarrow isEmpty()$	
$col \rightarrow select(P) \rightarrow size() = 0$	$col \rightarrow forAll(not(P))$
$col \rightarrow select(P) \rightarrow isEmpty()$	
$s \rightarrow select(P) \rightarrow size() > 0$	$s \rightarrow exists(P)$
$s \rightarrow select(P) \rightarrow notEmpty()$	
$s \rightarrow reject(P) \rightarrow size() > 0$	$s \rightarrow exists(not(P))$
$s \rightarrow reject(P) \rightarrow notEmpty()$	
$s \rightarrow select(P) \rightarrow size() = 1$	$s \rightarrow one(P)$
$s \rightarrow reject(P) \rightarrow size() = 1$	$s \rightarrow one(not(P))$
$s \rightarrow select(P) \rightarrow exists(Q)$	$s \rightarrow exists(P \,\&\, Q)$
$s \rightarrow select(P) \rightarrow forAll(Q)$	$s \rightarrow forAll(P \Rightarrow Q)$
$col \rightarrow select(P) \rightarrow any()$	$col \rightarrow any(P)$
$col \rightarrow select(P) \rightarrow first()$	

Table 3 OES flaws (red) and refactorings.

Non-optimal expression	Refactored expression
$col \rightarrow select(P) \rightarrow select(Q)$	$col \rightarrow select(P \,\&\, Q)$
$col \rightarrow reject(P) \rightarrow reject(Q)$	$col \rightarrow reject(P \text{ or } Q)$
$s \rightarrow forAll(P)$ & $s \rightarrow forAll(Q)$	$s \rightarrow forAll(P \,\&\, Q)$
$s \rightarrow collect(e) \rightarrow sum() +$ $s \rightarrow collect(f) \rightarrow sum()$	$s \rightarrow collect(e + f) \rightarrow sum()$
$s \rightarrow collect(e) \rightarrow prd() *$ $s \rightarrow collect(f) \rightarrow prd()$	$s \rightarrow collect((e) * (f)) \rightarrow prd()$
if e then a&b else c&b endif	$(\textit{if e then a else c endif})$ & b

Table 4 OES flaws (amber) and refactorings.

Likewise, additional energy use flaws may arise from design model elements and coding (Table 5). Some example program reduction transformations are given in Table 6. These refactorings may improve design quality in terms of improved comprehensibility and reduced complexity, in addition to reducing energy use.

Flaw	Level	Refactoring
while loops with *true* condition and no loop exit.	Red	Ensure termination by *break*, *exit* or *return*.
repeat loops with *false* condition and no loop exit.	Red	Ensure termination by *break*, *exit* or *return*.
LCE in loop body.	Red	Replace by local constant and lookups.
Nested loops.	Amber	Restrict loop ranges (Lano et al. 2018); Optimise inner loop (Gries 1971).
while or *repeat* loops.	Amber	Replace by bounded loop.
Use of reflection, process creation, remote method calls.	Amber	Replace where possible.
High CC or code nesting depth; Redundant code.	Amber	Apply program reduction (Sun et al. 2018).

Table 5 Design-level energy use flaws and refactorings.

3.2. Refactoring for energy use improvement

Some refactorings from Tables 2, 3, 4, 5 and 6 can be automated as Java-coded model transformations on AgileUML models. The implemented specification refactorings are: Replacing recursion by iteration or by caching of operation results; Introducing a new local constant for cases of DEV or LCE; the OES refactorings of Table 3; Removing unused operation parameters; Replacing an operation call by its definition. At the design level, removal of LCE cases in loop bodies by introducing a new local constant, and program reduction transformations, are automated.

For example, the transformation of a tail-recursive operation in OCL:

```
operation op(fpars) : T
activity:
  C1 ;
  if E1 then return op(apars) else return value ;
```

where $C1$ contains no call of *op*, to:

```
operation op(fpars) : T
activity:
  while true
  do
    (C1 ;
     if E1
     then
       (fpars := apars ; continue)
     else return value) ;
```

Original	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 $c2$	$c2$
if e then $c1$ else $c2$ $c1$ ends with break, exit, continue or return.	if e then $c1$ else skip; $c2$
while false do c	skip
repeat c until true c has no break, continue	c
for $i : s$ do $r := r + e$ $r \notin vars(e), r \neq i$	$r := r + s{\rightarrow}collect(i \mid e){\rightarrow}sum()$
for $i : s$ do $r := r * e$ $r \notin vars(e), r \neq i$	$r := r * (s{\rightarrow}collect(i \mid e){\rightarrow}prd())$

Table 6 Program reductions.

is a special case of an implemented refactoring rule which replaces the body of the recursive operation *op* by a while loop, and each recursive call of *op* by the assignment of actual parameters to formal parameters, followed by *continue*. The restructured code should have lower energy use, because it does not involve the creation and stacking of activation records. Applied to the COBOL example of Section 2, this refactoring produces the improved OCL design:

```
operation SUM_AMTS()
activity:
  while true
  do
    (TOTAL := TOTAL + AMOUNT->at(SUB) ;
     SUB := SUB + 1 ;
     if SUB < 13
     then continue else skip;
     OclProcess.exit());
```

However, some refactorings require manual intervention. For example, to replace $op(pars){\rightarrow}at(i)$ by a call to a new specialised operation op_i that produces the i'th result of *op*, requires manual coding of op_i, in general.

It is important to notice that general refactorings for quality-improvement, such as 'Extract operation', may increase energy use (Sahin et al. 2014), thus only certain specific refactorings may be used for energy use reduction. Similarly, some OCL refactorings from (Cabot & Teniente 2007; Correa & Werner 2007; Wimmer et al. 2012), such as removing chained implications in a formula, or replacing $sq{\rightarrow}indexOf(x) > 0$ by

$sq \rightarrow includes(x)$, are intended to improve readability and specification clarity, and do not necessarily improve energy efficiency. In this paper we have identified important cases of refactorings which both improve quality and energy efficiency. There may also be situations where these goals conflict, and refactorings such as inlining operation calls could be chosen to improve energy efficiency, whilst negatively affecting quality.

3.3. Forward engineering

From the abstracted and refactored UML/OCL representation, code can be generated by AgileUML in multiple target languages: ANSI C, C++, C#, Java, Go, Swift and Python. The Java, C++ and C# generators of AgileUML are written in Java. The C and Python generators are written in a model transformation (MT) language, UML-RSDS (Lano et al. 2017). The Go and Swift code generators are written in CGTL and can be directly configured by end-users.

4. Improvements to the energy-efficiency of MDRE

A MDRE process typically consists of a chain of several successive stages (Deltombe et al. 2012; Lano, Haughton, et al. 2024):

1. Parsing of source code into abstract syntax trees (ASTs).
2. Application of an abstraction transformation to map from ASTs to a semantic model.
3. Analysis and restructuring of the model.
4. Forward engineering to a target platform.

Each of these steps can potentially be optimised to improve their energy-efficiency. Steps 1, 2 and 4 are usually automated, step 3 may be interactive.

With regard to step 1, we have adopted ANTLR[3] as a relatively efficient and lightweight parsing tool, which is widely-used and which supports a large number of different source languages. With regard to step 2, we can consider two alternatives:

1. Explicitly-coded Java transformations abstracting from ASTs to UML/OCL models;
2. Declarative CGTL transformation specifications of the abstraction mappings, which are expressed as rules from the source concrete syntax to the target concrete syntax.

At present, there are Java-coded abstractors for C, Java and JavaScript, and CGTL-coded abstractors for Java, Pascal, VB, COBOL and Python.

In principle, a CGTL script or UML-RSDS transformation could be used to implement refactoring actions for step 3. However at present, these are only implemented in Java.

With regard to step 4, there are the following alternatives:

1. Explicitly-coded Java transformations generating code from UML/OCL models;
2. Code generators specified in the UML-RSDS MT language (Lano et al. 2017);
3. Declarative CGTL transformation specifications of code generation mappings (Lano, Xue, & Haughton 2024).

In Section 5 we compare the energy use of these alternatives for steps 2 and 4.

5. Evaluation

In this section we provide evidence for RQ1, RQ2, RQ3 in the context of COBOL, VB6/VBA and Python re-engineering tasks. All data of the evaluation cases is provided on Zenodo[4].

We evaluate software energy use in milli-Watt hours (mWh) using the calculator at `https://calculator.green-algorithms.org`. The main evaluation configuration used is Windows 10 OS, with JDK 8, Python 3.10, on a laptop with a 4 core i5-9400 processor, UK location, 8GB available memory. The varying inputs to the computation are the processing time in ms, processor utilisation percentage and memory use (in GB). The average of three energy use values is used for each result, with the computer in low-power mode and with a stable background energy use level.

5.1. RQ1, RQ2: Energy use reduction by MDRE

To answer these questions we applied the AgileUML re-engineering process to three typical re-engineering tasks: (i) translation of COBOL code to Python; (ii) translation of VB6/VBA code to Python; (iii) translation of Python code to Java. In each case energy use analysis, and refactoring for energy use improvement, were applied to the abstracted code.

According to (Georgiou et al. 2017; Pereira et al. 2017), VB is somewhat more energy-efficient than Python, whilst Java is significantly more energy-efficient than Python. However, the problems considered in (Georgiou et al. 2017) to compare VB and Python execution do not include the use of VB embedded in Excel applications. The version of Python used in (Georgiou et al. 2017) is 2.7, whilst we use version 3.10. A more recent study, (Marini et al. 2025), found that Python was more than 3 times more energy-expensive than Java for ML training and inference, although this varied depending on the ML algorithm being executed.

5.1.1. COBOL Re-engineering

We applied AgileUML to reverse-engineer the first COBOL example of (Gandhi et al. 2024):

```
PROCEDURE DIVISION.

ACCEPT INP.
PERFORM ABLEN TIMES
  PERFORM VARYING J FROM CUR BY 1
  UNTIL INP ( J : 1 ) = SPACE
  END-PERFORM
  COMPUTE LEN = J - CUR
  MOVE INP ( CUR : LEN ) TO AB11 ( I )
  COMPUTE CUR = J + 1
  ADD 1 TO I
```

[3] www.antlr.org

[4] zenodo.org/records/14900982

```
END-PERFORM.

COMPUTE DIV = AB11 ( 2 ) - AB11 ( 1 ) .
MOVE 0 TO S1 .
MOVE 1 TO I .

PERFORM DIV TIMES
  ADD I TO S1
  ADD 1 TO I
END-PERFORM.

COMPUTE T = S1 - AB11 ( 2 ) .
MOVE T TO ANS .
DISPLAY ANS .
STOP RUN.
```

This program is abstracted to an OCL specification, and energy use analysis applied. Flaws of nested loops and a potential code reduction are detected, i.e., the second PERFORM TIMES loop can be replaced by the two assignments

```
S1 := S1 + DIV*I + (DIV*(DIV - 1))/2 ;
I := I + DIV ;
```

Applying this code reduction, and then generating Python from the OCL specification, we obtain:

```
def ABC_099_B(self) :
  self.INP = getOclFileByPK("System.in").readLine
    ()

  for _performTimes in range(1, self.ABLEN +1) :
    self.J = self.CUR
    while not (self.INP[(self.J-1):self.J + 1 -
    1] == " ") :
      self.J = self.J + 1
    self.LEN = self.J - self.CUR
    self.AB11[self.I -1] = ocl.toInteger(self.INP
    [(self.CUR-1):self.CUR + self.LEN - 1])
    self.CUR = self.J + 1
    self.I = (self.I + 1)
  self.DIV = (self.AB11[2 -1] - self.AB11[1 -1])
  self.S1 = 0
  self.I = 1
  self.S1 = self.S1 + self.DIV*self.I + (self.DIV
    *(self.DIV - 1))//2
  self.I = self.I + self.DIV
  self.T = (self.S1 - self.AB11[2 -1])
  self.ANS = self.T
  print(("" + str(self.ANS)))
```

In Table 7 we compare the energy use of the original COBOL code with the usage of the direct translation of the program to Python, and the usage of the refactored Python code. The Python energy use was estimated using execution on the Windows 10 i5-9400 laptop described above, however the COBOL execution was carried out on a Google cloud server.

This shows a consistent reduction in energy use for the refactored Python version compared to the directly translated version. Because of the different hardware platform used for execution of the original COBOL version, it is difficult to compare the energy use of this version to the re-engineered versions. We also compared the energy use of COBOL execution using GnuCOBOL[5] and Python 3.10 on a Linux Mint 22.1 laptop. The results are shown in Figure 1.

[5] https://gnucobol.sourceforge.io

Input size	Energy use (mWh)		
	COBOL	Translated Python	Refactored Python
99	0	0.0023	0.0015
999	0	0.0026	0.0014
9999	0.052	0.0067	0.0015
99999	0.104	0.0646	0.0022

Table 7 COBOL to Python evaluation case: energy use reductions.

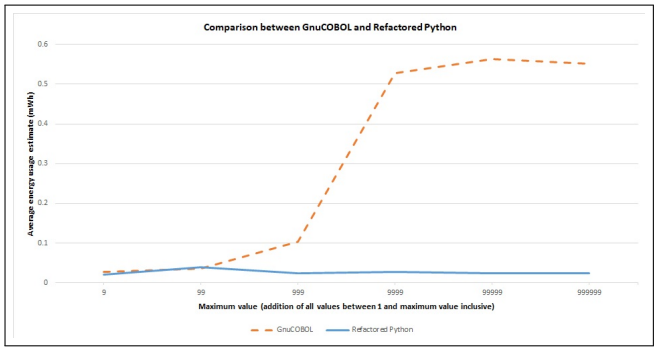

Figure 1 Energy use of COBOL case versions

5.1.2. VB6/VBA Re-engineering

We consider three parts of the VBA case study, which is a financial analysis suite:

1. Binary tree estimation of option prices.
2. Routines for computing factorial and combinatorial functions.
3. A routine for binary search.

The first routine is already in an energy-efficient iterative form, but (2) and (3) are defined recursively rather than iteratively. We produce direct translations of the VB programs, and also refactored Python versions of (2) and (3) to remove the energy use flaw of recursively-defined operations. In the case of the factorial function, because it is not tail-recursive, it is given a cached implementation instead of being replaced by an iterative version. The source code in VB6 is:

```
FUNCTION Factorial(N AS Integer) AS LongLong
  IF N <= 1 THEN Factorial = 1 ELSE Factorial =
    Factorial(N - 1) * N
END FUNCTION
```

This is abstracted to the OCL design:

```
operation Factorial(N : int) : long
pre: true post: true
activity:
  var Factorial : long;
  if N <= 1 then (
    Factorial := 1 )
  else (
    Factorial := (Factorial((N - 1)) * N) ) ;
  return Factorial;
```

Refactoring at the model level simply consists of attaching the stereotype ≪*cached*≫ to this operation. This has the effect of producing cached implementations of the operation in each target programming language. The generated code in Python is:

```
def Factorial(self, N) :
  if str(N) in self.Factorial_cache :
    return self.Factorial_cache[str(N)]
  result = self.Factorial_uncached(N)
  self.Factorial_cache[str(N)] = result
  return result

def Factorial_uncached(self, N) :
  Factorial = 0
  if N <= 1 :
    Factorial = 1
  else :
    Factorial = (self.Factorial((N - 1)) * N)
  return Factorial
```

This implementation ensures that the recursive computation of $factorial(n)$ is only performed once for each n.

Table 8 shows the average energy use of the VBA versions over a range of inputs, and the average energy use of the directly translated and re-engineered Python versions for the same inputs. MS Office version 16 and VBA version 7.1 is used to execute the VBA code, and Python 3.10 for the Python code.

Case	Input size	Energy use (mWh)		
		VBA	Translated Python	Refactored Python
Binomial	100	0.16	0.002	–
Option	1000	0.277	0.003	–
Pricing	10000	0.413	0.013	–
Factorial/	100000	4.25	6.69	2.54
Combinatorial	200000	10.9	14.2	7.47
	300000	19.5	24.3	9.96
Binary	1000	0.169	0.0033	0.001
Search	10000	1.22	0.009	0.0023
	100000	2.93	0.02	0.007

Table 8 VB6 to Python evaluation cases: energy use reductions.

These results show a consistent reduction in energy use between the source and target versions of these systems. However, these cases have different characteristics. For the first case, identical algorithms are used in the source and target versions, and the gain in energy efficiency (by a factor of 30 to 80 times) appears to be due to the change of programming language and environment, in particular, the original version uses multiple calls to Excel worksheet functions, whilst the re-engineered version instead uses calls to Python library functions (an example function of this case is shown in Section 1.1). The second case involves only numeric processing within VB, and a direct translation to Python is marginally less efficient. Introducing caching of *Factorial* then reduces energy use below the source program level (Figure 2).

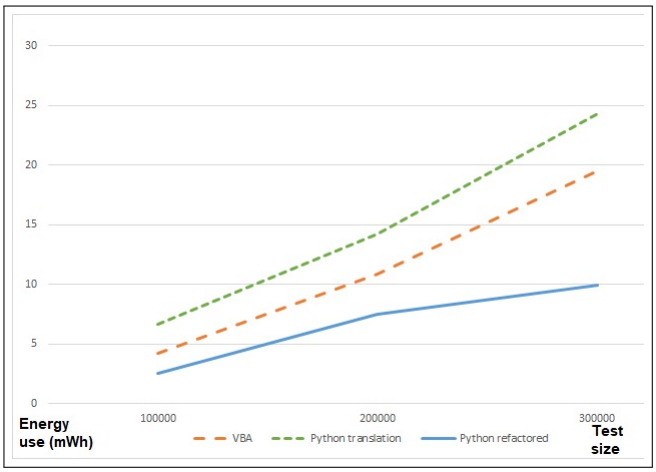

Figure 2 Energy use of Factorial/Combinatorial case versions

The third case involves the processing of large collections (arrays), and the direct translation is substantially more efficient than the source (improvement by a factor of over 100 times in the largest case). This difference could be explicable by the use of default BYVAL value parameter passing of the arrays in the VB code (by-value passing is the default in some versions of VB, but not in VBA). Replacing recursion by iteration further improves energy efficiency by a factor of 3 (Figure 3).

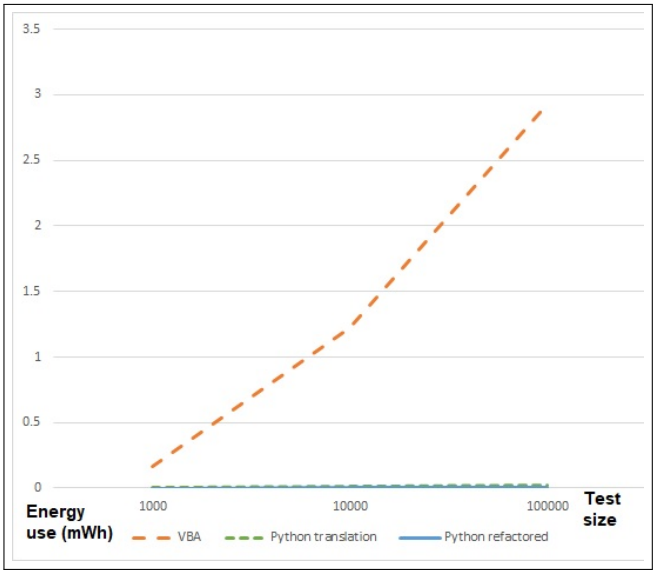

Figure 3 Energy use of Binary Search case versions

5.1.3. Python Re-engineering The case study for Python to Java re-engineering is a suite of functions for bond pricing. These functions contain several energy use flaws, including computation of redundant operation results (6 instances), and duplicated complex expression evaluations (12 instances). Of these 18 flaws, 16 are detected by the flaw analysis, and the 12 cloned expressions are automatically refactored out into new local constants at the model level, prior to generation of Java code. The redundant operation result cases are manually

removed. Table 9 shows the energy use of the original and re-engineered code, applied to datasets of different sizes.

Case	Input size	Energy use (mWh)		
		Python	Translated Java	Refactored Java
Bond	1000	0.021	0.058	0.013
Functions	10000	0.33	0.231	0.062
1	100000	3.34	1.87	0.428
Bond	1512	0.03	0.12	0.045
Functions	14112	0.61	0.58	0.16
2	140112	6.49	12.38	1.35

Table 9 Python to Java evaluation cases: energy use reductions.

The bond functions 1 case consists of date processing functions *MaxDate*, *Prevd*, *Nextd*, which contain multiple duplicated evaluation and redundant result computation flaws. The functions depend heavily upon basic functions *day*, *month* and *year*, which are called frequently in the Python source code:

```
def comp(date_str):
  return date_str.split('/')

def year(date_str):
  return int(comp(date_str)[2])  # used 6 times

def month(date_str):
  return int(comp(date_str)[1])  # used 9 times

def day(date_str):
  return int(comp(date_str)[0])  # used 9 times
```

These functions are abstracted to the following OCL functions:

```
operation comp(date_str : OclAny) : OclAny
pre: true post: true
activity:
  return date_str->split('/');

operation year(date_str : OclAny) : int
pre: true post: true
activity:
  return ("" + comp(date_str)->at(2+1))->
    toInteger();

operation month(date_str : OclAny) : int
pre: true post: true
activity:
  return ("" + comp(date_str)->at(1+1))->
    toInteger();

operation day(date_str : OclAny) : int
pre: true post: true
activity:
  return ("" + comp(date_str)->at(0+1))->
    toInteger();
```

The definitions of *year*, *month* and *day* are recognised as cases of the 'redundant operation results' energy use flaw, and manual optimisation of these produces the improved OCL definitions:

```
operation year(date_str : String) : int
pre: true post: true
activity:
  var lsub : int := date_str->lastIndexOf("/");
  return date_str.subrange(lsub+1)->toInteger();

operation month(date_str : String) : int
pre: true
post:
  result = date_str->after("/")->before("/")->
    toInteger();

operation day(date_str : String) : int
pre: true
post:
  result = date_str->before("/")->toInteger();
```

Translation of the original version to Java results in a small improvement in energy efficiency compared to the source program, while generation of Java from the refactored version improves energy efficiency over the source by a factor of 7.8 times lower energy use in the largest test case (Figure 4).

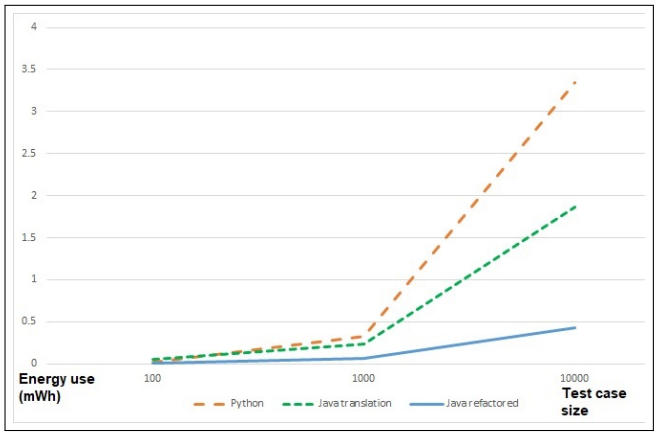

Figure 4 Energy use of Bond Functions 1 versions

Bond functions 2 has similar characteristics, but involves more complex iterative processing. The Java translation in this case is more energy-expensive than the source program, but the refactored version improves energy efficiency compared to the source by a factor of about 5 in the largest test (Figure 5).

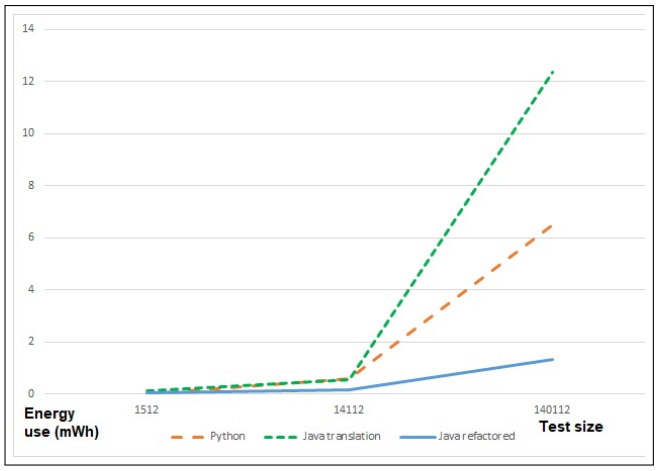

Figure 5 Energy use of Bond Functions 2 versions

5.1.4. *Summary for RQ1, RQ2*

For RQ1 we can conclude from the above cases that enhancing an MDRE process with model-based energy use analysis and improvement is feasible. The analysis techniques detected 92% of energy use flaws present in the case studies, and 72% of the flaws could be automatically corrected (Table 10).

Flaw	Count	Detected	Automatically refactored
Recursive operations	2	2	2
Redundant operation results	6	4	0
Duplicate expression evaluations	12	12	12
Unused operation parameters	3	3	3
Nested loops	1	1	0
Program reductions	1	1	1
Total	25	23 (92%)	18 (72%)

Table 10 Energy use flaws detected and refactored.

Moreover, for RQ2 we conclude from the cases of Tables 7, 8, 9, that such an enhanced MDRE process can achieve energy use reductions for the re-engineered versions of legacy systems. However the degree of improvement varies depending upon the characteristics of the source program and the extent to which energy use flaws are present.

5.2. RQ3: Energy use reduction of MDRE processes

To evaluate different MDRE processes, we compare Java-coded and CGTL versions of program abstractors, and Java-coded, MT-coded and CGTL versions of code generators. UML-RSDS has not been applied for code abstraction, so is omitted from the comparison of Table 11.

For reverse engineering, the Java2UML Java-coded program abstractor for Java reverse-engineering was compared with the cgJava2UML.cstl CGTL abstractor script with respect to their energy use when applied to four Java source programs of varying size and complexity (Table 11). Input program size is measured in lines of code (LOC). Each abstractor takes as input a textual abstract syntax tree representation of the source program, as produced by the ANTLR Java parser, and generates as output a textual UML/OCL model.

The Java-coded approach for defining program abstractors therefore appears to have reduced energy use compared to the CGTL approach (Figure 6). However, substantially more human effort is required to create and maintain a Java-coded transformation, compared to a CGTL transformation (Lano, Haughton, et al. 2024). In the case of large languages such as VB or COBOL, writing a Java abstractor would be a substantial multi-person-year project, whilst CGTL abstractors for these languages were each developed within a few person months. In addition, it is possible to verify properties of a CGTL transformation by inspection of the text-to-text rules. For example, to check that

Java application case	Size (LOC)	Java2UML Java-coded	cgJava2UML CGTL-coded
AVATAR case 113	34	1.56	13.9
nsapp	132	0.89	11.4
Transcoder case 20	168	4.63	25.9
CDOapp	290	2.8	16.6
Averages	156	2.47	16.95

Table 11 Energy use (mWh) for program abstraction approaches

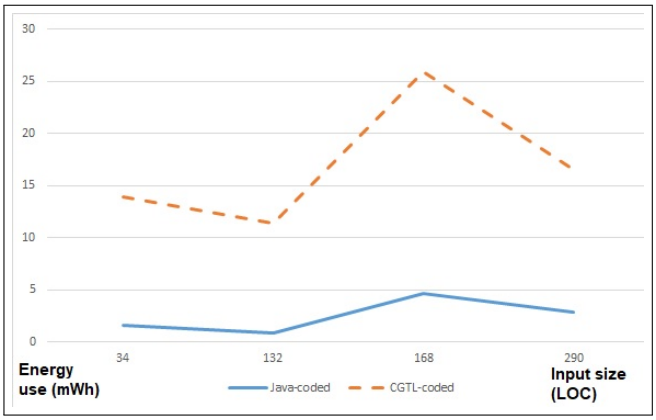

Figure 6 Energy use of code abstractor versions

the cyclomatic complexity of the code is not increased by the abstraction process. Such verification would require significantly higher resources for a Java-coded abstractor.

For code generation, the energy use of the Java-coded UML2Java7 code generator was compared to that of the UML-RSDS-specified uml2py3 code generator and the CGTL-coded cgJava8 generator using four input models of different sizes: mmUML (121 LOC), ocldate (250 LOC), matrixlib (541 LOC) and Excel (747 LOC). Each code generator takes as input a UML/OCL model and produces text files of Java or Python code for the model.

Table 12 shows the detailed evaluation data. Only the execution times of the three transformations are considered, not including model loading times. The results show that in this situation the MT-coded transformation has reduced energy use compared to the Java-coded and CGTL-coded versions (Figure 7). One reason may be the relatively small size (317KB) of the uml2p3 executable, whilst the other generators execute within the 5MB AgileUML IDE.

The development, maintenance and verification of code generators written in CGTL requires less effort than with equivalent UML-RSDS or Java-coded versions (Lano, Xue, & Haughton 2024).

For RQ3 we can therefore conclude that MDRE energy use can be reduced in principle by using Java-coded abstraction and code generation transformations, in preference to CGTL transformations. In addition, MT-coded transformations for code generation can have reduced energy use compared to Java-coded

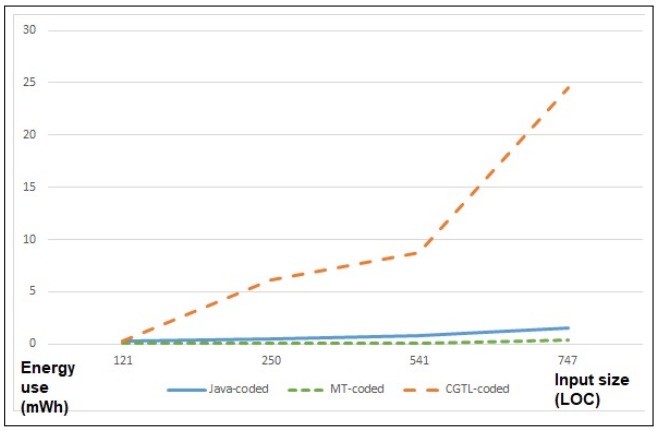

Figure 7 Energy use of code generator versions

Application case	Size (LOC)	UML2Java7 Java-coded	uml2py3 MT-coded	cgJava8 CGTL-coded
mmUML	121	0.238	0.07	0.273
ocldate	250	0.438	0.09	6.1
matrixlib	541	0.797	0.05	8.73
Excel	747	1.56	0.39	24.55
Averages	414.8	0.75	0.15	9.91

Table 12 Energy use (mWh) for code generation approaches.

transformations. However, there are tradeoffs with regard to the development effort and time required for these different transformation techniques: generally a UML-RSDS transformation is smaller and easier to maintain compared to a Java-coded equivalent, and a CGTL transformation is further smaller and requires reduced effort to maintain compared to an equivalent UML-RSDS transformation (Lano, Haughton, et al. 2024). Indeed it was considered infeasible to develop Java-coded abstractors for VB and COBOL because of the size and complexity of these languages.

6. Related work

The only previous research on this topic was (Cordero et al. 2015), which proposes an energy-measurement strategy to identify the most energy-expensive components within a Java legacy system, and hence to prioritise system components for refactoring to reduce energy consumption. In contrast, our approach is focussed on energy efficiency improvement by analysis and refactoring of the abstracted system models. The approach of (Cordero et al. 2015) could be used to select the most critical legacy system components to be optimised by applying our approach.

There has been considerable software sustainability research in the fields of programming languages and program design, whereby the energy use of different software design and implementation options are considered, including design patterns (Bree & O'Cinneide 2022; Maleki et al. 2017; Sahin et al. 2012), refactorings (Sahin et al. 2014), programming language choices (Georgiou et al. 2017; Marini et al. 2025; Pereira et al. 2017) and data structure choices (Michanan et al. 2017; Olivera et al. 2019; Singh et al. 2015). In contrast, only a few research works consider the estimation and analysis of energy use based on software models such as class diagrams or state machines (Alves et al. 2020; Brunschwig & Goaer 2024; Duarte et al. 2019; Lano, Alwakeel, & Rahman 2024b,a).

Addressing software energy use issues at the model level could enable energy use reduction to be achieved across multiple target platforms. The proposal of (Brunschwig & Goaer 2024) leverages this concept to identify energy use hotspots for mobile apps at a platform-independent model level. The work of (Alves et al. 2020; Duarte et al. 2019) defines techniques to estimate software energy consumption based on state machine models at the detailed design level. Some initial ideas on the use of an MDE process to manage sustainability requirements have been identified by (Sousa et al. 2024), and analysis of the general issues involved in using MDE for sustainable microservice architectures is given by (Morais et al. 2024). However, most MDE toolsets do not provide the necessary tools to support energy use analysis and improvement, and MDE specification and design languages such as UML and OCL do not provide any notations to specify or constrain energy use. The energy efficiency of MDE processes and tools themselves also need to be examined, for example, to compare the energy use of different MDE tooling approaches and different model transformation (MT) languages, and to compare MT languages to 3GLs as alternatives for implementing transformations (Lano & Rahimi 2024).

7. Limitations and future work

A significant limitation of the approach presented here is the need for greater automation in supporting refactorings for energy use improvement. Several improvement actions, such as defining separate operations to address redundant result computation flaws, currently need manual intervention.

There is scope for further automation support of refactorings, for example, a technique for removing the use of reflection would be to represent the type structure of the model internally within the implementation, by pre-initialising maps of reflection information about the classes and features of a model. The range of OCL simplifications and program reduction transformations could also be extended, to include simplifications of expressions involving maps (such as replacing $mp \rightarrow keys() \rightarrow includes(k)$ by $mp \rightarrow includesKey(k)$ for map mp) and other OCL extensions supported by AgileUML.

In Section 5 we encountered problems with comparing the execution of COBOL source programs with re-engineered versions in Python. While running GnuCOBOL on a general-purpose computer has made the testing of COBOL possible for this paper, in practice, COBOL is usually run on mainframe systems dedicated to performing repeated and intensive business operations. An option for mainframe emulation is Hercules, an open-source mainframe emulator that can emulate several IBM mainframe operating systems, specifically those of the System/370, ESA/390 and z/Architecture architectures (Bowler et

al. 2025a,b). This will be used in future analysis work involving COBOL.

Here we have used rule-based and deterministic reverse and re-engineering techniques. Instead, Large Language Models (LLMs) could be investigated as an alternative approach for mapping from programs to UML/OCL (Siala & Lano 2025) and for implementing energy use improvement refactorings. Finally, improved visualisations of energy use flaws could be added, including code highlighting, and the range of analyses could be extended to include the detection of energy-expensive design patterns such as Visitor or Decorator, and to restructure the system into more energy-efficient versions without the patterns.

Conclusions

In this paper we have defined techniques for energy use analysis and improvement as part of model-driven re-engineering processes, and we have shown that these can be applied to real-world cases of re-engineering tasks. Energy use flaws can be detected and corrected by automated refactoring actions, resulting in reduced energy use by the re-engineered applications, compared to the legacy source versions. In addition, we have shown that the energy use of the MDRE process can itself can be reduced.

References

Alves, D., Ferreira, O., Duarte, L., Silva, D., & Maia, P. (2020). Experiments on model-based software energy consumption analysis involving sorting algorithms. *Revista de Informatica Teorica e Aplicada*, *27*(3).

Anthony, L., Kanding, B., & Selvan, R. (2020). Carbontracker: Tracking and predicting the carbon footprint of training deep learning models. In *ICML workshop "challenges in deploying and monitoring ML systems"*.

Betz, S., et al. (2015). Sustainability debt: a metaphor to support sustainability design decisions. In *RE4SuSy workshop.*

Bowler, R., Maynard, J., Jaeger, J., et al. (2025a). *The Hercules System/370, ESA/390, and z/Architecture emulator.* (http://www.hercules-390.org/)

Bowler, R., Maynard, J., Jaeger, J., et al. (2025b). *The Hercules System/370, ESA/390, and z/Architecture emulator: SDL Hercules 4.x Hyperion.* (https://sdl-hercules-390.github.io/html/)

Bree, D. C., & O'Cinneide, M. (2022). The energy cost of the Visitor pattern. In *ICSME.*

Brunschwig, L., & Goaer, O. L. (2024). Cross-detection of mobile-specific energy hotspots: MBSE to the rescue. In *SusMOD workshop, MODELS 2024.*

Buttner, F., & Gogolla, M. (2014). On OCL-based imperative languages. *Science of Computer Programming*, *92*, 162–178.

Cabot, J., & Teniente, E. (2007). Transformation techniques for OCL constraints. *Science of Computer Programming*, *68*, 179–195.

Cordero, V., de Guzman, I., & Piattini, M. (2015). A first approach on legacy system consumption measurement. In *International conference on global software engineering workshops* (pp. 35–43). IEEE.

Correa, A., & Werner, C. (2007). Refactoring OCL specifications. *SoSyM*, *6*, 113–138.

Deltombe, G., Goaer, O. L., & Barbier, F. (2012). Bridging KDM and ASTM for model-driven software modernization. In *SEKE 2012.*

Duarte, L., et al. (2019). A model-based framework for the analysis of software energy consumption. In *SBES 2019.*

Fahad, M., Shahid, A., Manumachu, R., & Lastovestsky, A. (2019). A comparative study of methods for measurement of energy of computing. *Energies*, *12*.

Fowler, M., & Beck, K. (2019). *Refactoring: improving the design of existing code, 2nd edition*. Pearson.

Fuhr, A., Horn, T., Riediger, V., & Winter, A. (2013). Model-driven software migration into service-oriented architectures. *Comput. Sci. Res. Dev*, *28*, 65–84.

Gandhi, S., Patwardhan, M., Khatri, J., Vig, L., & Medicherla, R. (2024). Translation of low-resource COBOL to logically-correct and readable Java leveraging high-resource Java refinement. In *Llm4code.*

Georgiou, S., Kechagia, M., & Spinellis, D. (2017). Analyzing programming languages energy consumption: An empirical study. In *Proceedings of PCI 2017.*

Gries, D. (1971). *Compiler construction for digital computers*. Wiley.

He, X., Avgeriou, P., Liang, P., & Li, Z. (2016). Technical debt in MDE: A case study on GMF/EMF-based projects. In *MODELS 2016.*

Kerner, S. (2023). *COBOL language still in demand as application modernization efforts take hold.* (www.itprotoday.com)

Khadka, R., Batlajery, B., Saeidi, A., Jansen, S., & Hage, J. (2014). How do professionals perceive legacy systems and software modernization? In *ICSE 2014.* ACM Press.

Krasteva, I., Stavru, S., & Ilieva, S. (2013). Agile software modernization to the service cloud. In *ICIW 2013* (pp. 1–9).

Kurtz, T. (1978). BASIC, ACM history of programming languages conference, 1978. *SIGPLAN Notices*, *13*(8), 103–118.

Lacoste, A., et al. (2019). Quantifying the carbon emissions of machine learning. *arXiv*(1910.09700v2).

Lannelongue, L., Grealey, J., & Inouye, M. (2021). Green algorithms: Quantifying the carbon footprint of computation. *Advanced Science*, *8*.

Lano, K., Alwakeel, L., & Rahman, Z. (2024a). Design patterns for software sustainability. In *Pattern languages of programs (PLoP).*

Lano, K., Alwakeel, L., & Rahman, Z. (2024b). Software modelling for sustainable software engineering. In *2nd Agile MDE workshop, STAF 2024.*

Lano, K., Haughton, H., Yuan, Z., & Alfraihi, H. (2024). Agile model-driven re-engineering. *Innovations in Systems and Software Engineering*.

Lano, K., & Kolahdouz-Rahimi, S. (2021). Extending OCL with map and function types. In *FSEN 2021.*

Lano, K., Kolahdouz-Rahimi, S., & Jin, K. (2022). OCL libraries for software specification and representation. In *OCL 2022, MODELS 2022 companion proceedings.*

Lano, K., Kolahdouz-Rahimi, S., Yassipour-Tehrani, S., & Shar-

baf, M. (2018, June). A survey of model transformation design patterns in practice. *JSS*, *140*, 48–73.
Lano, K., & Rahimi, S. (2024). Sustainability analysis and improvement of model driven engineering and model transformation languages. In *IKT 2024.*
Lano, K., & Siala, H. (2024a). Using MDE to automate software language translation. *Automated Software Engineering*, *31*.
Lano, K., & Siala, H. (2024b). Using OCL for verified reengineering. In *MoDeVVa 2024, MODELS 2024.*
Lano, K., Xue, Q., & Haughton, H. (2024). A concrete syntax transformation approach for software language processing. *Springer Nature Computer Science*.
Lano, K., Yassipour-Tehrani, S., Alfraihi, H., & Kolahdouz-Rahimi, S. (2017). Translating from UML-RSDS OCL to ANSI C. In *OCL 2017, STAF 2017* (pp. 317–330).
Maleki, S., Fu, C., Banotra, A., & Zong, Z. (2017). Understanding the impact of object-oriented programming and design patterns on energy efficiency. In *IGSC workshop on sustainability in multi/many-core systems.*
Mann, R. (2023). Business risk and user-created solutions. *IT Now*, 38–39.
Marco, A. D., Iancu, V., & Asinofsky, I. (2018). COBOL to Java and newspapers still get delivered. In *Proceedings IEEE international conference on software maintenance and evolution* (pp. 583–586). IEEE Press.
Marinescu, R. (2012). Assessing technical debt by identifying design flaws in software systems. *IBM Journal of Research and Development*, *56*(5).
Marini, N., Pampaloni, L., Martino, F. D., Verdecchia, R., & Vicario, E. (2025). Green AI: Which programming language consumes the most? In *GREENS 2025.*
Michanan, J., Dewri, R., & Rutherford, M. (2017). GreenC5: An adaptive, energy-aware collection for green software development. *Sustainable Computing: Informatics and Systems*, *13*, 42–60.
Microsoft Corp. (2022). *Office VBA Reference.* (https://learn.microsoft.com/en-us/office/vba/api/overview)
Morais, G., Adda, M., & Bork, D. (2024). Breaking down barriers: Building sustainable microservices architectures with model-driven engineering. In *SusMOD workshop, MODELS 2024.*
Naumann, S., Dick, M., Kern, E., & Johann, T. (2011). The GREENSOFT model: a reference model for green and sustainable software and its engineering. *Sustainable Computing: Informatics and Systems*, *1*, 294–304.
Noureddine, A. (2022). PowerJoular and JoularJX: Multiplatform software power monitoring tools. In *18th international conference on intelligent environments.* IEEE.
Object Management Group. (2014). *Object Constraint Language (OCL) 2.4 specification.*
Olivera, W., et al. (2019). Recommending energy-efficient Java collections. In *IEEE/ACM MSR.*
Pathania, P., et al. (2023). Towards a knowledge base of common sustainability weaknesses in green software development. In *ASE '23.* IEEE.
Pereira, R., Couto, M., Ribeiro, F., Rua, R., Cunha, J., Fernandes, J. P., & Saraiva, J. (2017). Energy efficiency across programming languages: How do energy, time and memory relate? In *SLE '17* (pp. 256–267). ACM.
Perez-Castillo, R., de Guzman, I. G.-R., & Piattini, M. (2011). Knowledge discovery metamodel ISO/IEC 19506: A standard to modernize legacy systems. *Computer Standards and Interfaces*, *33*, 519–532.
Rahimi, S., Lano, K., Sharbaf, M., Karimi, M., & Alfraihi, H. (2020). A comparison of quality flaws and technical debt in model transformation specifications. *JSS*, *169*.
Sahin, C., Cayel, F., Gutierrez, I., Clause, J., Kiamilev, F., Pollock, L., & Winbladh, K. (2012). Initial explorations on design pattern energy usage. In *International workshop on green and sustainable software* (p. 55-61).
Sahin, C., Pollock, L., & Clause, J. (2014). How do code refactorings affect energy usage? In *ESEM '14.* ACM.
Sammet, J. (1978). The early history of COBOL, ACM history of programming languages conference, 1978. *SIGPLAN Notices*, *13*(8), 121–161.
Siala, H., & Lano, K. (2025). Towards using LLMs in the reverse engineering of software systems to OCL. In *SANER 2025.*
Siala, H., Lano, K., & Alfraihi, H. (2024). Model-driven approaches for reverse engineering – a systematic literature review. *IEEE Access*.
Singh, J., Naik, K., & Mahinthan, V. (2015). The impact of developer choices on energy consumption of software on servers. *Procedia Computer Science*, *62*, 385–394.
Sneed, H. (2011). Migrating from COBOL to Java: A report from the field. In *IEEE Proc. of 26th ICSM* (pp. 1–7). IEEE Press.
Sneed, H., & Jandrasics, G. (1987). Inverse transformation of software from code to specification. In *IEEE conf. soft. maintenance.*
Sousa, T., Ries, B., & Guelfi, N. (2024). Model-driven software product line engineering of AI-based applications for achieving the sustainable development goals: Vision paper. In *SusMOD workshop, MODELS 2024.*
Sun, C., Li, Y., Zhang, Q., Gu, T., & Su, Z. (2018). Perses: Syntax-guided program reduction. In *ICSE 2018.* ACM.
Wimmer, M., Martinez, S., Jouault, F., & Cabot, J. (2012). A catalogue of refactorings for model-to-model transformations. *Journal of Object Technology*, *11*(2), 1–40.

About the authors

Kevin Lano is Reader in Software Engineering at King's College London. He is the main developer of the AgileUML toolset, and Director of AgileMDE Ltd. You can contact the author at kevin.lano@kcl.ac.uk or visit https://agilemde.co.uk.

Shekoufeh Rahimi is a Senior Lecturer at the University of Roehampton, London. You can contact the author at Shekoufeh.Rahimi@roehampton.ac.uk.

Zishan Rahman is a PhD student at King's College London. You can contact the author at zishan.rahman@kcl.ac.uk.