Energy Use Analysis and Improvement of Mamba Specifications

Zishan Rahman[1], Kevin Lano[1]

[1]King's College London, London, UK

Abstract

Software quality is a key factor in the ongoing cost of maintaining software systems. The concept of 'technical debt' is used to express the relationship between the flaws present in a software system and the consequent costs for evolution or correction of the flawed system. Recently there has also been concern with the presence of energy-use flaws within software systems, which leads to ongoing waste of resources by the systems, and which could be termed the 'sustainability debt' of a software system.

The analysis of energy use at the software modelling level in model-driven engineering (MDE) is a new research field, and there has been no prior work specifically on specification model energy analysis for low code development platforms (LCDPs). In this paper we apply software quality and energy use analysis to specifications in the Mamba language of the zAppDev low code MDE toolset. We provide a catalogue of common energy use flaws in the language, together with corresponding refactorings to improve energy-efficiency. We validate that these do reduce computational load for both client-side and server-side code generated by the zAppDev platform.

Keywords

Software quality analysis, Software energy use, Mamba, zAppDev, Agile MDE

1. Introduction

Information and Computing Technology (ICT) is a significant and increasing contributor to global energy use and consequently to global greenhouse gas (GHG) emissions. ICT is currently responsible for 3.9% of global GHG, and is predicted to reach 14% by 2040 [1]. Data centers have a similar carbon impact to the aviation industry [2], and global data center electricity demand is projected to reach 945 TWh per annum by 2030 [3]. Machine learning (ML) and AI systems are a particular source of ICT GHG emissions which is rapidly increasing due to the pervasive use and increasing scale of ML systems. It has been estimated that AI systems contributed up to 80 million tonnes of GHG emissions in 2025 [4].

Software quality flaws and embedded *technical debt* in software has been considered as one consequence of agile development, due to the agile emphasis on rapid software production and quick delivery of working code [5]. However the same factors can also lead to agile development delivering software with *energy-use flaws* and embedded *sustainability debt* [6], representing the burden of excessive energy use caused by inefficient code. A difference between sustainability debt and technical debt is that the ongoing costs of sustainability debt are not increased maintenance costs, but instead: (i) poor service performance and hence possible loss of business; (ii) higher energy costs (if locally executed) or higher cloud provision costs (if executed on the cloud); (iii) ongoing environmental impacts and costs.

As with technical debt, one remedy for the tendency of agile development to introduce sustainability debt is to perform *refactoring* to identify and remove this debt. In previous work we defined a catalogue of UML/OCL quality and energy-use flaws for models in the AgileUML Agile MDE platform, together with corresponding refactorings to remove or reduce these flaws [7, 8, 9]. In this paper we investigate how this catalogue can be adapted to address flaws in Mamba specifications for the zAppDev low code development platform (LCDP) [10], and we also consider specialised energy-use flaws and issues which arise with this platform. Section 2 describes Mamba and the zAppDev platform, and its relation to UML/OCL. Section 3 describes how energy and quality flaws are detected using our analysis tools, and

Joint Proceedings of the STAF 2026 Workshops: AgileMDE, GCM, ICMM, LLM4SE, TTC. Rennes, France, June 29-July 3, 2026

zishan.rahman@kcl.ac.uk (Z. Rahman); kevin.lano@kcl.ac.uk (K. Lano)

0000-0002-9706-1410 (K. Lano)

© 2026 Copyright (C) for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

CEUR Workshop Proceedings ceur-ws.org ISSN 1613-0073

published 2026-11-26

CEUR-WS.org/Vol-4338/AgileMDE-paper5.pdf

Section 4 describes the specialisation of energy and quality flaw analysis techniques for Mamba, and identifies appropriate refactorings for improvement of Mamba specifications. Section 5 gives empirical measurement of the effect of selected refactorings. Section 6 describes related work. Section 7 gives conclusions and suggestions for future work.

2. Mamba and zAppDev

The zAppDev LCDP toolset[1] has been used in a range of industrial applications across Europe, in the transport, utilities, energy and other sectors. It adopts a graphical 'low code' approach, whereby developers use the zAppDev cloud-based toolset to visually specify the business objects of their system, together with operations expressed in a high-level specification language, Mamba, and a UI designed via a 'drag-and-drop' style process. The toolset can then produce code and other artifacts to build complete executable implementations of the system for different platforms. Mamba specifications are typically transformed into JavaScript or C# code for execution.

A three tier architecture underlies zAppDev systems, with a GUI tier running JavaScript on client browsers, communicating with a .NET back end running C# code on a cloud server. This server layer interacts with a persistence/resource layer via a predefined API. Mamba specification is used at the GUI and server layer, but the persistence layer is not explicitly defined by users (Figure 1).

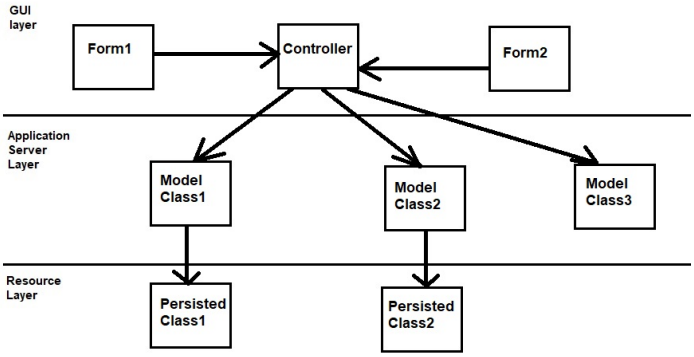


Figure 1: Architecture of zAppDev systems

Mamba is conceptually similar to the Object Constraint Language (OCL) of UML [11], supplemented with procedural statement constructs for introducing local variables, assignment, sequencing, conditional behaviour and looping. Its syntax is however based on C# rather than the semi-mathematical notation of OCL. Tables 1 and 2 detail the semantics of Mamba types and expressions, and list the corresponding OCL types and expressions. Table 3 summarises the statement constructs of Mamba. Whilst OCL provides four different kinds of collection: sets, sequences, bags and ordered sets, Mamba only provides sequences and arrays (fixed-size sequences). Mamba collection indexing is 0-based rather than 1-based as in OCL. Mamba has a dictionary type, corresponding to the map type commonly adopted in OCL versions, and function types, which are also supported in OCL versions such as AgileUML OCL and Eclipse OCL [12, 13]. Similar collection operators are provided in Mamba and OCL (Table 2), such as $\rightarrow select$ (Where in Mamba) and $\rightarrow collect$ (Select in Mamba). However, Mamba also provides more general forms of the operators *first*, *last*, *max*, *min* and *count*, which take a function or predicate parameter. These enable a single iteration to be used in cases where OCL would need to use double iterations (e.g., $col.Max(x \Rightarrow F)$ in Mamba instead of $col \rightarrow collect(x \mid F) \rightarrow max()$ in OCL). The statement forms of Mamba (Table 3) are similar to those of AgileUML procedural statements [9].

[1] https://www.zappdev.io

Table 1
Mamba types and corresponding OCL types

Mamba	Definition	OCL
string, char	String and character types	String
void	void type	OclVoid
bool	boolean type	Boolean
byte, int, long	integer types	Integer
double, float, decimal	real-number types	Real
Object	object type	OclAny
CollectionBase, Array, Collection	sequential collection types	Sequence
Func, Predicate	function types	Function
IndexedCollection	map with computed index	Map + Function
Dictionary	map type	Map

Table 2
Mamba expressions and corresponding OCL expressions

Mamba	Definition	OCL
C.GetAll()	all instances of class C	$C.allInstances()$
C.GetByKey(k)	the C object with key value k	$C[k]$
$\{x, y, z\}$	literal collection	$Sequence\{x, y, z\}$
col.Length	size of string/collection	$col \rightarrow size()$
col.Contains(x)	membership test	$col \rightarrow includes(x)$
col.Skip(n)	subsequence of elements from $n+1$ onwards	$col.subrange(n+1)$
col.Take(n)	subsequence of first n elements	$col.subrange(1, n)$
col.Except(s)	collection subtraction	$col - s$
col.Intersect(s)	collection intersection	$col \rightarrow intersection(s)$
col.Concat(s)	sequence concatenation	$col \rightarrow concatenate(s)$
col.First(x => P)	first col element that satisfies P	$col \rightarrow any(x \mid P)$
col.Last(x => P)	last col element that satisfies P	$col \rightarrow reverse() \rightarrow any(x \mid P)$
col.Select(x => expr)	collection of *expr* values over *col*	$col \rightarrow collect(x \mid expr)$
col.Min(x => expr)	minimum value of *expr* over *col*	$col \rightarrow collect(x \mid expr) \rightarrow min()$
col.Max(x => expr)	maximum value of *expr* over *col*	$col \rightarrow collect(x \mid expr) \rightarrow max()$
col.OrderBy(x => expr)	sort *col* by *expr* values	$col \rightarrow sortedBy(x \mid expr)$
col.Count(x => P)	count of x satisfying P	$col \rightarrow select(x \mid P) \rightarrow size()$
col.Any(x => P)	some element of *col* satisfies P	$col \rightarrow exists(x \mid P)$
col.All(x => P)	all elements of *col* satisfy P	$col \rightarrow forAll(x \mid P)$

3. Quality and energy use flaws

Software quality flaws are often identified by the presence of technical debt indicators, such as those listed in Table 4. By analogy with quality flaws and technical debt, we also define a set of *sustainability debt indicators* to detect energy-use flaws (Table 5). Expression/code size thresholds are specified using the expression/code size measure $c(s) = number\ of\ operators + number\ of\ identifiers$. Both quality and energy-use flaws should be considered together when analysing and refactoring code: changes made to improve energy-efficiency should not impact negatively on code quality, and vice-versa. For example, removing duplicated code can be achieved by factoring out the duplicates into a new operation, by the 'Extract method' refactoring. However this refactoring can increase energy use [14], and it is therefore preferable to remove DC flaws by reorganisation of the code without creating new operations.

Table 3
Mamba statements

Mamba	Description
x = v;	assignment of v to x
T x = v;	declare local variable x of type T, initialised to v
S1; S2;	sequencing of statements
if expr { S1; } else { S2; }	conditional statement
switch expr { case v1: S1; ... case vn: Sn; }	switch statement
foreach x in col { S; }	for loop over collection
for (T x = v; cond; incr) { S; }	integer range for loop
while expr { S; }	while loop
throw expr;	raise exception
try { S; }	try statement
catch E ex { S; }	catch exception clause of try statement
finally { S; }	finally clause of try statement

Table 4
AgileUML technical debt indicators

Indicator	Definition
DC	Number of duplicated statements/expressions *s*.
EFO	Number of operation calls in one operation.
EFI	Number of calls of one operation.
CBR	Number of operations with self-dependencies either directly or indirectly in call graph.
UVA	Number of unused parameters/local variables of operation.
CC	Number of basic control flow conditions in an operation + 1.
MGN	Number of literal constant values appearing in code.
EPL	Number of operations with ≥ 7 parameters.
EOS	Number of operations with $c(op) > 100$.
EDN	Nesting depth of statements.

Table 5
AgileUML sustainability debt indicators

Indicator	Definition
LCE	Number of iterator/loop test/body expressions *expr* independent of iterator/loop variables.
DEV	Number of cloned expressions in operation, with unchanged value across clones.
RL	Resource leaks: Number of files/datasources/sockets etc opened and not closed.
LRC	Number of chained references in expression.
UOR	Number of expressions of form $op(pars) \rightarrow first()$ or $op(pars) \rightarrow at(i)$ for constant *i*.
OES	Number of inefficient OCL expressions/statements.
MEL	Maximum expression size $c(expr)$.
RC	Number of statements that can be removed/replaced.
MNC	Type complexity: number of nested constructors in a type.

4. Energy and quality flaw analysis in Mamba

The similarities between Mamba and OCL described in Section 2 above enable us to interpret the OCL flaw cases and refactorings of [15] as corresponding potential flaws and refactorings in Mamba. There are also additional flaws which we have observed in large zAppDev projects, where excessively complex expression or code structures are used instead of simpler and potentially more efficient alternatives. Mamba also supports more effective refactorings than OCL, because of the extended forms of the *Max*, *Min*, *First* and *Last* operators provided by the language.

To analyse a zAppDev development, we parse the business object files $f.bo$ of the development, using a custom Mamba parser constructed using ANTLR. A CSTL transformation [16] is then used to map the Mamba code to UML/OCL for analysis. The Mamba import option is the "Load metamodel/Load zAppDev' option on the AgileUML *File* menu. Quality and energy analysis options are listed on the *Analyse* menu.

4.1. Analysis of expression flaws

Table 6 lists some of the Mamba expression energy flaws and refactorings which correspond to inefficient OCL expression (OES flaw) cases identified in [15, 8]. The numbering of flaw cases is based on the listing of OCL flaws in [9], for instance, the corresponding OCL flaw and refactoring of case 1 concerns the rewriting of an inefficient expression $col{\rightarrow}select(x \mid P){\rightarrow}select(x \mid Q)$ to $col{\rightarrow}select(x \mid P \,\&\, Q)$.

An additional refactoring not directly available in OCL is from the statement

```
v = col.OrderBy(x => expr).First();
```

to

```
var minVal = col.Min(x => expr);
v = col.First(x => expr == minVal);
```

This replaces the $O(n \log n)$ sort by two $O(n)$ iterations. Likewise *OrderBy*.*Last* can be replaced by a combination of *Max* and *Last*.

Table 6
Energy-use flaws in Mamba expressions

#	Original Mamba	Flaw	Refactored Mamba
1	Col.Where(x => P).Where(x => Q)	Multiple iterations: combine	Col.Where(x => P && Q)
3	Col.Where(x => P).First(x => Q)	Multiple iterations: combine	Col.First(x => P && Q)
5	Col.Where(x => P).Last(x => Q)	Multiple iterations: combine	Col.Last(x => P && Q)
17	Col.Where(x => P).Any(x => Q)	Multiple iterations: combine	Col.Any(x => P && Q)
30	Col.Where(x => P).Single(x => Q)	Multiple iterations: combine	Col.Single(x => P && Q)
7	Col.Where(x => P).Length > 0 Col.Count(x => P) > 0	Complete iterations of Col: replace by early return	Col.Any(x => P)
18	Col.Select(x => e).Contains(y)	Complete iteration of Col: replace by early return	Col.Any(x => e == y)
8 9	Col.Select(x => expr).Skip(n) Col.Select(x => expr).Take(n)	Redundant computation: only part of result is used	Col.Skip(n).Select(x => expr) Col.Take(n).Select(x => expr)
20	Col.OrderBy(x => x).Get(0) Col.OrderBy(x => x).First()	Redundant computation: sorting entire list	Col.Min(x => x)

The same approach given in [9] to justify energy efficiency optimisation refactorings of OCL can also be used here for Mamba. Atomic computation actions a such as accessing the contents of a memory address are assigned a notional *computational energy cost* $cost_s(a) \in \mathbb{R}^+$, measured in Joules or mWh, and possibly dependent upon the current state s [9]. The computational energy cost $cost_s(expr)$ of evaluating expression *expr* in state s will then be a sum of the costs of the basic actions of which its evaluation consists.

Because the short-circuit and operator && has the definition

$$P \;\&\&\; Q \;=\; \mathit{if}\ P\ \mathit{then}\ Q\ \mathit{else}\ \mathit{false}\ \mathit{endif}$$

in OCL terms, we also have:

$$\mathit{cost}_s(P \;\&\&\; Q) \;=\; \begin{cases} \mathit{cost}_s(P) + \mathit{cost}_s(Q) & \mathit{when}\ P\ \mathit{holds\ in}\ s \\ \mathit{cost}_s(P) & \mathit{when}\ P\ \mathit{does\ not\ hold\ in}\ s \end{cases}$$

Thus in the first case of Table 6, we have that the total cost for the original version Col.Where(x => P).Where(x => Q) is the sum of (i) the cost of evaluating $x.P$ for each element of *Col*, (ii) the cost δ, say, of creating a new empty collection *Col*′, and (iii) the cost $p * \alpha$ of adding p elements to this intermediate collection, where p is the number of *Col* elements with P true; (iv) the cost of evaluating $x.Q$ for the elements in *Col*′, (v) the cost $\delta + q * \alpha$ of creating the final result collection and adding q elements to it: the number of *Col* elements that satisfy both P and Q. Therefore the total cost is:

$$\Sigma_{x \in Col} \mathit{cost}_s(x.P) \;+\; (\delta + \alpha * p) \;+\; \Sigma_{x \in Col'} \mathit{cost}_s(x.Q) \;+\; (\delta + \alpha * q)$$

But for the refactored version, the intermediate collection *Col*′ is not created, so the costs are:

$$\Sigma_{x \in Col} \mathit{cost}_s(x.P \;\&\&\; x.Q) \;+\; (\delta + \alpha * q)$$

The $\mathit{cost}_s(x.P \;\&\&\; x.Q)$ is simply $\mathit{cost}_s(x.P)$ if P is false for x in s. Otherwise, for the p elements with P true, i.e., the elements of *Col*′, the cost is $\mathit{cost}_s(x.P) + \mathit{cost}_s(x.Q)$. In either case, these costs are included in the summation terms of the original expression version. Thus the costs of the refactored version are reduced by at least $\delta + \alpha * p$ compared to the original version. Similar reasoning can be applied for the other expression flaw cases in Table 6.

A common idiom in Mamba code is the combination `Domain.Entity.Find(d => P).First()` of a database search for all elements of *Entity* satisfying a property P, and the first-element operator *First*(). Interpreted literally, this idiom would be a source of inefficiency, however the zAppDev compilers automatically implement this combination as a 'get first element satisfying P' operation. Any additional operator between the *Find* and *First* would prevent the optimisation. In future it may be preferable to introduce an explicit *FindFirst* operator into the language in order to make the optimised semantics clear to specifiers.

4.2. Analysis of statement flaws

Quality and energy-use flaws in Mamba statements, such as duplicated code (DC) or redundant statements (RC), are also observed in real-world Mamba developments. For example, one observed situation is the unnecessary duplication of condition tests, in statements of the form:

```
if e { C1; }
if !e { C2; }
```

where `C1` does not modify the variables of `e`. This is a duplicated expression evaluation (DEV) flaw, and the code can be replaced by

```
if e { C1; } else { C2; }
```

A similar code flaw has the structure:

```
if e { C1; } else { C2; }
if e { C3; } else { C4; }
```

where `C1`, `C2` do not write the variables of `e`. Again, this is a DEV flaw with duplicated condition `e`, and can be refactored to:

```
if e { C1; C3; }
else { C2; C4; }
```

Code refactorings can be justified using computational cost analysis, as with expression refactorings in Section 4.1 above.

For example, in the case of code

```
if e { C1; }
if !e { C2; }
```

the computational cost starting in state s is either $cost_s(e) + cost_s(C1) + cost_{s1}(!e)$ if e is true in s, and $s1$ is the state resulting from execution of $C1$, or $cost_s(e) + cost_s(!e) + cost_s(C2)$ if e is false in s.

But for the refactored code

```
if e { C1; } else { C2; }
```

the respective costs are: $cost_s(e) + cost_s(C1)$ and $cost_s(e) + cost_s(C2)$, so that the costs are no higher and may be lower for the improved version.

5. Evaluation

In this section we evaluate the effect of the proposed refactorings of Section 4 on the runtime energy use of zAppDev systems. We also compare the energy use of two alternative techniques to define sets in Mamba.

To estimate energy use, we apply the Green Algorithms tool for energy measurement [17]. We consider both client-side execution (JavaScript implementation) and server-side execution (C# implementation) of the original and refactored versions of a Mamba specification to provide information on the relative computational loads resulting from these different versions. The client-side execution uses a JavaScript implementation of the Mamba specification, executed by a Microsoft Edge browser on a laptop with a 4 core i5-9400 processor, UK location, 8GB available memory. The server-side execution uses C# Version 3.1 on the same hardware. Additionally we used Linux Mint 21.3 on a i7-1365U processor laptop with 10 cores, UK location, 16GB available memory, for server-side evaluation. The average of three energy-use measurements is taken for each data point for each case, with the device in low-power mode and with only background processes running. All evaluation data and artefacts are at: zenodo.org/records/20122436.

5.1. Energy efficiency refactorings in Mamba

Figures 2, 3, 4 and 5 show the energy efficiency gains achieved by the refactorings 1, 3, 7, 9, 17 and 18 from Table 6 across a range of collection sizes. Figure 6 shows the energy efficiency gains of refactoring 20 and the extra refactoring of OrderBy.First to Min.First.

Table 7 shows the total and average energy efficiency gains of the evaluated cases. Collection sizes 1000 to 5000 are used for client-side evaluation, and sizes 10000 to 50000 for server-side evaluation. Similar results were obtained for Linux server-side execution. The average is the average of the percentage energy use reductions $100 * \frac{original - refactored}{original}$ for the different collection sizes. The total reduction for a case is the percentage reduction of the sum (over different sizes) of energy use of the refactored version relative to the original version. Although there is significant variability in the energy measures for client-side execution, there is a general trend in the results which demonstrates that the proposed refactorings are effective for both client-side and server-side energy use reduction in Mamba.

5.2. Energy efficiency of Mamba sets

Mamba does not provide a built-in set type (i.e., a type of unordered unique-element collections). Instead, sets are defined using either (i) sequences which have unique elements (Collection type in Mamba), or

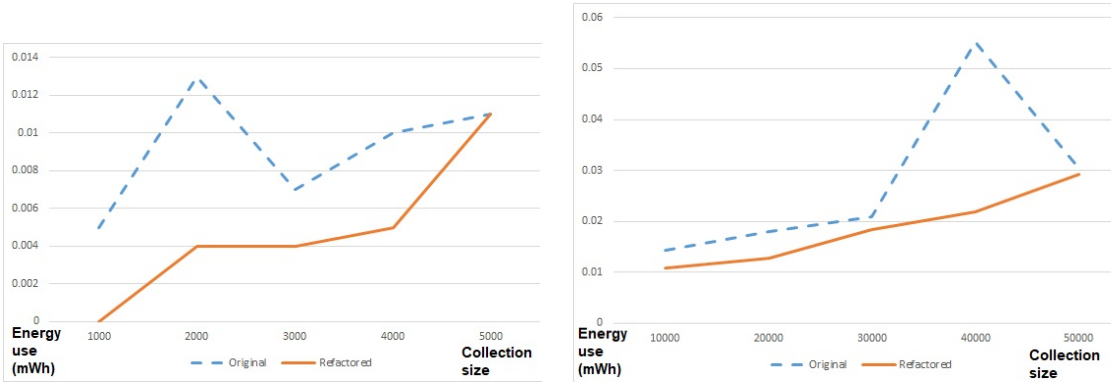


Figure 2: Expression refactoring case 1, client-side and server-side

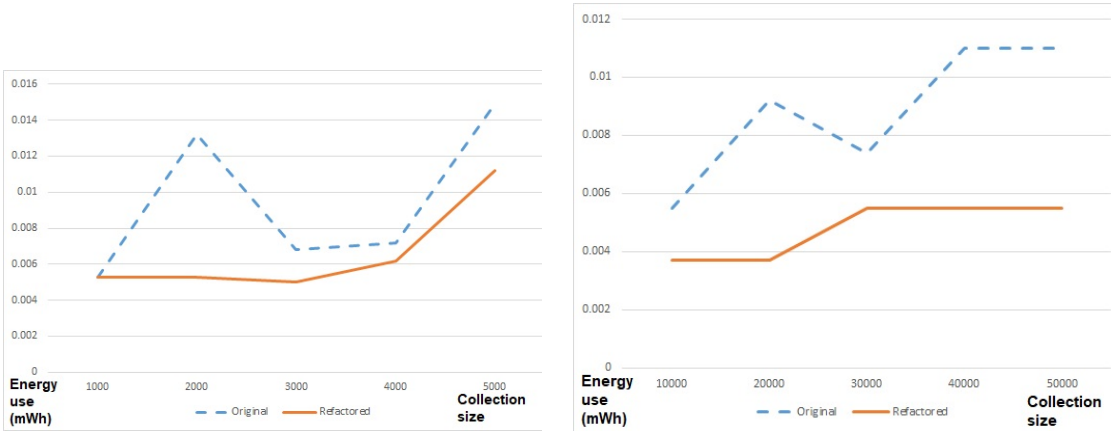


Figure 3: Expression refactoring case 3, client-side and server-side

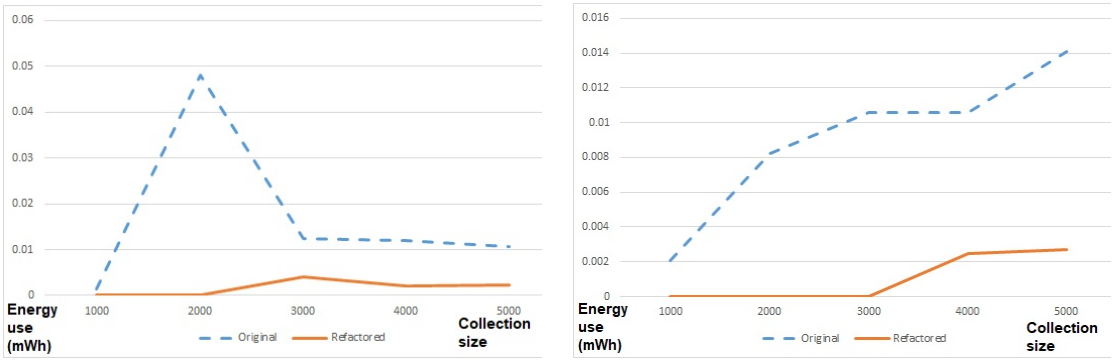


Figure 4: Expression refactoring cases 7 and 9 (client-side)

(ii) the key sets of maps (Dictionary type in Mamba). To analyse which of these alternatives is more energy efficient, we evaluated the energy used by sets of sizes N = 1000, 2000, 3000, 4000 and 5000, implemented with each technique, and with N element additions and N element membership checks in each case. Figure 7 shows the respective energy use of the two techniques for these situations. In general the Dictionary approach to representing sets is substantially more energy-efficient, and exhibits an almost constant energy use for sets of different sizes, whilst the Collection technique exhibits linear or worse growth in energy use. Overall there is a 97.5% reduction of energy use (client side) and 97% reduction (Windows server side) and 89.5% (Linux server side) for the Dictionary version compared to the Collection version. However, there are fewer applicable operators for dictionaries compared to collections in Mamba, in particular there are no union, subtraction or intersection operators on

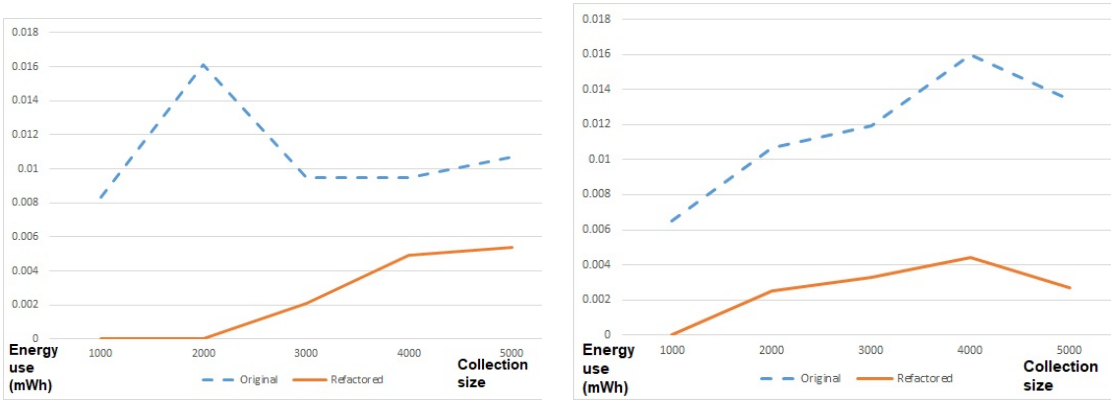


Figure 5: Expression refactoring cases 17 and 18 (client-side)

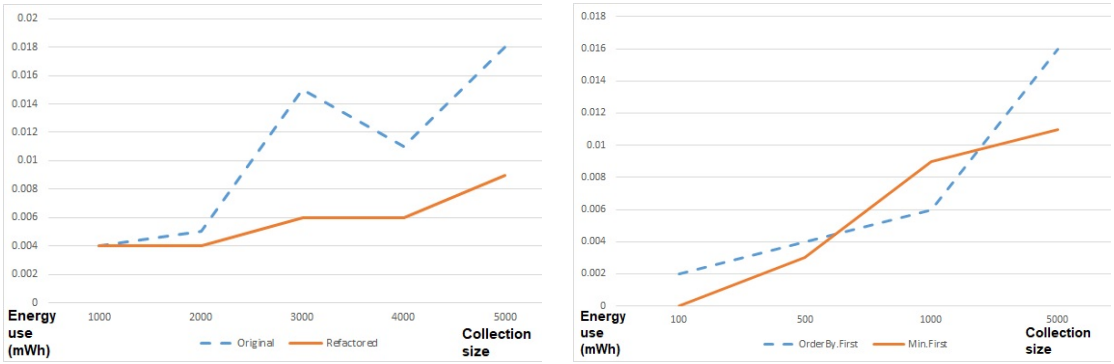


Figure 6: Expression refactoring case 20 and OrderBy.First refactored to Min.First (client-side)

Table 7
Energy efficiency gain of Mamba refactorings

Case	Client-side energy reduction			Server-side energy reduction		
	Total	Average	Std. deviation	Total	Average	Std. deviation
1	47.8%	52.4%	32.8	33%	26%	19.3
3	30.3%	25%	19.8	45.8%	43.6%	12.5
7	89.5%	85.4%	12.9	79.2%	74.9%	12.7
9	88.6%	91.5%	10.6	87.9%	82.1%	8.7
17	77.1%	75.2%	22.9	55.6%	52.4%	14.0
18	77.9%	80.2%	10.3	82.2%	78.7%	8.4
20	45.3%	35.1%	21.9	96.9%	97.3%	2.6
OrderBy.First to Min.First	16.1%	24.6%	56	91.8%	90.6%	4.2

dictionaries, so the collection definition of sets must be used if such operators are required.

6. Related work

There has been considerable work on the energy-efficiency analysis and improvement of software systems at the code level [18, 19, 20], but model-level analysis and improvement is a new field of research, with few published papers [21, 22, 23, 24, 25]. No previous work seems to have been carried out on the energy-use analysis of system specifications in low-code development platforms (LCDPs).

LCDPs such as zAppDev have grown in popularity since the term was first used in 2014 [26]. LCDPs

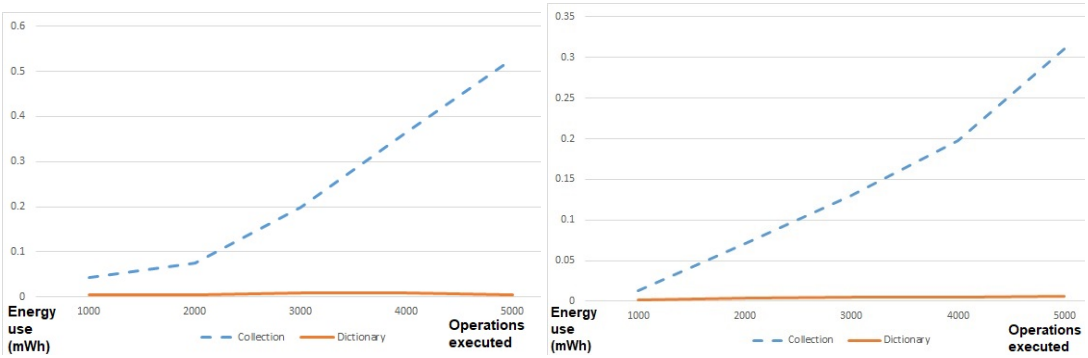


Figure 7: Energy use of Set definitions in Mamba, client-side and server-side

are characterised by providing UI design and construction facilities without coding, and often seamless data management facilities without explicit coding. However, specification of business functionality often needs some form of coding (graphical or textual) to specify business rules and workflows. In zAppDev, the UI construction and database management are mainly carried out without the need for user coding, however the business tier is defined using classes and operations, specified using Mamba. Thus the overall coding load is reduced compared to manual full-stack development of Web applications. Syntactic validation and some semantic checks are carried out on Mamba specifications by zAppDev, but software quality, performance or energy-use analyses are not provided.

LCDPs are reported to reduce development time by up to 90%[2]. More recent surveys confirm that substantial productivity gains can result from the use of LCDPs [27, 28], however there has been a lack of research either on the quality of LCDP-generated systems, or on their energy efficiency. It has been hypothesised that LCDP should reduce quality problems because of the (presumed) high quality of the predefined components of the platforms [28], however this assertion needs to be validated for the particular LCDP being used [29]. We observed that zAppDev system specifications had significant numbers of occurrences of technical and sustainability debt, resulting either from time pressures during development, or from the inexperience of developers with the specialised Mamba language. Therefore there seems to be a need for extension of zAppDev with tools for quality flaw and energy-use flaw analysis, to reduce system technical debt and sustainability debt prior to deployment.

7. Conclusion

We have shown that the energy use analysis and refactorings can be applied for zAppDev/Mamba. We have also identified new improvement refactorings in Mamba, which could be applicable to OCL if extensions of OCL operators such as $\rightarrow max$, $\rightarrow min$, $\rightarrow first$, $\rightarrow last$ were available.

Further work will examine other cases of Mamba expression and statement flaws, and in particular will consider issues related to the Web app implementation approach of zAppDev, based on the MVC pattern. Potential optimisations of MVC for energy-efficiency [24] will be analysed as possible alternatives to this implementation choice. Analysis of database interaction and optimisation will also be carried out. Ultimately we aim to use this research to provide an advisory tool which can be incorporated into zAppDev to assist developers to improve the quality and energy efficiency of their systems at the specification stage.

Declaration on Generative AI

The author(s) have not employed any Generative AI tools.

[2]Gartner report “Magic quadrant for enterprise low-code application platforms", Gartner research, 2016

References

[1] BJSS, The current state of the industry on green software development, 2024. Https://www.bjss.com/articles.
[2] E. Zie, How to engineer more sustainable software, IT Now (2023) 14–15.
[3] International Energy Agency (IEA), Electricity 2025: analysis and forecast to 2030, 2025.
[4] B. de Vries-Gao, The growing energy footprint of AI, Joule (2025).
[5] 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.
[6] S. Betz, C. Becker, R. Chitchyan, L. Duboc, S. Easterbrook, B. Penzenstadler, N. Seyff, C. Venters, Sustainability debt: a metaphor to support sustainability design decisions, in: RE4SuSy Workshop, 2015.
[7] K. Lano, H. Alfraihi, H. Haughton, Research directions for agile model-driven engineering, in: AgileMDE 2024 Workshop, STAF 2024, 2024.
[8] K. Lano, S. K. Rahimi, Z. Rahman, Using MDE to support sustainable software re-engineering, in: ECMFA, 2025.
[9] K. Lano, Z. Rahman, L. Alwakeel, Model-based software sustainability analysis and improvement, Software and Systems Modeling (2026).
[10] I. Ibrahimi, D. Moudilos, Model slicing on low-code platforms, in: FVPM, 2022.
[11] Object Management Group, Object Constraint Language (OCL) 2.4 specification, 2014.
[12] K. Lano, S. Kolahdouz-Rahimi, Extending OCL with map and function types, in: FSEN 2021, 2021.
[13] Eclipse, 2022. Eclipse OCL Version 6.4.0, https://projects.eclipse.org/projects/ modeling.mdt.ocl.
[14] C. Sahin, L. Pollock, J. Clause, How do code refactorings affect energy usage?, in: ESEM '14, ACM, 2014.
[15] K. Lano, S. Rahimi, Z. Rahman, Specification and design refactorings for sustainable agile model-driven engineering, in: AgileMDE 2025, STAF 2025, 2025.
[16] K. Lano, Q. Xue, Agile specification of code generators for model-driven engineering, in: 2020 15th International Conference on Software Engineering Advances (ICSEA), 2020, pp. 9–15.
[17] L. Lannelongue, J. Grealey, M. Inouye, Green algorithms: Quantifying the carbon footprint of computation, Advanced Science 8 (2021).
[18] J. Michanan, R. Dewri, M. Rutherford, GreenC5: An adaptive, energy-aware collection for green software development, Sustainable Computing: Informatics and Systems 13 (2017) 42–60.
[19] W. Olivera, et al., Recommending energy-efficient Java collections, in: IEEE/ACM MSR, 2019.
[20] J. Singh, K. Naik, V. Mahinthan, The impact of developer choices on energy consumption of software on servers, Procedia Computer Science 62 (2015) 385–394.
[21] D. Alves, O. Ferreira, L. Duarte, D. Silva, P. Maia, Experiments on model-based software energy consumption analysis involving sorting algorithms, Revista de Informatica Teorica e Aplicada 27 (2020).
[22] L. Brunschwig, O. L. Goaer, Cross-detection of mobile-specific energy hotspots: MBSE to the rescue, in: SusMOD Workshop, MODELS 2024, 2024.
[23] L. Duarte, et al., A model-based framework for the analysis of software energy consumption, in: SBES 2019, 2019.
[24] K. Lano, L. Alwakeel, Z. Rahman, Design patterns for software sustainability, in: Pattern Languages of Programs (PLoP), 2024.
[25] K. Lano, L. Alwakeel, Z. Rahman, Software modelling for sustainable software engineering, in: 2nd Agile MDE Workshop, STAF 2024, 2024.
[26] D. D. Ruscio, D. Kolovos, J. de Lara, A. Pierantonio, M. Tisi, M. Wimmer, Low-code development and MDE: two sides of the same coin?, Sosym 21 (2022) 437–446.
[27] I. Alsaadi, A. Alsadi, A. Ahmad, A comparative study of low-code development platforms, International Journal of Advanced Computer Science and Applications 5 (2021) 256–263.
[28] K. Rokis, M. Kirikova, Exploring low-code development: a comprehensive literature review, Complex systems informatics Systems and modeling quarterly 36 (2023) 68–86.
[29] D. Liu, H. Jiang, S. Guo, Y. Chen, L. Qiao, What's wrong with low-code development platforms? an empirical study of low-code development platform bugs, IEEE Transactions on Reliability 73 (2024) 695–709.