Digitalization and the work against its rebound effects – sustainability as quality characteristic in the product and service life-cycle

Alexander Poth, Olsi Rrjolli, Andreas Riel

Available online at www.sciencedirect.com

ScienceDirect

Procedia CIRP 122 (2024) 861–866

www.elsevier.com/locate/procedia

31st CIRP Conference on Life Cycle Engineering (LCE 2024)

Digitalization and the work against its rebound effects – sustainability as quality characteristic in the product and service life-cycle.

Alexander Poth[a], Olsi Rrjolli[a], Andreas Riel[b,*]

[a]*Volkswagen AG, Berliner Ring 2, D-38436 Wolfsburg, Germany*
[b]*Univ. Grenoble Alpes, CNRS, Grenoble INP, G-SCOP, 46 Avenue Félix Viallet, 38000 Grenoble, France*

* Corresponding author. Tel.: +33 476825156; *E-mail address:* andreas.riel@grenoble-inp.fr

Abstract

The environmental footprint of Information Technology has become an increasingly investigated subject with in the digitalization era. Furthermore, the ever increasing power and ubiquity of computing devices and infrastructures often lead to more convenience and better experience for users, which threatens the effectivity of any sustainability measures through a rebound effect. It is crucial to consider these aspects early in the product and service life cycle to handle them adequately. This article proposes a set of aspects and relevant parameters that can be used as levers contributing to the optimization of the environmental footprint of Information Technology with a long-term perspective on the entire product and service life-cycle. Issued from action research in a large company setting, it distinguishes releases, (software) technology, electronic hardware and product/service life-cycles. In order to facilitate the management of these aspects and their effects, it proposes a set of actionable indicators that have been applied on an experimental basis in the company environment.

Keywords: sustainable Information Technology; life-cycle management; rebound efffect

1. Introduction and motivation

The digitalization is a chance to build new processes and re-think existing ones. Thereby, integrating sustainability has become a must. Sustainability of processes can be defined in different ways. In the context of digitalization, we focus on the contributions to the UN SDGs [17], especially the goal 12 (responsible production and consumption) and 13 (climate action). In the context of a company's digitalization initiative, one of the sustainability levers is the energy footprint of digital products and services, in particular in the context of Product-Service-Systems [2]. The energy footprint of an Information Technology (IT) system respectively product or service has to be assessed over the entire life-cycle (LC). For this, it is important to identify the key sustainability levers that exist in each LC phase, and their actual impact in the life cycle management (LCM). However, similarly to a mechanical assembly, the LC of an IT product or service largely depends on the different individual life cycles of the IT products and services that are integrated in the IT stack, i.e., all the IT systems that are involved in the provision of an IT product or service. In this context, this work seeks to identify the core LCs of IT products and services. It presents typical sustainability levers in the LC of software releases. Starting with the requirements of the demand to the operating to measure the impact of actions. The overall perspective is that sustainability in the context of digitalization should be regarded as a kind of quality characteristic which has to be developed explicitly, such as any other software quality characteristic of the ISO 25010:2023 [16] e.g. as a refinement of (resource unit utilization) efficiency or (adequate resource unit) scaling. This standard is typically applied to (IT) products which can be considered as a system. Furthermore, this work gives examples from the Volkswagen Group IT product T-Rex, a testing-as-a-service (TaaS) offer, to present the impact of sustainability actions associated to the LC phases.

10.1016/j.procir.2024.02.029

The remainder of this article is structured as follows: Section 2 elaborates on the background of this work, and cites related work. Section 3 provides the fundamentals required to understand the methodology applied, and explains the latter in detail. Section 4 presents results obtained from the application of the proposed methodology to an industrial case study related with product/service development. Section 5 critically discusses the results obtained in the case study, while section 6 concludes and provides an outlook on further research.

Nomenclature

IT	Information Technology
LC	Life Cycle
LCM	Life Cycle Management
LoC	Line of Code
MVP	Minimum Viable Product
TaaS	Testing-as-a-Service
UI	User Interface

2. Background

Aligned with [5], the "rebound" or "take-back" effect can by defined as the effect that lower service costs, due to increased providing efficiency, leads to an increase in services consumption. In the IT context, the rebound effect can be considered as an increased consumption of IT resources due to the ease and experience of use, e.g. []. An example to think about IT rebound effects is Wikipedia.org, which democratizes knowledge. Has the IT providing Wikipedia services a smaller CO2e footprint than the expensive and therefore limited sold books?

Decreasing costs of IT (Moore's Law [14]) respectively its services and more ease to use them motivates to think on how this service-consuming demand mechanism can be slowed down or interrupted. The Kano model is established in requirements engineering for the classification of requirements in terms of their real need [15]. From a quality characteristic perspective, this model is considered relevant in the context of digitalization and its potential rebound effects. In short, the Kano model distinguishes three types of requirements related to customer satisfaction [7]: must-have requirements, one-dimensional requirements, and attractive requirements [1]. One-dimensional requirements have a linear effect related to the satisfaction. Attractive requirements have an exponential effect. In [20], the model is extended to address customer needs more systematically by adding the type indifferent. Indifferent is an aspect which does not contribute to customer satisfaction nor dissatisfaction. An important further aspect is, that over time, an attractive requirement can become a one-dimensional requirement, and later on, a must-have requirement. This expectation movement over time is a socio-economic aspect. Who wants to vote to switch off Wikipedia.org as the de-facto standard library today (whatever the CO2e footprint relation to books is)?

However, the Kano model does only partly facilitate the definition of "sufficiency" [3]. To obtain a clearer view what sufficiency has to include or to exclude, we chose the approach to merge other useful concepts. This understanding can be used to take a look at the quality need form a Robin Hood index [8] point of view: from a social innovation perspective, one can distinguish between unserved, underserved, and overserved customers, markets, or people [4]. Back to the Wikipedia.org example, the cheap serving of high-quality information has become (over one decade) a must-have and can be considered as a basic need (sufficiency) for the people. In order to transfer this people's needs serving model to IT products and services, one has to identify what the three model groups represent in the specific IT context. The unserved customer group is happy with everything which improves their situation. The underserved customer group will start behaving as described in the Kano model: some things are expected as a "basic standard". Overserved customers are the interesting area from the sustainability perspective: what can be reduced without significant impact from the user perspective to build a more sustainable offer? In order to quantify the potentials of these optimizations, the entire LC has to be analyzed based on assumptions about the future usage and options of the product LC, respectively service LC.

3. Methodology

The methodology applied in this work merges the Kano model, the serving model, with a sustainability model to identify the relevant requirements for a sufficiency implementation in a software release within the LC from a sustainability view. The sustainability model used in this context distinguishes necessity, convenience, and luxury. To bring the models together, they have been mapped and merged iteratively. Furthermore, the model has to be able to handle the expectation change of users over the LC.

Figure 1 relates the Kano model with the Robin Hood view serving model by overlapping areas. The figure shows that Underserved contains many of the Must-have requirements, a few One-dimensional requirements and mostly non-Attractive requirements. Overserved means that the Must-have requirements, most of the One-dimensional requirements and a lot of the Attractive requirements are fulfilled. Unserved is per definition without any mapping (it can be mapped with the Kano model extension indifferently).

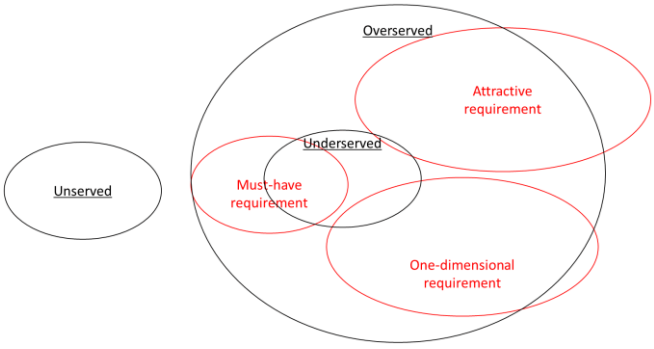

Fig. 1: Kano's model and Robin Hood's model combined (own figure)

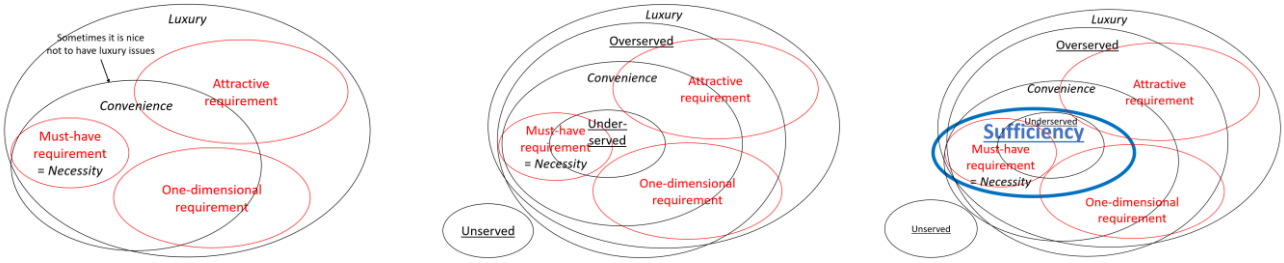

Fig. 2: a) Kano's model for convenience, luxury, necessity (own figure), b) Fig 1 and a) combined, c) Sufficiency in the combined Kano model (own figure)

Figure 2 relates the Kano model with necessity, convenience and luxury. Where luxury is defined as "things you have that I think you should not have" [18]. In the technology domain, a movement from luxury to necessity can happen in a short term, e.g. as described in [6]. Distinguishing respectively separating them into three groups with convenience is useful to define the group of "facilitating features respectively functionality". This helps to find a spot between over- and under-served. This resulting spot is the area to focus the requirements engineering to identify the sufficiency.

Figure 3 shows the result of merging Figure 1 and Figure 2. The area between convenience and underserved is the relevant area to define sufficiency in the next step.

Figure 4 presents the sufficiency area. Sufficiency contains all must-have requirements, some of the one-dimensional requirements and a few attractive requirements. Furthermore, it covers the complete necessity and underserved area. An open issue is that sufficiency does by definition not "migrate" the group of unserved into at least underserved. To do this, the question about why they are unserved has to be addressed within the demand definition respectively its requirements engineering. The presented approach only "reduces" demand requests to sufficiency and does not "expand" the demands in the direction of unserved. It also hides the dynamic of the Kano model. A feature which is an attractive requirement in the luxury area today can "move" into the sufficiency area over time. This implies that the assessment result of a feature request depends on the time in the LC. This makes it useful to evaluate the feature backlog e.g. cyclically. A not implemented feature request from the luxury area can become a necessity by the change of the usage context or the environment. An example could be the avoidance of a new fancy UI framework that consumes more processing resources. However, it can happen that in the future all state-of-the-art frameworks include this behavior as the new standard.

Based on Figure 4, we propose the following definition: Sufficiency, within the context of digitalization, has to address the necessities of user requirements. It also includes the must-haves, all requirements to avoid underserving of users and some select requirements for a basic convenience of the one-dimensional and attractive requirements. The Figure 5 transforms Figure 4 to a check-list, which can be applied during an assessment of feature requests for IT products or services. It helps to handle business demands from a sustainability perspective. A positive sustainability case is a "joker" (question 1). Every time the digitalization can contribute to the improvement of the physical world's sustainability, e.g. through a reduction of the carbon footprint, the respective action is a candidate for implementation. Questions 3 and 4 are interesting from the rebound-effect perspective. However, only demands of question 4 are "options" which can be postponed as long as possible to reduce pre-visible rebound effects. Rebound-effects caused by the actions implemented based on a positive response on Question 3, contribute to a better life, and have to be accepted in most cases. It is recommended to initiate for all "acceptable" features/functions related with question 4 a rebound-estimation as a legitimation to postpone them as long as possible.

Check-List for sufficiency decision
1. Has the feature a positive sustainability case (improve the world principle)?
   → implement it
2. Is the feature over-serving or luxury?
   → don't implement it
3. Is the feature a must-have, a necessity or implies it un-/under-serving of user?
   → implement it
4. Is the feature convenient AND close to under-serving?
   → think about the implementation, because yet it could be or soon it can become "relevant"

Fig. 3: Figure 4 transformed to a checklist

After focusing the feature/function demand to a sufficiency set of features, the product respectively service LC has to be evaluated. Each phase of the LC can contribute to footprint reduction. A smaller footprint of user value units makes rebound-effects smaller. Figure 6 integrates short-running LC's with long-running LC's, based on [12]. In this visualization the LCs from outer to inner are accelerating. One product/service LC (decades) is realized with hardware LCs (years). The team's LC is typically initiated more often by e.g. hiring new team members (months), while the software release LC is repeated every few weeks or days. This shows that the focus on the product/service LC is to stay lean and aim at maximizing sufficiency. All embedded LCs have to contribute to the overall objective. The software release LC with the high iteration frequency is the vehicle to optimize the sustainability continuously. However, software releases typically include small, incremental changes ultimately leading to bigger changes on a long-term view. So, over time the unchanged part becomes legacy and is a kind of constraint for future releases which can only be changed with significant refactoring efforts. The art is to avoid unnecessary features which are convenience

or, worse, luxury. This makes the footprint of the software bigger, with negative impact to the goal of breaking with Wirth's law that says that software grows faster than hardware performance [19]. A risk is that the software grows too fast and the hardware becomes slow respectively obsolete before its planned usage time ends. To have a chance to overcome Wirth's law, all relevant phases of a software release LC have to be addressed. First, the demand has to be shaped, e.g. with the check-list. Then, the architecture and design have to consider footprint changes respectively impacts of different realization options. The designers have to implement "green" code. The operating has to scale the deployment and infrastructure units to address the workload demand. Furthermore, the operating has to measure the current footprint to identify environmental impact improvement potentials which can be integrated in future releases.

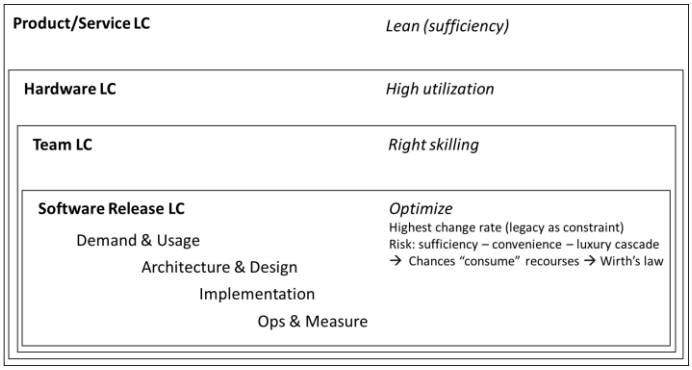

Fig. 4: Short running LCs integrated in long-running LCs

The impact of decisions about energy footprint depends on "early" decision points. Figure 7 presents the impact of the release LC phases. It indicates that an avoided feature implementation in the demand phase has the "best" footprint contribution. Also, an architecture which is able to keep the business value units isolated (scale them on demand) and small is a big deal. Then, the design with smart algorithms can still have significant impact through selecting the best fitting complexity handling with e.g. the application of the big-O, big-Omega, respectively big-Theta notation [9]. The notations define complexity boundaries of algorithms to estimate their computing and run-time demand. During implementation, the selection of an efficient language can have an impact through the reduction of code. Reduction of code mostly is realized by a careful selection of libraries. Furthermore, green coding patterns can also safe bytes and CPU cycles and some of them be checked automatically by tools during development and build procedures. During operation, a focus can be set on scaling the workload demands by keeping the utilization of the commissioned resources high. Additionally, the measurement about the resource footprint and the utilization allows closing the feedback loop to validate the impact of the optimization actions during software release development.

The feedback from operating can be used to initiate footprint improvements in future releases. Most impact comes from the selection and shaping of the demands. However, the LC actions require continuous iterative work to improve the product/service energy footprint. Systematic quality engineering shall be integrated in this endeavor as a contribution to the ISO 25010 quality characteristic (energy) efficiency.

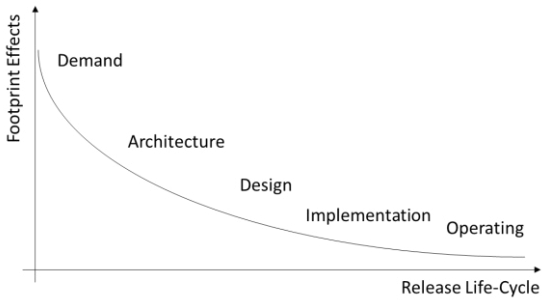

Fig. 5: The shift-left impact on future resource usage of sustainability decisions

Figure 8 shows an example of demand management with over-shaped and forgotten features. The left side of figure 8 shows the positive effects in comparison to a non-shaped demand on the right side. The demand block is the initial scope. The second block is the sufficiency scope which is delivered in the first release. As is it less effort, it is also delivered earlier. In the second release, the over-shaped and forgotten features are delivered. The initial demand typically forgets features which are recognized with the first release. In the long-run, more and more ideas become new necessity features. All further releases run in the same way in both approaches, as shown in the right block. Thereby, demand shaping reduces the software size and complexity in the long-term. The left and right upper boxes are shipped with the 4th release, however the feature amount is lower after shaping.

The approach proposed in Figure 8 to improving the sustainability fits well with agile approaches. The sufficiency demand is the scope of a Minimum Viable Product (MVP). The next increments respectively their releases integrate the "over-shaped", forgotten and new features. This approach does not delay features as it can be realized without changing the amount of releases. However, as the initial release contains less features, it can also be delivered earlier. This, in turn, reduces time-to-market, which is an increasingly important economic argument.

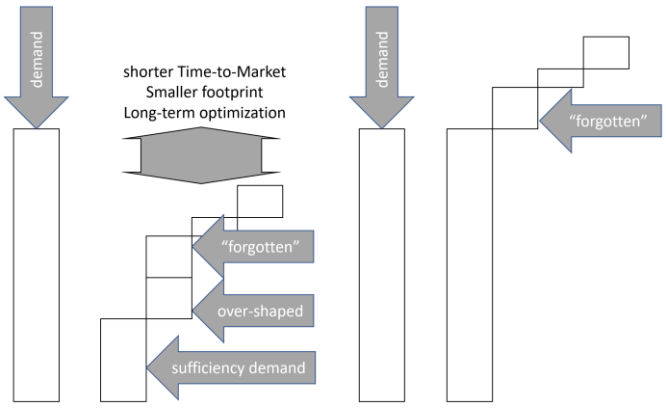

Fig. 6: Benefits of demand shaping

4. Results

The proposed methodology is meant to be applied in IT product/service development. The case study presented here is the Volkswagen AG Group IT cloud-native testing service T-Rex [13]. It shows the application of the proposed sufficiency approach to different releases of a test engine provided as a service (TaaS) to development teams at the Volkswagen AG.

Demand: The decision to integrate the new test engine was accepted after a positive sustainability case. A test engine is the tooling executing a specific test-job, e.g., for load and performance tests or one for user workflows tests. Individual fixed test engine deployments naturally have a higher footprint than an on-demand service. Therefore, T-Rex fulfills the check-list's criterium 1. In addition, the demand was shaped to a sufficiency set of features to keep the footprint of the T-Rex implementation small (criterium 3). Knowing that there is a risk of over-shaping, the initial scope was reduced to core functions and configurations. User behavior and feedback should form the scope of the next release with the over-shaped and forgotten features. This is motivated by the potential rebound-effect, because the new service approach offers the value units (i.e., executed test jobs) on demand, and it scales easily. This can lead to more testing in total, especially for the case that the current testing facilities or services "underserved" the testing demands. This can be addressed by scaling options and shows that sufficiency lowers potential rebound-effects.

Architecture: The core architectural pattern of the T-Rex architecture is to provide the demanded specific configuration for each test-job (i.e., value unit for the user) individually per user. This is realized by providing the dedicated infrastructure units to handle the specific test-job workload to avoid over-provisioning of resources. After test-job execution, the deployed resources are decommissioned. Relevant data including in particular test results are stored for the user. Therefore, value-unit resource allocation efficiency lowers potential rebound-effects.

Design: The engine-specific requirements were identified and analyzed with a footprint perspective. It is important to define the infrastructure units with the best fit to the workload type. This is the base for a high utilization and fine-grained elasticity. An example is the OpenAPI microservice release A to release B in table 2. The container base image was reduced from 638 MB to 527 MB. Furthermore, the data for configuration and results were analyzed for reduction and retention options to keep the long-term storage footprint and the related computation effort per test-job small. The usage options of the engine were analyzed to define environmental footprint-optimized default values. Moreover, the design ensures that for each test-job, the allocated resource units are associated. This is used to give feedback to the user about the service consumption footprint, and is the base to identify improvement potentials for future releases.

Implementation: To ensure a small footprint of the engine service, the micro-service was implemented with an optimized micro-service framework which also offers native compilation to platforms such as x86 or ARM. This offers provisioning of energy efficient Virtual Machines (VM) for the test-jobs.

Operating: The dynamic resource allocation is realized in the operating phase. Furthermore, the monitoring about utilization is performed during service provision in form of the test-jobs. This information is used to manage the commission and decommission of resource units and for improvement in future releases. These activities show that value-unit resource allocation efficiency lowers potential rebound-effects.

Table 1: Selenium and openAPI-test-generating microservice

Indicator/ Feature	**Release A Selenium**	**Release B Chaos Mesh**	**Release C Large Results**	**Release D Browser-Conf.**
LoC changed	Baseline LoC	474	2003	5833
Container size	378 MB	378 MB	380 MB	378MB
Container Runtime Resources	Mem: 512MB - 1GB CPU: 250-500	Mem: 512MB - 1GB CPU: 250-500	Mem: 512MB - 1GB CPU: 250-500	Men: 256Mb-768MB CPU: 250-500
Indicator/ Feature	**Release A OpenAPI-Gen**	**Release B AI-Gen (EvoM)**	**Release C AI-Refactoring**	**Release D AI-Gen workflow**
LoC changed	Baseline LoC	2389	193	1375
Container size	638 MB	527 MB	527 MB	586 MB
Container Runtime Resources	Mem: 256MB - 1GB CPU: 250-500	Mem: 256MB - 1GB CPU: 250-500	Mem: 256MB - 1GB CPU: 250-500	Mem: 256MB - 1GB CPU: 250-500

Table 1 presents selected indicators such as Lines of Code (LoC) changes of the engine specific micro-services. Release A is the reference release as baseline. The container size of the scalable deployment units to handle the test-job workload. Container resources show selected runtime resource unit allocations. The Selenium microservice (table 1a) is a deployment unit that has been optimized over years. Here, the release A is not equal to v1.0. The OpenAPI microservice (table 1b) is new and shows the shaping and optimizations in release B and C. Release C contains forgotten configurations of Release B. The Release D delivers more features and performance improvements – especially the improvements also include "footprint"-enhancements for the RAM allocation during runtime. The case study shows that eco-efficiency is still improved if no additional resources units are commissioned to deliver more functionality. The best-case is to reduce the resource unit footprint by shipping more and new functionality, however this becomes a difficult task in IT systems optimized for energy footprint over years [11]. The presented approach helps to make the right decisions to keep the resource unit footprint as small and their utilization high as possible by allowing evolvement and adaptation to the expectations of users evolving over time. But, it shows also that each new feature comes with a footprint and over time a feature which was implemented with a positive sustainability case and running with an optimized footprint can become a rebound-effect candidate in the future if the usage behavior is changing.

5. Discussion

The sufficiency set of the features and functions definition is still fuzzy and not rigorously defined by the proposed methodical approach. It is difficult to handle long-term socio-economic behavior up-front during IT product and service design. Especially, sufficiency depends on cultural and local aspects, and there is no global standard. This makes it difficult to identify and handle user and customer groups adequately. Furthermore, any user segmentation can lead to ethical issues. An idea related to the Wikipedia.org example could be to activate geo-fencing to segment people, for whom information access is more "standard" than for others. This could serve billions of requests and CO2e tonnage, but from an ethical viewpoint, this is not acceptable. This example shows that the proposed approach has to be applied with responsibility. Future research to evolve the proposed approach to be ethically correct by design is needed. Other options such as working with a price-tag for a specific functionality can be considered to handle the "moving target" from luxury (high price offer) to necessity (free-tier offer) to manage consumption and potential rebound effects. Most requirements engineering approaches [10] are not rigorous with respect to the priority of requirements for sufficiency, neither. Instead, they require an evaluation and trade-off decisions about what must-have requirements are.

Another open point is how different (enterprise) cultures will handle the proposed approach. As some terms are not rigorously defined, the organizational culture can impact the classification of features and decisions about the sufficiency set. This aspect needs further investigation in the future.

Additionally, it is not known if the parameters observed in the case study cover all those needed for showing that the approach works. However, it reveals a lot of positive effects of the T-Rex service in terms of reducing the use of IT resources through a consequent focus on sufficiency.

6. Summary and outlook

The proposed approach considers sustainability with a focus on energy footprint within the product/service LC. It sets the focus on the software release LC to leverage a systematic footprint optimization. It shows that the demand management is a key to manage potential rebound-effects by keeping the feature scope in the right balance by classification in terms of necessity, convenience and luxury to identify sufficiency. The early phases within a (release) LC have the biggest levers to impact the footprint. Sustainability becomes a quality characteristic which requires systematic quality engineering.

In practical implementation, it demonstrates sufficiency scoping facilitation with a methodical approach and check-list, as well as a high compatibility with agile approaches for easy integration into established procedures. This leads to time-to-market improvement facilitated by sustainability-oriented demand-shaping decisions.

Among the numerous questions that remain open for future work are the following: Is it possible to classify the sufficiency functionality respectively feature set more precisely? Is it possible to find generic arguments to avoid implementing potential rebound-effect driving features?

7. Author Contributions

Conceptualization, A. Poth; methodology, A. Poth, A. Riel; software, O. Rrjolli; validation, A. Poth, A. Riel; formal analysis, A. Poth; investigation, A. Poth; resources, O. Rrjolli; data curation, O. Rrjolli; writing—original draft preparation, A. Poth; writing—review and editing, O. Rrjolli, A. Riel; visualization, A. Poth; supervision, A. Poth, A. Riel.; project administration, A. Poth, O. Rrjolli.

References

[1] C. Berger, Kano's methods for understanding customer-defined quality, Center for Quality Management Journal 2 (1993) 3–36.
[2] D. Brissaud, T. Sakao, A. Riel, J.A. Erkoyuncu, Designing value-driven solutions: The evolution of industrial product-service systems, CIRP Annals 71 (2022) 553–575.
[3] D. Chen, D. Zhang, A. Liu, Intelligent Kano classification of product features based on customer reviews, CIRP Annals 68 (2019) 149–152.
[4] C.M. Christensen, H. Baumann, R. Ruggles, T.M. Sadtler, Disruptive innovation for social change, Harv Bus Rev 84 (2006) 94.
[5] L.A. Greening, D.L. Greene, C. Difiglio, Energy efficiency and consumption—the rebound effect—a survey, Energy Policy 28 (2000) 389–401.
[6] M.Z. Iqbal, A.G. Campbell, From luxury to necessity: Progress of touchless interaction technology, Technol Soc 67 (2021) 101796.
[7] N. Kano, Attractive quality and must-be quality, Journal of the Japanese Society for Quality Control 31 (1984) 147–156.
[8] B.P. Kennedy, I. Kawachi, D. Prothrow-Stith, Income distribution and mortality: cross sectional ecological study of the Robin Hood index in the United States, Bmj 312 (1996) 1004–1007.
[9] A.A. Nasar, The history of algorithmic complexity, The Mathematics Enthusiast 13 (2016) 217–242.
[10] C. Pacheco, I. García, M. Reyes, Requirements elicitation techniques: a systematic literature review based on the maturity of the techniques, IET Software 12 (2018) 365–378.
[11] A. Poth, E. Nunweiler, Develop Sustainable Software with a Lean ISO 14001 Setup Facilitated by the efiS® Framework, in: A. Przybyłek, A. Jarzębowicz, I. Luković, Y.Y. Ng (Eds.), Lean and Agile Software Development, Springer International Publishing, Cham, 2022: pp. 96–115.
[12] A. Poth, O. Rrjolli, Sustainable IT Products and Services Facilitated by "Whole Team Sustainability"–A Post-mortem Analysis, in: European Conference on Software Process Improvement, Springer, 2023: pp. 151–165.
[13] A. Poth, O. Rrjolli, A. Riel, Integration-and System-Testing Aligned with Cloud-Native Approaches for DevOps, in: 2022 IEEE 22nd International Conference on Software Quality, Reliability, and Security Companion (QRS-C), IEEE, 2022: pp. 201–208.
[14] R.R. Schaller, Moore's law: past, present and future, IEEE Spectr 34 (1997) 52–59.
[15] S. Bühne, M. Glinz, H. von Loenhoud, S. Staal, Certified Professional for Requirements Engineering Syllabus, 2022.
[16] The International Standardization Organization, ISO 25010:2023, Systems and Software Engineering., 2023.
[17] The United Nations, The 17 Goals - Sustainable Development, (n.d.).
[18] J.B. Twitchell, Living it up: Our love affair with luxury, Columbia University Press, 2002.
[19] N. Wirth, A plea for lean software, Computer (Long Beach Calif) 28 (1995) 64–68.
[20] Q. Xu, R.J. Jiao, X. Yang, M. Helander, H.M. Khalid, A. Opperud, An analytical Kano model for customer need analysis, Des Stud 30 (2009) 87–110.