Revealing the Complexity of Automotive Software

Vard Antinyan
Volvo Car Group
Gothenburg, Sweden
vard.antinyan@volvocars.com

ABSTRACT

Software continues its procession into the core of the modern cars. Sophisticated functionalities, like connectivity and active safety, provide gratifying comfort to its users. With the sophisticated functionality, however, comes the underlying complexity that grows overwhelmingly year by year. But the invisibility of software hinders the practitioners and researchers grasping the magnitude of complexity thoroughly. Rather, from time to time, the consequences of complexity surface in forms of ultra-high design efforts, waves of defect reports, and explosions of warranty costs. This article reveals the complexity of software in four key areas of automotive software development. It points out that the existing practices are severely insufficient for systematic complexity management.

CCS CONCEPTS

• Software and its engineering → Software creation and management

KEYWORDS

Complexity, metrics, automotive software engineering

ACM Reference Format:

Vard Antinyan. 2020. Revealing the Complexity of Automotive Software. In Proceedings of the 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '20), November 8–13, 2020, Virtual Event, USA. ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3368089.3417038

1 SOFTWARE SIZE IN AUTOMOTIVE

Size is the container of complexity, but not all big software have to be complex. Size is also the simplest property indicating software magnitude [1]. Therefore, it is reasonable to present measurements of software size before going into more sophisticated aspects of complexity.

Lines of code (LOC) is used widely for size measurements. Showing sheer large numbers of LOC, however, does not resonate among the non-software experts. That is not because the amount of software in the car is unimpressive, but because the scale of software is incomprehensible due to software invisibility and abstractness. Therefore, here a figurative parallel is drawn between lines of code and lines of natural language text, where one line of code is equivalent to one line of text in a book. In this comparison an average book has 300 pages and 50 lines of text per page. One can imagine a novel like "Pride and Prejudice" by Jane Austen.

Measurement data at Volvo showed that a Volvo vehicle in year 2020 had about 100 million LOC. This means that the vehicle had software equivalent to 6000 average books, which can be enough for a decent town library. This software implements electronic fuel injection, transmission control, engine control, cruise control, integrated powertrain control, adaptive suspension, multiplex networks, navigation systems, stability control, infotainment, collision mitigation, collision avoidance, advanced driver-assistance system, active safety, connectivity, and many more advanced functionalities.

Generally, measurements indicate that automotive software has an order of magnitude increase in every ten years, starting from 2000. If this trend continues for the next 10 years, which is possible with the upcoming autonomous drive and connected cars, then Volvo in 2030 will have software equivalent to the number of books of MIT Science Fiction Society Library, which counts for 90% of all science fiction books that was ever printed in English (as of 2010). Knowing that most luxury cars have a great overlap of functionality, it is likely that they have similar magnitudes of software.

2 SOFTWARE COMPLEXITY

Complexity of a system is defined by the numbers and types of system elements and interconnection [2], [3]. Complexity is essential for endowing a system with functions and characteristics. But complexity is also the reason for the difficulty of system understanding. In the code, the interconnections of millions of such elements as operators, local variables, global variables, conditional statements, preprocessors, threads, pointers, function calls, etc. create complex units and abstraction layers of large scales. Furthermore, dependencies of code with corresponding software versions, requirements, tests, product variants, and electronic components create prodigious complexity in automotive software.

Not all complexity is essential for developing the software, however. Often, a portion of complexity builds up in the system due to non-optimal design decisions and unaware business needs. Such complexity does not contribute to the added functionality of the software; it consumes substantial effort and produces software defects, nonetheless. This is called accidental complexity [4]. Hence, this article explores four key areas of automotive software development and shows that in practice accidental complexity is a large problem. Insofar as research does not make serious efforts to methodically distinguish accidental complexity and eliminate it, the complexity management methods are going to stay severely insufficient for practice. Given the exceedingly growing software in automotive systems and in software systems in general, it is urgent that new rigorous methods are developed to overcome this problem.

2.1 Complexity of Code

Measurement at Volvo shows that as of 2020 there are 10 million conditional statements in the entire source code. These are the decision-making blocks and looping blocks. There are about three million functions defined in the entire source code, which are invoked at about 30 million places in the source code. Many of these functions conduct a task during the runtime of the vehicle. The probability that a function will produce wrong results depends on the programmer's skill, cleanness of code, quality of tests, etc. Even if this probability is stupendously small, e.g. 0.0001, the post-development defects will still count in hundreds. And they do: not because the developers are not skilled or the tests are bad, but because the aggregate complexity is prodigious. Prodigious complexity generates defects by pure probability.

Function calls and conditional statements introduce essential complexity [4]. These are elements which are essential for writing a piece of code to solve a problem. More daunting is the accidental complexity, emerging particularly from the depth of the legacy code. Code that is written 20-30 years back introduces areas of deeply nested code. Measurements of nesting, as one accidental aspect of complexity, revealed that 20% of all functions have high nesting levels (>7) in the oldest functional area of the car. New code that should be built on top of its ancestral skeleton becomes a source of lengthy discussions. Should the old code be refactored or stay untouched?

With the increasing awareness of the repercussion effect of high complexity, software engineers pay higher attention on developing simple code. Following coding guidelines has become a stricter practice than ever (e. g. [5]). A piece of code cannot be merged to the main code branch unless it complies with the complexity management guidelines; Deeply nested code, crossing signals, unjust use of pointers and global variables, inconsistent names, lengthy lines, and other accidental complexity triggers are rejected. Static analysis tools, as imperfect as they are though, attempt to help engineers in this task systematically. Legacy code is thoroughly tested and kept as isolated and unchanged as possible.

It is not clear in the guidelines, nevertheless, how much of the code complexity can be reduced. Here, a crucial problem that stays unresolved is that the existing complexity metrics do not distinguish essential and accidental complexities [1]. Meanwhile, such a distinction could help engineers with clear instructions to identify and reduce accidental complexity systematically. This question was elevated by Fred Brooks more than 30 years ago [6], and surprisingly stays dormant in the research of software engineering.

2.2 Complexity of Architecture

Measurement shows that there are about 7000 external signals, connecting more than 120 Electronic Control Units (ECU) in a modern Volvo. Furthermore, the number of internal signals, connecting software components, are two orders of magnitude larger than the number of external signals. An important task in software architecting is that architectural components should be well-isolated from one another in an orderly way, creating a system with plausible hierarchy and relatively independent compartments. This is critical for developing new functionalities on top of the existing ones tomorrow. Another, equally important task is to make architectural decisions that support maintaining such quality attributes as safety, security, reliability, performance, usability, etc.

To the contrast of the aforementioned two tasks, which coexist in a trade-off, the problem is that there are orthogonal views of architecture, according to which design decisions are made. The electronic view deals with the ECUs and their interconnections. The software design view deals with software design components and their interconnections. The business view deals with product functions, corresponding requirements, and their connections. The variant view deals with subsets of software to be compiled for corresponding vehicle variants. Constructing a superset of these views is a futile task. Juxtaposing different views in a plausible way is a challenge.

While this challenge of complexity grows, there have been techniques proposed to address it [7], [8], [9]. One attempt in simplifying the representation of architecture, which allows exploring this challenge, is the Object-Process Methodology (OPM) [10]. The idea is that two central but orthogonal views of architecture, namely, structural and behavioral, are represented in one coherent view. Stateful object and state-modifying process are two pivotal concepts in this representation. A stateful object represents an architectural design component, which has a set of states and can switch from one state to another when applying a process. Even though this method looks promising, not much data is reported on its use.

AUTOSAR has been another step to simplify complexity at least in the electronic level and in the highest software abstraction level [11]. By unifying expertise in the automotive domain, a standard separation of concerns is developed for software architecture, making the basic software, the runtime environment, and the adaptive services separate.

In relation to AUTOSAR, Service Oriented Architecture (SOA) [12] has been relatively successful in complexity reduction. It aligns the functionality and design, while compartmentalizing the services. A crucial challenge with SOA, however, is that services themselves can be complex structures, composed of many other services. This can create dependencies between the designs of the services, jeopardizing the SOA philosophy. Another crucial challenge is that certain services are time-critical and need to be prioritized by basic software processes. In modern cars, such services are abundant, making it difficult to employ a thorough SOA.

These two challenges need a proper scrutiny in the research of software architecture. But it seems that these two challenges are confined by two trade-offs, first between reuse and complexity, and second, between performance and complexity. And the way forward seems obscure so far.

In this situation it is unclear how to distinguish accidental complexity, because such a distinction depends on the particularities of the product: For example, depending on the performance requirements, certain services may need to be in the central ECU, or depending on the reusability requirements, certain services get coupled (become more complex). How much decoupling should be sacrificed for performance or reusability is difficult to quantify.

2.3 Complexity of Variants

Two ordinary people, one from China and one from Russia, would not imagine that even though they are buying the same Volvo, their cars still have slightly different behavior. And this is because of the software variants, which take care of the different environmental regulations demanded by the two countries.

Software variants introduce the third major dimension of complexity. It is remarkable that there are several hundreds of variants for the same software in order to produce cars with different parameters adjusted for different landscapes, temperatures, altitudes, emission regulations, preferences of cultures, etc. Variants provide one of the biggest competitive advantages to premium car manufacturers, allowing them to satisfy the needs of different market sectors. But at the same time they also become one of the most influential complexity sources. Variants are not parts or artifacts of software, but rather the whole software with slightly different ranges of variable values and compiled sets of codes. This means that there is a software superset producing a multiverse of software, each of them having slightly different properties. With variants come the multiplicity of the complexity of code, architecture, requirements, and tests. Variants obscure code conformance to requirements and tests. Variants complicate software compliance to ISO 26262 and Automotive SPICE. Variants make Continuous Integration a semi-complete paradigm. Variants blind the practitioners to an unfamiliar dimension of complexity explosion and urgency of action.

Figure 1 shows an example of scaling variants for a large ECU software. The names of labels are simplified on the left side and software labels are blurred on the right side because of information sensitivity and irrelevance. Four variation points from the business perspective are vehicle model, engine type, emission area, and drive type. Each of the variation points has several business variants. The seemingly small number of business variants on the left side amounts to hundred and fourteen software variants on the right side of the figure. Crucially, adding business variants linearly, increases software variants exponentially.

An advancement in dealing with variants was the development of software product lines [13], [14], [15]. Methods of managing software product lines undoubtedly decrease the complexity of software, because they help with rigorous planning and reusing software for different market needs; otherwise developing separate software development branches per market need would result in multifold larger codebase. But at the same time this does not mean that software complexity becomes as small as it would be if software had only one variant. In fact, increasing variants trigger creating more complex code, however well-designed the product line is. A tricky fact is that increasing variants create essential complexity in the software, because certain amount of software is needed to implement these variants. The variants themselves can be accidental, nonetheless.

For example, statistical analysis of variants, their sales frequency, and the associated warranty cost for a period of two years indicated that the sales and warranty cost are not at all homogenously distributed along the variants. There were variants that were selling thousands of cars, but there were also variants that were selling only few cars (I cannot disclose this statistics). Moreover, rare-selling variants were generating bigger warranty cost. There are at least two eminent strategies to reduce variants:

- Cutting down variants of insignificant markets
- Eliminating redundant variants

Reducing variants entails a rigorous investigation of which variants provide insignificant business value, so they can be cut down. Statistical analysis is usually helpful to understand which variants are sold the most. Otherwise there is a risk of engaging into lengthy discussions with business practitioners, who want to keep their market sectors unique by creating unique variants. For newly created variants customers must be kept in a close feedback loops, otherwise, however impressive a variant is, it might have insignificant business value [16]. Tesla and Apple are examples of companies which have radical approach to variants. Minimizing variants for better quality and development flexibility seems to be a paramount strategy in these companies.

It is hard to estimate in how many sold cars a variant should be used in order for it to be qualified as worthy for keeping. This is due to the complicated interplay of the added engineering complexity (unmaintainable code, defects) on one hand and the added benefit of sales on the other hand. Furthermore, the added variant increases the complexity of the entire superset of software, because all variants live in a single underlying software. Thus, the corresponding increase of the defects randomly inflicts all variants rather than that particular variant of the car.

Often, in practice not all variant requests are completely unique. Since there are many sectors in the market and the development organizations are big, sometimes a vacuum of information is created between the teams who request variants. This situation produces near-identical variants, which are accidental by-products of engineering with no value. The complexity they introduce, however, can consume substantial development, verification, and defect fixing efforts.

To this end, practitioners need methods for systematic optimization of the number of variants. Research, however, does not make serious attempts to solve this problem. Rather, the literature of software product lines is referred to, which, as indicated above, solves a different problem.

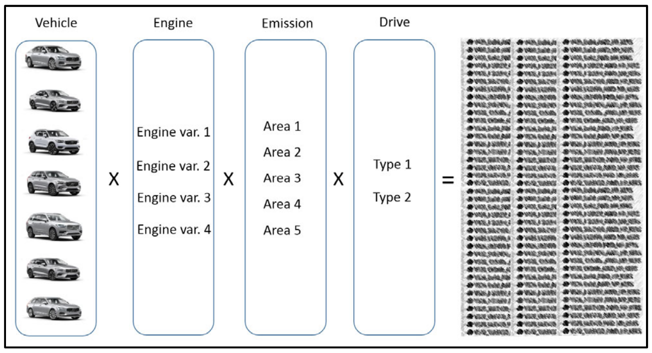

Figure 1. Software variants as a product of market demands

2.4 Complexity of Requirements

Automotive software, which has criticality in safety, reliability, and performance, needs rigorous functionality design before its implementation. Such a design primarily emerges from meticulous discussions of system designers and is represented by natural language texts and diagrams. These are software requirement specifications.

Like source code, software requirements tend to become complex too. In part, this complexity is driven by the problem domain complexity. Sophisticated conditions and state machines are typical sources of complexity. The rest of the complexity builds up accidentally, due to the inept use of the language. To reduce this complexity several techniques were proposed in literature [17], [18], [19]. In short, these techniques promote standardized patterns for writing simple requirements and recommend language constructs that decrease complexity. A crucial challenge in improving these techniques is that natural language is rich in its morphology, which hinders creating clear sets of rules for developing simple requirements. While this challenge is being resolved, there is a more problematic challenge laying in the entirety of the requirements.

Measurement shows that there are more than 100000 functional requirements for the entire superset of Volvo software. The functional requirements are broken further down to multifold more design requirements. This tremendous number of requirements then should be continually aligned with the source code and tests in order to maintain traceability. Otherwise, with upcoming defects and change requests it becomes daunting to address the respective source files and tests correctly [20]. This is particularly challenging if defects appear some years after the production. When source code undergoes changes and the corresponding requirements miss this update, a dissonance of knowledge is created, which confuses software practitioners. To complicate this matter even further, for every source function variant there must be a corresponding requirement variant. While source code variants can be handled by preprocessors inside the code, and requirements' variants can be handled by the requirement management system, the tough question is – how should the traceability be maintained between variants of source functions and variants of requirements?

This question is tightly connected to the optimization of variants, discussed in the previous subsection. But it is also tightly connected to the tools that manage code and requirements. Incidentally, the code management toolchain and the requirements management toolchain have had relatively independent paths of evolution in software engineering practice, inhibiting these two worlds to synchronize thoroughly. Understandably, developing an overarching tool that manages both requirements and code is a complex task. It is unequivocal, however, that practitioners need tools which support system thinking, when it comes to working with code and requirements as one system. Despite some intermediary tools that can help to a significant degree (e. g. Yakindu traceability) and despite the intensive research on this line [21], [22], exhaustive solutions are still far away. There is a need of a linchpin, integrating the environment of code and the environment of requirements, otherwise synchronization of code variants and requirements variants for thousands of requirements is a practically impossible task.

3 CONCLUDING REMARKS

This paper revealed software complexity in automotive domain by measurements and discussions. Furthermore, it elevated the needs for rigorous complexity management methods in four pivotal areas: code, architecture, variants, and requirements. Even though complexity takes different forms in these four areas and requires different methods for measurement, it needs an articulate distinction between essential and accidental complexities in all areas. In some cases, this distinction seems clearer; for example, unused variants should be eliminated, or deeply nested code should be refactored. In other cases, this distinction becomes more like an optimization problem; for example, understanding the optimal trade-off between performance and modularization. Additionally, there are also tooling problems introducing accidental complexity; for example, the disconnected development of code and requirements as one system. If automotive companies aspire to develop more and more sophisticated cars and keep the current pace of development, they should be more open to the research community and be more consequential towards the promised research on complexity management. The research community, on the other hand, should make more serious attempts in creating methods that can truly help practitioners reducing complexity methodically.

REFERENCES

[1] Gil Y, Lalouche G. "On the correlation between size and metric validity." Empirical Software Engineering. 22 (5): pp. 2585-611, 2017.
[2] M. W. Maier, "The art of systems architecting": CRC press, 2009.
[3] Simon HA. "The architecture of complexity." Facets of systems science (pp. 457-476). Springer, 1991.
[4] Antinyan V. "Evaluating Essential and Accidental Code Complexity Triggers by Practitioners' Perception." 2020 Feb 24, IEEE Software.
[5] Motor Industry Software Reliability Association, "Guidelines for the Use of the C Language in Critical Systems". MISRA, 2008.
[6] Brooks FP, Bullet NS. Essence and accidents of software engineering. IEEE computer. 1987 Apr;20(4):10-9.
[7] Sturtevant D. "Modular architectures make you agile in the long run." Dec 25;35(1):104-8. IEEE Software, 2017.
[8] Traub M, Maier A, Barbehön KL. Future automotive architecture and the impact of IT trends. May 15;34(3):27-32. IEEE Software, 2017.
[9] Bosch J. Achieving simplicity with the three-layer product model. Aug 28;46(11):34-9. Computer, 2013.
[10] Dori D, Crawley EF. "Model-based systems engineering with OPM and SysML." New York: Springer; 2016.
[11] Fürst S, Spokesperson AU. "Autosar the next generation–the adaptive platform." In Proc. Conf. CARS@ EDCC, 2015.
[12] Arsanjani A, Booch G, Boubez T, Brown P, Chappell D, deVadoss J, Erl T, Josuttis N, Krafzig D, Little M, Loesgen B. "The soa manifesto. SOA Manifesto." 2009 Oct;35:82-8.
[13] Famelis M, Rubin J, Czarnecki K, Salay R, Chechik M. "Software product lines with design choices: reasoning about variability and design uncertainty." International Conference on Model Driven Engineering Languages and Systems (MODELS) (pp. 93-100). IEEE, 2017.
[14] Schaefer I, Bettini L, Bono V, Damiani F, Tanzarella N. "Delta-oriented programming of software product lines." InInternational Conference on Software Product Lines 2010 Sep 13 (pp. 77-91). Springer, Berlin, Heidelberg.
[15] Chen L, Babar MA. "A systematic review of evaluation of variability management approaches in software product lines." Information and Software Technology. 2011 Apr 1;53(4):344-62.
[16] Olsson HH, Bosch J. "From opinions to data-driven software r&d: A multi-case study on how to close the'open loop'problem." Conference on Software Engineering and Advanced Applications (pp. 9-16). IEEE, 2014.
[17] Femmer H, Fernández DM, Wagner S, Eder S. "Rapid quality assurance with requirements smells." Journal of Systems and Software, pp. 190-213. 2017.
[18] Mavin A, Wilkinson P, Harwood A, Novak M. "Easy approach to requirements syntax (EARS)." IEEE International Requirements Engineering Conference (pp. 317-322). IEEE, 2009.
[19] Antinyan V, Staron M. Rendex: "A method for automated reviews of textual requirements." Journal of Systems and Software. 1; 131:63-77, 2017.
[20] Rempel P, Mäder P. Preventing defects: "The impact of requirements traceability completeness on software quality." IEEE Transactions on Software Engineering. 2016 Oct 27;43(8):777-97.
[21] Tufail H, Masood MF, Zeb B, Azam F, Anwar MW. "A systematic review of requirement traceability techniques and tools." International Conference on System Reliability and Safety (pp. 450-454). 2017.
[22] Maro S, Anjorin A, Wohlrab R, Steghöfer JP. "Traceability maintenance: factors and guidelines." International Conference on Automated Software Engineering (pp. 414-425), 2016.