A Call for Removing Variability

Mathieu Acher
Univ Rennes, CNRS, Inria, IRISA
Institut Universitaire de France (IUF)
UMR 6074, F-35000 Rennes, France
mathieu.acher@irisa.fr

Luc Lesoil
Univ Rennes, CNRS, Inria, IRISA
UMR 6074, F-35000 Rennes, France
luc.lesoil@irisa.fr

Georges Aaron Randrianaina
Univ Rennes, CNRS, Inria, IRISA
UMR 6074, F-35000 Rennes, France
georges-aaron.randrianaina@irisa.fr

Xhevahire Tërnava*
Univ Rennes, CNRS, Inria, IRISA
UMR 6074, F-35000 Rennes, France
xhevahire.ternava@irisa.fr

Olivier Zendra
Univ Rennes, CNRS, Inria, IRISA
UMR 6074, F-35000 Rennes, France
olivier.zendra@inria.fr

ABSTRACT

Software variability is largely accepted and explored in software engineering and seems to have become a norm and a must, if only in the context of product lines. Yet, the removal of superfluous or unneeded software artefacts and functionalities is an inevitable trend. It is frequently investigated in relation to software bloat. This paper is essentially a call to the community on software variability to devise methods and tools that will facilitate the removal of unneeded variability from software systems. The advantages are expected to be numerous in terms of functional and non-functional properties, such as maintainability (lower complexity), security (smaller attack surface), reliability, and performance (smaller binaries).

KEYWORDS

software variability, removing variability, software bloat

1 WHY REMOVING VARIABILITY?

To support and deliver values to a wide spectrum of users, today's software systems include an abundance of features that tend to expand over time in real-world industrial contexts [9, 29]. For instance, a highly complex system is the Linux kernel with more than 25 M lines of code and 20 K configuration options, and it keeps growing [13, 17]. Current software development approaches also strive for increasing variability in software systems. To enable expanding features, significant progress is being made in the forward and reverse engineering of software variability, which is frequently mentioned in research on software reuse, software families, or software product lines (SPL) (*e.g.*, [3, 4, 6, 14, 19]). But, in the frame of variability management and evolution, the need for a systematic removal of superfluous or underutilized variability has received far less to no attention from the community in software variability.

Since more support for usage context in a software system is desirable, the removal of variability might look counter-intuitive. Hence, the question is: *how reducing variability can be a good idea?*

The ultra-high amount of variability in software systems is exceeding human and even machine limits to deal with it, namely, to manage, test, comprehend, or even be able to use every feature or option ever [13, 17]. Xu et al. [34] show that up to 54.1% of configuration options in a system are rarely set by any user. Some systems are also completely avoided by users, who are overwhelmed by the too many choices available [16, 30]. More often, end-users lack the expertise and time to configure the system to get the right functional and non-functional properties for their usage context.

Yet, Soto-Valero et al. [27] show that up to 75.1% of software libraries that exist in a software system are *not* needed to compile and run it. Still, they are packed in the system binaries. Quite similarly, 75% of *feature toggles*, a temporary form of variability, become unused after 49 weeks [10, 22]. It's normal to imagine an underutilized feature or library could be useful in the future, yet the opposite is true. However, 89.2% of libraries as direct dependencies that are unused today in a software system will likely never be used [27]. These and similar findings show that today's software systems, ranging from small-scale size utilities in Unix [9, 18] to large-scale size web browsers, such as Chromium [20], are bloated.

Discovering the *unneeded* variability in software systems is not trivial and is a complex socio-technical task. So, *why should someone bother and remove variability from a codebase?*

Removing unneeded or unused variability from software systems is very important first because it can negatively impact the system security: the disabled or never used features in a software system may contain security vulnerabilities that could be exploited, threatening the whole system and its users. On the other hand, the attacker can for example chain small code sequences called *gadgets* and threaten the security of the system [20, 25]. The larger the binary size of a software system, the larger its attack surface [2].

Another reason lies in the system's reliability. Studies show that many software failures arise from the misconfiguration of software systems [24, 35]. In such cases finding the cause of a failure among large sets of options is difficult. Moreover, unused variability can introduce technical debt, for example, when a feature toggle has served its purpose and needs to be removed from the codebase [22].

Poor configuration choices in a software system may also result in bad or even its worst possible performance (see *e.g.*, Hadoop [8]). Assuming that the unused features may be among the poor choices and will further degrade system performance is reasonable. Moreover, software systems and their variability grow whether or not there is a rational need for them [9], and successful and widely used software systems tend to become encrusted with dubious features [18]. Some unnecessarily added features can even change the primary purpose of a system and threaten its reliability. Last but not least, attempting to use stale variability within a system has the power to bankrupt the entire company, as in the often-cited

*This is the corresponding author. The authors' names are in alphabetical order.

case of the Knight Capital Group. It unintentionally did a poor use of a stale feature toggle, causing it to lose more than 460 M dollars in 45 minutes and go bankrupt [5, 28].

Thus, exploring methods and tools for removing underutilized variability from software systems is vital and a realistic way to reduce system complexity and testing burden, improving its comprehension, reliability, security, maintainability, and performance.

2 HOW TO REMOVE VARIABILITY

Why is removing variability not yet a major trend?

First, because removing code in general, including variability, is not a rewarding activity for developers and product (line) managers, since it does not bring any new functionality or feature.

The main challenge to stakeholders in removing variability is the lack of automated or integrated technologies. In general, there are limited studies, understanding, and expertise on removing variability, specifically how stakeholders (may) operate over software artefacts in modern development workflows. To confirm our assumption, we investigated all publications in the past 10 years in the VaMoS [1] and SPLC [2] venues. Specifically, we searched the presence of "remove", "reduce", "debloat", and "delete" keywords, including their variations, in their title. It resulted that only 3 papers contain the word "reducing", but they are about reducing feature models or energy consumption, and not variability [3]. The existing technologies for removing code, such as dead code, technical debt, or software bloat, perhaps can be adapted to a certain extent. But, our call is for *removing variability*, which is different from disabling it, debloating system binaries [26, 33], or removing an unreachable piece of code.

Thus this section elaborates on possible research directions to support variability removal, discussing early attempts in literature.

Debloating variability. Software debloating has been recently explored to reduce the size of deployed containers [23], or reduce the attack surface of specific programs (*e.g.*, [7, 11, 12, 21, 25, 26, 31, 32]). Often, proposed approaches debloat a system compiled binaries [20, 33], remove its unused libraries [27], and rarely configuration options [12]. Existing approaches are all heirs of existing works in program specialization [15], so the idea of removing code is not completely new. Yet, the proposed techniques do not cover the plethora of variability units and mechanisms that can be subject to removal and debloating, namely *features, command-line options, feature toggles, design patterns, and configuration files*. For instance, removing run-time options within the source code still is an investigation direction with many challenges: How to trace and locate run-time options within the source code? How to remove a subset of run-time options without breaking other options and core functionality? How to guarantee that the removal is safe, that is, the remaining functionalities are unaffected?

Revisiting the reverse engineering of variability. Numerous works have been conducted on the reverse engineering of variability and to specifically create variability representations of artefacts in other forms or at higher levels of abstraction. Feature extraction and location have for example been subject to intensive research [1]. In our case, removing unused variability in existing systems requires locating and tracing variability that can cross-cut numerous functions, classes, modules, or files. Reverse engineering approaches have been proposed to support the maintenance and re-engineering of legacy variability-rich systems; most of the literature aims to add features and create new variants with richer functionalities. We make the point that reverse engineering techniques should be revisited in light of a new objective, that is, of removing the unneeded variability from legacy software systems while expecting novel and more precise variability location methods and metrics.

Designing for variability removal. We argue that, when developing software variability, developers should also pursue the objective of easing the removal of variability. Developers should thus find it gratifying to remove features and quantify its benefits. Any newly devised approach should support the removal of different variability units, be they features, configuration options, settings, or feature toggles. To this end, we see two complementary paths. First, the development of automated tools and methods to address a legacy, existing codebase, so that with limited effort developers can pilot tools to remove specific variability in a software system. A second and unexplored direction is to design, by construction, variability-rich systems that are "variability removal friendly". Extensibility and modularity, two key principles of software engineering, have been subject to intensive inquiry; we argue that similar research should be conducted now to ease variability removal. In both cases, the technical process should also include the removal of all other artefacts bound to a variability unit, such as tests or comments. Ideally, the remained system after removing some variability should not only be compilable and functional but also well formatted.

Developers' workflow for removing variability. Currently, there are very few proposed methodologies to support variability removal and quantifying its benefits. A notable case is the semi-automated approach of Piranha to remove stale feature toggles in Uber applications realized in Java, Swift, and Object-C [22]. In general, we are missing studies that observe how developers remove code and specifically variability-related code. We ignore what are the difficulties of removing variability. Characterizing and understanding such difficulties could help to design tools supporting their activities. There also exists little quantification of the human cost of removing variability. Another open question is how to integrate variability removal as part of modern development workflows (*i.e.*, code reviews, continuous integration, issues, ticketing systems, etc.).

Removing variability: application or domain engineering? SPL engineering usually distinguishes two complementary phases: domain engineering and application engineering [19]. Removing variability can occur within application engineering when concrete products are derived using the common and reusable artefacts developed in domain engineering. The issue is that tailoring a product to remove unneeded variability can be very specific (*i.e.*, to the product) and not reusable for other products. In a sense, the effort of removing variability would be one-shot and hard to replay in the long run, for instance, when software evolves. Hence, another possibility is to lift the removal of variability as part of domain engineering. It would

[1] VaMoS proceedings: https://dl.acm.org/conference/vamos/proceedings
[2] SPLC proceedings: https://dl.acm.org/conference/splc/proceedings
[3] The list of resulting papers: https://doi.org/10.5281/zenodo.7360593

allow developers to then systematically configure what variability can be easily removed in the future for any product. If not designed for removal, the counterpart is that this lifting has arguably a development cost. Overall, it is an open question of where to position the removal of variability as part of SPL engineering.

Scenarios for removing variability. One possible scenario is to remove all variability (*a.k.a.*, based on a usage profile), for example, fixing a configuration once and for all and forbidding any reconfiguration at run-time. This scenario is extreme and not applicable in many contexts, since users would lack the needed flexibility for their systems [33]. Another scenario is to find the unused variability units (*e.g.*, *stale* feature toggles) and remove them. In a sense, the variability space is specialized and gradually reduced. Ideally, developers should have the power to control what variability can be removed and kept.

3 CONCLUSION

As we are already aware, we are surrounded by variability in our software world. The ability of a software system to be efficiently extended, changed, and customized for use in a particular context is precisely what makes software "soft" and adaptable. Hence, although not new in the software world, the idea of *removing software variability* might sound counter-intuitive at first.

But there is evidence that software systems are bloated with variability units (*i.e.*, features and options). Removing variability is a missing piece in the research landscape in software variability. In this paper, we call for developing approaches and tools by our community which will facilitate the removal of variability and bloat from software systems, as a way to improve their security, performance, reliability, comprehension, maintainability, and testing.

REFERENCES

[1] Wesley KG Assunção, Roberto E Lopez-Herrejon, Lukas Linsbauer, Silvia R Vergilio, and Alexander Egyed. 2017. Reengineering legacy applications into software product lines: a systematic mapping. *EMSE* 22, 6 (2017), 2972–3016.

[2] Michael D Brown and Santosh Pande. 2019. Is less really more? towards better metrics for measuring security improvements realized through software debloating. In *12th USENIX Workshop on CSET 19*. USENIX, ., 1–9.

[3] Rafael Capilla, Jan Bosch, Kyo-Chul Kang, et al. 2013. Systems and software variability management. *Concepts Tools and Experiences* 10 (2013), 2517766.

[4] Rafael Capilla, Barbara Gallina, Carlos Cetina, and John Favaro. 2019. Opportunities for software reuse in an uncertain world: From past to emerging trends. *Journal of software: Evolution and process* 31, 8 (2019), e2217.

[5] Knight Capital. 2013. SEC Charges Knight Capital With Violations of Market Access Rule. https://martinfowler.com/articles/feature-toggles.html.

[6] Paul Clements and Linda Northrop. 2002. *Software product lines*. A-W, Boston.

[7] Kihong Heo, Woosuk Lee, Pardis Pashakhanloo, and Mayur Naik. 2018. Effective program debloating via reinforcement learning. In *Conf. on Computer and Communications Security*. ACM, NY, 380–394.

[8] Herodotos Herodotou, Harold Lim, Gang Luo, Nedyalko Borisov, Liang Dong, Fatma Bilgen Cetin, and Shivnath Babu. 2011. Starfish: A Self-tuning System for Big Data Analytics. In *Cidr*, Vol. 11. cse.fau.edu, California, 261–272.

[9] Gerard J. Holzmann. 2015. Code inflation. http://spinroot.com/gerard/pdf/Code_Inflation.pdf. Accessed: 2022-10-01.

[10] Juan Hoyos, Rabe Abdalkareem, Suhaib Mujahid, Emad Shihab, and Albeiro Espinosa Bedoya. 2021. On the Removal of Feature Toggles. *EMSE* 26, 2 (2021), 1–26.

[11] Hyungjoon Koo, Seyedhamed Ghavamnia, and Michalis Polychronakis. 2019. Configuration-driven software debloating. In *12th EuroSec*. ACM, NY, 1–6.

[12] Hsuan-Chi Kuo, Jianyan Chen, Sibin Mohan, and Tianyin Xu. 2020. Set the configuration for the heart of the OS: On the practicality of operating system kernel debloating. *POMACS* 4, 1 (2020), 1–27.

[13] Hugo Martin, Mathieu Acher, Juliana Alves Pereira, Luc Lesoil, Jean-Marc Jézéquel, and Djamel Eddine Khelladi. 2021. Transfer learning across variants and versions: The case of linux kernel size. *TSE* 48, 11 (2021), 4274–4290.

[14] M. Douglas McIlroy. 1968. Mass-Produced Software Components. In *NATO in SE*, J. M. Buxton, Peter Naur, and Brian Randell (Eds.). NATO, Garmisch, 88–98.

[15] Dylan McNamee, Jonathan Walpole, Calton Pu, Crispin Cowan, Charles Krasic, Ashvin Goel, Perry Wagle, Charles Consel, Gilles Muller, and Renauld Marlet. 2001. Specialization tools and techniques for systematic optimization of system software. *TOCS* 19, 2 (2001), 217–251.

[16] Katherine Noyes. 2010. Does Linux Offer Too Much Choice? https://www.linuxinsider.com/story/does-linux-offer-too-much-choice-70806.html.

[17] Leonardo Passos, Leopoldo Teixeira, Nicolas Dintzner, Sven Apel, Andrzej Wąsowski, Krzysztof Czarnecki, Paulo Borba, and Jianmei Guo. 2016. Coevolution of variability models and related software artifacts. *EMSE* 21, 4 (2016), 1744–1793.

[18] Rob Pike and Brian Kernighan. 1984. Program design in the UNIX environment. *AT&T Bell Laboratories Technical Journal* 63, 8 (1984), 1595–1605.

[19] Klaus Pohl, Günter Böckle, and Frank Van Der Linden. 2005. *Software product line engineering*. Vol. 10. Springer, Springer.

[20] Chenxiong Qian, Hyungjoon Koo, ChangSeok Oh, Taesoo Kim, and Wenke Lee. 2020. Slimium: debloating the chromium browser with feature subsetting. In *SIGSAC CCS20*. ACM, NY, 461–476.

[21] Anh Quach, Aravind Prakash, and Lok Yan. 2018. Debloating software through piece-wise compilation and loading. In *27th Sec. Symposium*. USENIX, ., 869–886.

[22] Murali Krishna Ramanathan, Lazaro Clapp, Rajkishore Barik, and Manu Sridharan. 2020. Piranha: Reducing feature flag debt at Uber. In *ICSE*. ACM, NY, 221–230.

[23] Vaibhav Rastogi, Drew Davidson, Lorenzo De Carli, Somesh Jha, and Patrick McDaniel. 2017. Cimplifier: Automatically debloating containers. In *11th Joint Meeting on Foundations of SE*. ACM, NY, 476–486.

[24] Mohammed Sayagh, Noureddine Kerzazi, Bram Adams, and Fabio Petrillo. 2018. Software configuration engineering in practice interviews, survey, and systematic literature review. *TSE* 46, 6 (2018), 646–673.

[25] Hashim Sharif, Muhammad Abubakar, Ashish Gehani, and Fareed Zaffar. 2018. TRIMMER: Application specialization for code debloating. In *ASE*. ACM, NY, 329–339.

[26] César Soto-Valero. 2021. Software debloating papers. https://www.cesarsotovalero.net/software-debloating-papers.

[27] César Soto-Valero, Nicolas Harrand, Martin Monperrus, and Benoit Baudry. 2021. A comprehensive study of bloated dependencies in the maven ecosystem. *EMSE* 26, 3 (2021), 1–44.

[28] Bishr Tabbaa. 2018. The Rise and Fall of Knight Capital - Buy High, Sell Low. Rinse and Repeat. https://medium.com/dataseries/the-rise-and-fall-of-knight-capital-buy-high-sell-low-rinse-and-repeat-ae17fae780f6.

[29] Xhevahire Tërnava, Luc Lesoil, Georges Aaron Randrianaina, Djamel Eddine Khelladi, and Mathieu Acher. 2022. On the Interaction of Feature Toggles. In *VaMoS*. ACM, NY, 1–5.

[30] Linus Torvalds. 2020. Fragmentation is Why Linux Hasn't Succeeded on Desktop. https://itsfoss.com/desktop-linux-torvalds/.

[31] Qi Xin, Myeongsoo Kim, Qirun Zhang, and Alessandro Orso. 2020. Program debloating via stochastic optimization. In *42nd ICSE: NIER*. ACM, NY, 65–68.

[32] Qi Xin, Myeongsoo Kim, Qirun Zhang, and Alessandro Orso. 2020. Subdomain-based generality-aware debloating. In *35th ASE*. ACM, NY, 224–236.

[33] Qi Xin, Qirun Zhang, and Alex Orso. 2022. Studying and Understanding the Tradeoffs Between Generality and Reduction in Software Debloating. *ASE* 1, 1 (2022), 1–13.

[34] Tianyin Xu, Long Jin, Xuepeng Fan, Yuanyuan Zhou, Shankar Pasupathy, and Rukma Talwadker. 2015. Hey, you have given me too many knobs!: Understanding and dealing with over-designed configuration in system software. In *Proceedings of the 2015 10th Joint Meeting on Foundations of SE*. ACM, NY, 307–319.

[35] Zuoning Yin, Xiao Ma, Jing Zheng, Yuanyuan Zhou, Lakshmi N Bairavasundaram, and Shankar Pasupathy. 2011. An empirical study on configuration errors in commercial and open source systems. In *23rd ACM SOSP*. ACM, NY, 159–172.