Software modelling for sustainable software engineering

Kevin Lano[1], Lyan Alwakeel[1] and Zishan Rahman[1]

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

Abstract

Sustainable Software Engineering (SSE) is concerned with the development of *sustainable software*, software which minimises negative impacts on the environment and maximises positive impacts. In this paper we examine how model-driven engineering (MDE) can contribute to software sustainability, by assisting developers to identify energy use flaws at the software modelling level, and by providing guidance and transformations to achieve more sustainable designs.

Keywords

Software Sustainability, Model-driven Engineering, Sustainability Debt, Agile Development

1. Introduction

According to some estimates, information technology is one of the largest single global producers of greenhouse gas (GHG) emissions, and is predicted to consume over 20% of global energy by 2030 [1]. While it is hardware which consumes energy, the software running on that hardware has major impact on the energy consumption, and hence there has been increased interest in *software sustainability* as a means to reduce the environmental impact of the digital sector.

In this paper we investigate how model-driven engineering (MDE), and specifically Agile MDE [2], can contribute to sustainable software engineering. We define sustainability analysis and improvement techniques, and implement these as extensions of the AgileUML toolset [3] for MDE, which is a lightweight tool for software specification, design and code generation, using UML class diagrams, together with a procedural extension of OCL [4] to define system data and behaviour. The proposed approach fits into an agile process, whereby refactoring to reduce or remove *sustainability debt* [5, 6] in software artefacts would be used, instead of the more usual agile methods refactoring to reduce *technical debt*.

2. MDE in software sustainability research

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 [7], refactorings [8], programming

AMDE 2024: Agile Model-driven Engineering Workshop, Part of the Software Technologies: Applications and Foundations (STAF) federated conferences, Eds. S. Tehrani, H. Alfraihi, S. Rahimi and J. Troya, 20 July 2024, Netherlands.
kevin.lano@kcl.ac.uk (K. Lano); lyan.alwakeel@kcl.ac.uk (L. Alwakeel); zishan.rahman@kcl.ac.uk (Z. Rahman)
0000-0002-9706-1410 (K. Lano)

language choices [9] and data structure choices [10, 11]. In contrast, only a few research works have considered the estimation and analysis of energy use based on software models such as class diagrams or state machines [12].

Addressing energy-use issues at the model level could reduce the development effort expended upon implementing sustainable software design at the code level, and could help to bridge the gap between sustainability requirements and implementation. However, sustainability has also been a neglected topic within MDE, with only a few papers considering this topic (e.g., [13, 14]). MDE toolsets do not provide the necessary tools to support sustainability analysis and improvement, and MDE specification and design languages such as UML and OCL do not provide any means to specify or constrain energy use.

In order to address these deficiencies, we have extended the AgileUML toolset to provide analysis to detect and remedy design elements that have potentially high energy use (Section 3). We have also extended the OCL specification language used in AgileUML with additional datatypes such as *SortedSet* and *SortedMap*, to enable more sustainable designs to be expressed in software models (Section 3.1), and we revised the core OCL libraries of AgileUML to improve their sustainability.

3. Sustainability flaws

A *sustainability flaw* in a software application is a software design or coding element which can lead to negative impact on the environment through unnecessarily high energy use by the application, and hence excessive GHG emissions. An example would be the repeated evaluation of a complex expression whose value will be the same at the different evaluation points.

The concept of *sustainability debt* [5, 6] due to sustainability flaws, is analogous to the concept of technical debt due to code quality flaws. However, unlike technical debt, the existence of sustainability debt causes an ongoing cost (to the environment) due to the use of the software containing the debt, instead of additional costs during maintenance.

Measurement or estimation of the energy use of a system is a key initial step to enable developers to detect energy use flaws in their code and to improve the sustainability of their software [15]. The more specific and localised this analysis is, the more effective the remedial actions can be. Energy measurement techniques include external power meters, internal (on-chip) power sensors, or energy predictive models based on performance monitoring counters [16]. Tools such as Green Algorithms [17], mlco2 [18], Codecarbon[1] and Carbontracker [19] provide estimates of software carbon footprint based on the use of computational resources and the location of such resources.

Since we are concerned with identifying the potential energy use ‘hotspots’ in a design, and with the relative energy use of different designs, which may be implemented using many different programming languages and platforms, exact energy measurements are not possible and are not needed, instead a generalised energy use estimation approach will be used. This involves estimating the *computational cost* or effort involved in performing an operation or statement at the design level, by breaking down any computational task into a series of basic actions. The computational cost of an execution of an operation or statement is then the sum

[1]https://codecarbon.io

of the cost of each basic action which may be executed as part of the operation/statement execution.

Unlike the detection of code quality flaws or 'code smells', the presence of energy use flaws in code cannot generally be determined without considering the *scenarios of use* of the code [20, 10, 21]. For example, in scenarios where the operations applied to a string or numeric collection are primarily membership tests together with less-frequent element additions, with no indexing used, then representing the collection by a sorted bag would typically use less processing resources than using a sequence. The same principle applies at the software design model level.

There may also be conflicts between traditional software flaw reduction goals and sustainability goals. For example, factoring out duplicated code is usually considered desirable from a quality viewpoint, but can increase energy use [8].

We have extended the quality analysis tools of AgileUML [3] to identify sustainability flaws based on estimated computational costs. Potential sustainability flaws are categorised as 'red flags' for severe cases, or 'amber flags' for moderate cases.

The flaws are detected by computational cost analysis of the design level coding of operations, which uses the procedural OCL language of AgileUML, similar to Pascal code in syntax [22]. Along with flaws, suggested refactorings or alternatives to the flawed element are suggested. Table 1 summarises the flaws and recommendations.

Table 1
Energy-use flaws and recommendations

Flaw	*Level*	*Recommendation*
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 *Sequence* type together with coding to enforce unique membership.	Red	Replace by use of *Set* or *SortedSet* if no indexing used, otherwise by *OrderedSet*.
Multiple evaluations of constant-valued complex expression	Red	Replace by new local variable v and lookups of v.
$col{\rightarrow}select(P){\rightarrow}any()$	Red	Replace by $col{\rightarrow}any(P)$
Use of reflection, process creation, network connection.	Red	Avoid unnecessary calls.
Nested loops.	Red	Restrict iteration ranges.
Nested iterators $col{\rightarrow}select(P){\rightarrow}select(Q)$, etc.	Amber	Replace by $col{\rightarrow}select(P\&Q)$, etc.
$col{\rightarrow}sort()$ for *Set* col.	Amber	Use *SortedSet* for *col*.
while or *repeat* loops.	Amber	Replace by bounded loop.
Long chains of method calls.	Amber	Replace call of chain end operation by its definition.
Floating-point operations $x{\rightarrow}pow(y)$, *MatrixLib.matrixMultiplication*	Amber	Replace by int versions if all values are *int*.

3.1. Choice of datatypes

The basic datatypes available in OCL and in AgileUML design specifications are (i) numeric types (integers and real numbers); (ii) boolean; (iii) string; (iv) enumerated types. AgileUML adds a reference type constructor $Ref(T)$ for any type T. This can be considered to define a type of pointers to T values [22].

OCL has class types, which can be regarded as a type of references to object values consisting of an aggregate of their data features [4]. Structured datatypes include sets, ordered sets, bags and sequences. AgileUML adds *Map* and *Function* type constructors.

Standard OCL collections are characterised by two aspects: whether the elements in the collection are unique (Uniqueness) and whether elements can be indexed by an integer index (Indexing). The different options for these characteristics give the four OCL collection types (Set, Sequence, OrderedSet and Bag). To support a wider choice of datatypes to enable more sustainable designs, we add two further characteristics for collections: *Sortedness* and *Fixed-size*. In AgileUML we provide a *SortedSet* collection type and *SortedMap* map type, with *SortedBag*, *SortedSequence* and *SortedOrderedSet* types available in prototype form.

The sorted versions of OCL collection datatypes enable the optimisation of OCL search operations such as $\rightarrow$*includes* and $\rightarrow$*indexOf* by using binary search instead of linear search. They also avoid the need for expensive sorting operations (of $O(n * log(n))$ computational complexity). The $Ref(T)$ types provide fixed-size sequences.

The third flaw in Table 1 refers to a situation where a specifier has defined additions to a Sequence-typed variable or attribute *col* so that these additions are always guarded by a check that the element to be added is not already in *col*:

```
post:
  col =
     if col@pre->includes(x)
     then col@pre
     else col@pre->including(x)
     endif
```

and variations on such specifications. This leads to a computational cost of the order of $col \rightarrow size()$ basic actions for each addition to the sequence, because of the membership test, instead of the usually log-based cost for an unguarded addition to a sequence. The specification is a misuse of the sequence type, and the appropriate extended OCL datatypes to use in such a situation to ensure uniqueness in a collection are either *Set*, *SortedSet* or *OrderedSet*.

Aggregate datatypes such as sets and sequences have different computational costs for various access and update operations. Table 2 summarises the typical growth rates in computational costs of common operations on a collection *s*, relative to the number *n* of elements in *s*. This assumes a hash-based implementation for sets and bags, array-based for sequences, tree-based for sorted sets and bags, and a combined set plus sequence implementation for ordered sets. The cost estimate of $s \rightarrow including(x)$ for sequences and ordered sets includes the cost of resizing actions. A common strategy is to double the size of the underlying array when more space is needed, this results in a $O(log\ n)$ cost for insertion overall. Sorted sequences consist of a sequence maintained in sorted order: adding an element to such a sequence will insert it in a position that ensures the sorted order, instead of inserting it at the end (as in the case of sequences and ordered sets), thus the cost of element insertion is $O(n)$ in general. On the other hand, using a sorted sequence can reduce the cost of operations such as $\rightarrow$*includes*, $\rightarrow$*count* and $\rightarrow$*max*. The union operation for sorted sequences will be of linear complexity. Holding bag elements in sorted order can also reduce the costs of bag equality tests (because inequality can be detected without needing to iterate through both collections), although the worst case behaviour will still be $O(n)$ in the size of the longest argument. A sorted ordered set is a pair of a set and sorted sequence.

Table 2
Computational cost complexity for collection types

Operation	*Sequence*	*Set/Bag*	*SortedSet/ SortedBag*	*OrderedSet*	*Sorted Sequence*	*Sorted OrderedSet*
$s \rightarrow includes(x)$	$O(n)$	$O(1)$	$O(log\ n)$	$O(1)$	$O(log\ n)$	$O(1)$
$s \rightarrow at(i)$	$O(1)$	—	—	$O(1)$	$O(1)$	$O(1)$
$s \rightarrow including(x)$	$O(log\ n)$	$O(1)$	$O(log\ n)$	$O(log\ n)$	$O(n)$	$O(n)$
$s \rightarrow excluding(x)$	$O(n)$	$O(1)$ (Set) $O(n)$ (Bag)	$O(log\ n)$ (Set) $O(n)$ (Bag)	$O(n)$	$O(n)$	$O(n)$
$s \rightarrow indexOf(x)$ $s \rightarrow lastIndexOf(x)$	$O(n)$	—	—	$O(n)$	$O(n)$	$O(log\ n)$
$s \rightarrow count(x)$	$O(n)$	$O(1)$ (Set) $O(n)$ (Bag)	$O(log\ n)$ (Set) $O(n)$ (Bag)	$O(1)$	$O(n)$	$O(1)$
$s \rightarrow max()$, $s \rightarrow min()$	$O(n)$	$O(n)$	$O(log\ n)$	$O(n)$	$O(1)$	$O(1)$
$s \rightarrow sort()$	$O(n\ log\ n)$	$O(n\ log\ n)$	$O(1)$	$O(n\ log\ n)$	$O(1)$	$O(1)$
$s \rightarrow asSet()$	$O(n)$	$O(1)$ (Set) $O(n)$ (Bag)	$O(1)$ (Set) $O(n)$ (Bag)	$O(1)$	$O(n)$	$O(1)$

The different datatypes also have varied memory requirements, with ordered sets and sorted ordered sets consuming more memory than sets of the same size.

3.2. Refactorings for software sustainability

According to [8], several refactorings which are usually considered beneficial to improve the quality of software, such as replacing a cloned segment of code by a call to an operation defined by the segment (the 'Extract method' refactoring), may increase energy use. Thus careful analysis is necessary to determine under what conditions a refactoring will benefit both quality and sustainability (i.e., reduce both technical debt and sustainability debt).

For the fourth case of Table 1, the potential energy flaw is characterised by clones of an expression *e* occurring in the postcondition of an operation *op* (or in the activity of *op*), and where no variable/attribute of *e* is modified by *op*.

The proposed refactoring is to introduce a new local variable *v* of *e*'s type, initialised to *e*:

```
var v : T := e;
```

and then to replace the clones of *e* within *op* by *v*. According to [8] this refactoring (Extract local variable) never increases energy use, and may decrease it. We can justify this refactoring as follows.

Denote by E the computational cost of evaluating e. Let D be the cost of the declaration of v, and L the cost of a lookup of v's value.

The cost prior to the refactoring of N evaluations of e is: $N * E$. The cost of the refactored evaluations is: $D + E + N * L$.

In the base case $N = 2$, the refactored version has lower cost if $E > D + 2 * L$. If this is the case, and if $E \geq L$, then the refactored version also has lower cost than the original for any $N \geq 2$. Generally then, the more complex e is, the more likely it is that this refactoring reduces computational cost and therefore energy use. A minimum bound on syntactic complexity can be set by the AgileUML user when clone detection and removal is applied. A similar argument applies to show that caching operation results can reduce computation costs by avoiding repeated evaluation of the same expressions.

4. Optimising design patterns for software sustainability

An initial investigation into design pattern energy use showed that introducing a pattern may either reduce or increase program energy use [7]. The analysis results showed that energy use tends to be increased by introducing patterns which involve additional object creation actions and/or additional method calls compared to the original version of a system.

Patterns which substantially reduced the number of objects (without adding extra calls), such as the Flyweight pattern, tended to reduce energy use. Related to Flyweight is the pattern Object Indexing used by AgileUML code generators to implement an ≪*identity*≫ stereotype on an attribute – the pattern enforces that objects of a class are uniquely identified by the values of the ≪*identity*≫ attribute [23]. Using this pattern avoids redundant object creations, and reduces the computational cost of accessing an object by means of an identity attribute value.

The Observer, Iterator and Adapter patterns are particularly important to examine from the viewpoint of software sustainability because of their extensive use in software applications and programming languages. Each of these patterns introduces new objects, and each can involve significant numbers of operation calls:

- Changes to a *Subject* in the Observer pattern are notified to each observing client *View*/*Observer*.
- Updates and queries on an iterator for an underlying collection may also involve delegated calls to that collection.
- Calls to the *Adapter* object are delegated to the *Adaptee*.

Observer was found to be one of the most energy-expensive patterns in [7]. In order to reduce the communication and processing costs of this pattern, a potential optimisation is to replace notifications from the subject to the views by explicit requests from each view to the subject when the view is initialised or refreshed (Figure 1). This is particularly applicable in situations such as mobile or tablet devices where only one view will be visible at a time. Such views would be refreshed when they become visible. If there are V views, each event that changes the subject data leads to a computational cost in the original version of

$$V * (C + U)$$

where C is the cost of an update call from the subject to a view, and U the local view update cost. The optimised version instead has a cost

$$C + U$$

for each view initialisation/refresh. Thus there is a potential for reducing computational costs if there are several $V > 1$ views and view refreshes occur less frequently than subject update events. If there are N update events and M refreshes over a given time period, then the optimised version reduces computational cost if: $M < V * N$. However this approach means that views may be out-of-date wrt subject data for various periods of time.

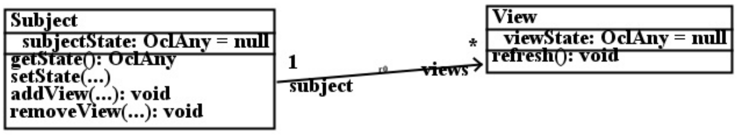

Figure 1: Optimised Observer design pattern structure

Object Indexing and the optimised version of Observer are provided as design pattern choices in AgileUML.

5. Optimising architectural patterns for software sustainability

There has been much work in the software architecture field on architectural sustainability in the sense of ensuring that an architectural design choice can support a system over the long term, including supporting system evolution. However there have been few works specifically addressing the energy-use or (environmental) sustainability implications of architecture choices [24]. This is a significant area of research because architectural choices may have substantial effects on the energy usage of software applications [25]. As with the detailed analysis of designs and evaluation of design alternatives, the scenarios of use of a system are significant when evaluating the sustainability implications of particular architecture choices, and there may be tradeoffs between other software quality goals and sustainability.

The Blackboard and MVC architectural patterns can be optimised using the demand-based approach adopted for Observer in Section 4. In the optimised versions, views update the model either directly (Blackboard) or via a Controller (MVC), but model changes are not propagated to views, instead views can query the model when they need up-to-date model information. These optimised pattern versions are provided as design choices in the AgileUML toolset.

6. Evaluation

In this section we evaluate the effect of design model choices upon the energy use of different implementations of the models, in order to show that these choices can have beneficial effects on different platforms. We consider (i) choices of data structures; (ii) refactorings; (iii) design patterns, and (iv) architectural patterns. For these experiments we evaluate energy use in milli-Watt hours (mWh) using the calculator at `https://calculator.green-algorithms.org`. The varying inputs to the computation are the processing time in ms, processor utilisation percentage and memory use in GB. We consider both Windows 10 (with Java version 8) and Linux Mint 21.3 (with Java version 21) platforms. Python and C++ implementations were also analysed, with similar results, but are not shown here. The average energy use value of three separate executions for each case is taken. All data is available at zenodo.org/records/11611486.

6.1. Data structure choices: sequences versus sorted sets

We compare Sequence-based and SortedSet-based versions of a specification which adds N distinct integers to an empty collection (scenario 1) or makes 10000 membership queries to collections of different sizes (scenario 2). For scenario 2, only the energy use of the membership checks are measured, not the total energy use.

The growth of energy use is approximately quadratic in the number of insertions for both sequence and sorted set versions in scenario 1, which implies a linear cost of a single insertion in terms of collection size, however the sorted set version has higher energy usage (Figure 2 (a)). In scenario 2 there is a linear growth in energy use with the size of the collection in the sequence version, and almost constant energy use in the sorted set version (Figure 2 (b)).

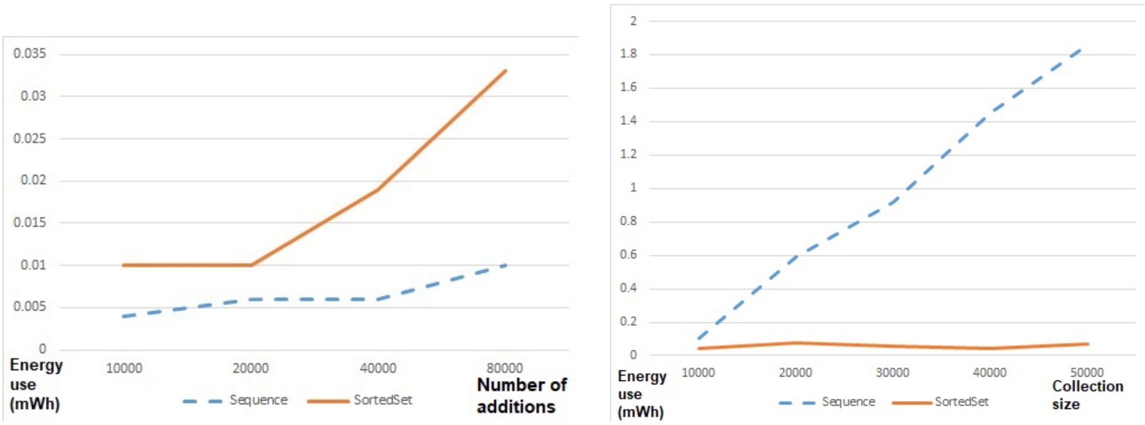

Figure 2: Sequence/sorted set energy use, Java: (a) insertions; (b) tests

6.2. Refactoring: replacing duplicated expression evaluations

Figure 3 shows the comparative energy use figures for the multiplication of two 3-dimensional matricies, using either integer-based or double-based multiplication, with 3 duplicate expression evaluations, compared with double-based multiplication with duplicate evaluations factored out. There is therefore some improvement gained by using the integer operation version, and by removing duplicated evaluations, although the differences are small.

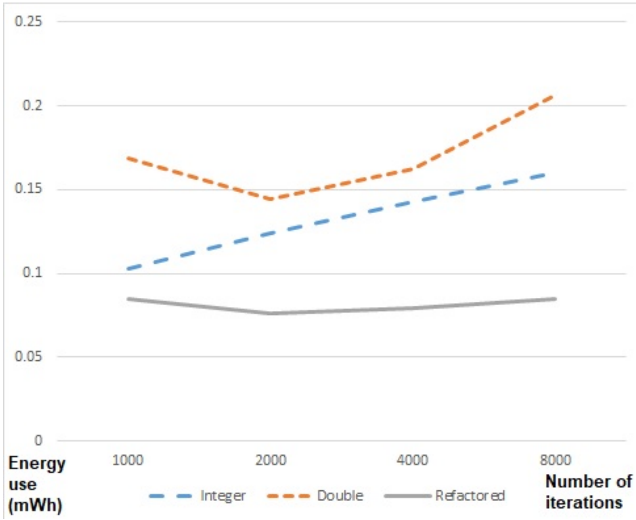

Figure 3: Matrix multiplication energy use, Java: duplicated versus factored expression evaluations

6.3. Design patterns: Observer optimisation

Here we evaluate the optimisation of the Observer pattern described in Section 4. The standard structure of Observer is used for the unoptimised version, with a single subject and N attached observers. This leads to the sending of the order of N*N messages in response to N subject update events. In contrast, in the optimised version where observers only request subject state in response to refresh events, N subject updates and N refresh events only lead to 2*N messages. Figure 4 shows that the growth of energy use also follows the pattern of quadratic growth for the original version and linear growth for the optimised version, in terms of the number of observers.

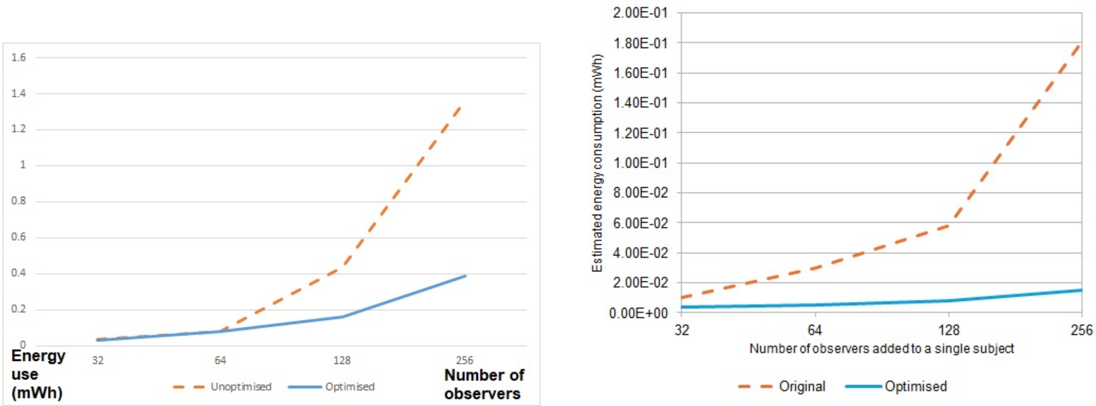

Figure 4: Observer pattern: unoptimised versus optimised versions: (a) Windows 10, Java; (b) Linux, Java

As described in Section 5, the Blackboard architectural style can be optimised in the same way as for Observer. Similar results to Figure 4 are obtained for this optimisation.

7. Conclusions and future work

In this paper we have investigated the possible contributions of MDE for achieving sustainable software engineering, and we identified tools and techniques to support sustainability analysis and improvement at the model level. We have provided extended OCL datatypes such as *SortedSet*, to enable software modellers to design more sustainable software, and we have optimised core OCL libraries. We found that it was possible to reduce software energy use by the choice of different data structures, design patterns and architectural patterns, and by the use of specific refactorings at the software model level. However in many cases these choices also have implications for the satisfaction of other non-functional requirements such as accuracy and availability, and hence decisions would need to be made by developers and stakeholders about the tradeoffs between these requirements.

We mainly considered the sustainability of software engineered using MDE. Another important topic is the sustainability of the MDE process itself, including the energy efficiency of model processing, model transformations and code generation. These are areas we will address in future work.

Acknowledgments

We acknowledge the support of UK EPSRC Seedcorn funding from the MDENet network grant. Dr Eric Zie of GoCodeGreen Ltd also contributed to this work.

References

[1] E. Zie, How to engineer more sustainable software, IT Now (2023) 14–15.

[2] H. A. A. Alfraihi, K. C. Lano, The integration of agile development and model driven development: A systematic literature review, The 5th International Conference on Model-Driven Engineering and Software Development (2017).

[3] Eclipse Agile UML project, 2024. Projects.eclipse.org/projects/modeling.agileuml, accessed 8.1.2024.

[4] OMG, 2014. Object Constraint Language (OCL) 2.4 Specification.

[5] S. Betz, et al., Sustainability debt: a metaphor to support sustainability design decisions, in: RE4SuSy Workshop, 2015.

[6] C. Becker, D. Walker, C. McCord, Intertemporal choice: decision making and time in software engineering, in: 10th International Workshop, CHASE '17, IEEE Press, 2017, p. 23–29. URL: https://doi.org/10.1109/CHASE.2017.6. doi:10.1109/CHASE.2017.6.

[7] C. Sahin, F. Cayel, I. Gutierrez, J. Clause, F. Kiamilev, L. Pollock, K. Winbladh, Initial explorations on design pattern energy usage, in: International Workshop on Green and Sustainable Software, 2012, pp. 55–61.

[8] C. Sahin, L. Pollock, J. Clause, How do code refactorings affect energy usage?, in: ESEM '14, ACM, 2014.

[9] R. Pereira, M. Couto, F. Ribeiro, R. Rua, J. Cunha, J. P. Fernandes, J. Saraiva, Energy efficiency across programming languages: How do energy, time and memory relate?, in: SLE ’17, ACM, 2017, pp. 256–267.

[10] 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.

[11] 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.

[12] L. Duarte, et al., A model-based framework for the analysis of software energy consumption, in: SBES 2019, 2019.

[13] M. Heithoff, A. Hellwig, J. Michael, B. Rumpe, Digital twins for sustainable software systems, in: 7th International Workshop on Green and Sustainable Software, IEEE, 2023.

[14] C. Vite, A.-S. Horvath, G. Neff, N. L. H. Møller, Bringing human-centredness to technologies for buildings: An agenda for linking new types of data to the challenge of sustainability, in: 14th Biannual Conference, CHItaly ’21, ACM, New York, NY, USA, 2021.

[15] P. Pathania, et al., Towards a knowledge base of common sustainability weaknesses in green software development, in: ASE ‘23, IEEE, 2023.

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

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

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

[19] L. Anthony, B. Kanding, R. Selvan, Carbontracker: Tracking and predicting the carbon footprint of training deep learning models, in: ICML workshop “Challenges in Deploying and Monitoring ML Systems", 2020.

[20] E. Jagroep, et al., The hunt for the guzzler: Architecture-based energy profiling using stubs, IST (2018).

[21] R. Pereira, T. Carcao, M. Couto, J. Cunha, J. P. Fernandes, Spelling out energy leaks: aiding developers locate energy inefficient code, Journal of Systems and Software 161 (2020).

[22] K. Lano, H. Siala, Using MDE to automate software language translation, Automated Software Engineering 31 (2024).

[23] K. Lano, et al., A survey of model transformation design patterns in practice, JSS 140 (2018) 48–73.

[24] T. Volpato, A. Allian, E. Y. Nakagawa, Has social sustainability been addressed in software architectures?, in: 13th European Conference on Software Architecture, ECSA ’19, ACM, New York, NY, USA, 2019, pp. 245–249.

[25] L. Singh, RMVRVM – a paradigm for creating energy efficient user applications connected to cloud through REST API, in: Proceedings of the 15th Innovations in Software Engineering Conference, ISEC ’22, ACM, New York, NY, USA, 2022.