In Three Steps to Software Product Lines: a Practical Example from the Automotive Industry

Matthias Eggert
Rhine-Main-Team (RMT)
Marquardt GmbH
Rietheim-Weilheim, Germany
matthias.eggert@marquardt.com

Karsten Günther
Rhine-Main-Team (RMT)
Marquardt GmbH
Rietheim-Weilheim, Germany
karsten.guenther@marquardt.com

Jochen Maletschek
Rhine-Main-Team (RMT)
Marquardt GmbH
Rietheim-Weilheim, Germany
jochen.maletschek@marquardt.com

Alexandru Maxiniuc
Rhine-Main-Team (RMT)
Marquardt GmbH
Rietheim-Weilheim, Germany
alexandru.maxiniuc@marquardt.com

Alexander Mann-Wahrenberg
Rhine-Main-Team (RMT)
Marquardt GmbH
Rietheim-Weilheim, Germany
alexander.mann-wahrenberg@marquardt.com

ABSTRACT

As with every business, being profitable does not ensure to remain competitive in the future. A competitive advantage requires a fast adaption to the market, to the regulations and customer needs. In the automotive industry, suppliers aim to increase their revenue and try to keep up with the pace of the market trends to stay competitive by offering off-the-shelf products to car manufacturers (also called original equipment manufacturers, OEMs), whereas the OEMs themselves request tailored products to gain unique selling points. Each new customer project may result in a new software project. To save time one might find it a good idea to create the new software project as a copy of an older one. Starting a new software project based on another one guarantees initial functionality, but prevents refactoring and leads to continuous software erosion. The implementations diverge from each other and improvements cannot be shared in all customer projects. Software Product Lines (SPL) can help to maximize reusability and quality by building up shared core assets and customer-specific functionality. However, migrating active projects into an SPL can be a risky task due to release schedules. The migration is especially risky if a big bang transition is the only option. In our paper, we propose a method to migrate a customer project landscape into a scalable SPL in three steps: (1) *Complexity Reduction* by reorganizing the multiple repositories (polyrepo) to a single repository (monorepo) with a new build system, (2) *Modularization* by building up core assets and making them configurable, (3) *Versioned Dependency Management* to allow reuse of versioned components across our product portfolio. This migration is possible in everyone's own pace – incrementally, iteratively and decentrally.

SPLC'22, 12–16 September, 2022, Graz, Austria

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

CCS CONCEPTS

• **Software and its engineering → Software product lines**; **Reusability**; *Software evolution*; V-model.

KEYWORDS

Software Product Line, migration, adoption, incremental migration, automotive, functionality reuse, active projects

ACM Reference Format:
Matthias Eggert, Karsten Günther, Jochen Maletschek, Alexandru Maxiniuc, and Alexander Mann-Wahrenberg. 2022. In Three Steps to Software Product Lines: a Practical Example from the Automotive Industry. In *Proceedings of 26th International Systems and Software Product Line Conference (SPLC'22).* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Development of product families rather than individual products is a goal for many engineering companies. Product Line Engineering (PLE) is a widely used approach for reaching this goal. It is common and accepted for electronic and mechanical developments and thus especially important for the automotive industry that historically comes from this area [9]. However, for quite some time the automotive industry is constantly moving its focus towards the development of software. This development is still ongoing and becomes more and more complex each year [13]. In order to stay competitive in the future and to realize a competitive advantage, a fast adaption to the market and to customer needs is also required for software. Offering off-the-shelf software products with tailored customer features will increase the company's revenue. Fortunately PLE has also been applied to software engineering for many years already in the form of Software Product Lines (SPL) [1]. An SPL increases reusability [7] and therefore increases quality and decreases development costs of software by building up shared core assets and customer-specific functionality. With this reuse in place, a business will be able to do larger scaling, with higher number of projects the development costs will drastically decrease compared to traditional engineering methods and the time-to-market will

improve due to less development effort [6]. Thus an SPL is one possible solution to win against competitors.

But even after years of research, source code migrations to SPL are still complicated and there are no bullet proof concepts available that are applicable to actively running projects with tight release plans. Much research has been done on product line architectures [28, 30] and system engineering [26], about feature model migrations [10–12, 14], SPL evolution [3, 24, 27, 29] and safety aspects [31] of SPLs. On the other hand, [16] and [2] have written about the overall process, general benefits and a change of mindset, but not specifically about source code. Contrary to [25], cloned variants are no option for us anymore. The authors of [22] reported different migration strategies more than 20 years ago already: (1) proactive, (2) extractive and (3) reactive, but no work described a mixture of extractive and reactive migrations, which we needed. Inspired by the work of [19] we aim for a strategy with which we are able to migrate multiple independent repositories to a scalable SPL, allowing developers and software product managers to proceed with their own pace and according to the project plan without risking its deadlines and always being able to do a step backwards if neccessary.

In this paper we propose an iterative and incremental migration strategy in three steps. We structured the paper in the following way: Section 2 stating the challenges we faced in our projects and industry to explain our view on the problem and why standard solutions did not work for us, section 3 explaining our three step solution of the source code migration in detail, presenting infrastructure and build system ideas and section 4 presenting our conclusion, an overview on where we stand right now and giving a forecast on future work.

2 CHALLENGE

Our migration method is supposed to be a generic solution, but our company's circumstances might have some influence on the ideas that we share in this paper. This section will explain the challenges we had, both coming from the products' nature and its industry and also from our corporate processes and strategies.

The projects that we migrated are in the area of electric cars for premium automobile manufacturers. The software is written in (Embedded) C and Matlab and modelled with AUTomotive Open System ARchitecture (AUTOSAR) with C code generators. These projects must fulfill Automotive Software Process Improvement and Capability Determination (ASPICE), ISO26262 and other processes, standards and regulations of the automotive industry during their development phases to stay competitive and meet legal requirements. A company will less likely get a new inquiry when the standards and norms are not fulfilled. The standards are relevant for the entire SPL development and do not only apply for the delivered software product, but also for supporting processes and tools like the build system. Additionally, the build system must be able to handle third party source code deliveries without modifications. One example for this are AUTOSAR deliveries. The dependency to a delivered AUTOSAR package is not only a challenge for the toolchain, but also a chance. Due to AUTOSAR's layered architecture and its interface and component design, it already provides modular abstraction for separation of concerns, that can help in building SPL core assets and to maximize functional reuse.

All migrated projects of our SPL base on the same product, a complex sensor cluster. For historic reasons every (customer) project was managed as a separate Dimensions repository. Dimensions is a Source Code Management (SCM) system similar to Revision Control System (RCS) used at the Marquardt GmbH for source code administration and project documentation. Each individual source code repository is a self-contained ready-to-build project, including all required tools and libraries to build the product binaries, like compilers, MSYS and GNU Make. The C code was validated with Jenkins nightly builds. Those Jenkins scripts were located in separate repositories, not part of the project's repository itself. Unit tests were not running with every change of the source code as part of CI/CD, but were running on demand.

Whenever a new customer project was kicked-off, it also triggered the start of a new software project. This software project then started as a clone of another already existing project ('clone-and-own'). This approach had several benefits: the setup time was short and because a mature project was taken as basis, the project's software was stable already and debugging and adapting was possible. There was less work for generic parts and only hardware and customer-specific modifications were required. However, this approach also caused a lot of disadvantages. By simply cloning an existing project there was not only a copy of functionality available in the new project, but also a copy of all its flaws. Because of a large number of copies a possible refactoring of reused code had to be merged back and this meant a high effort. If refactoring is never done, this approach can result in software erosion, as it did in our projects. The software erosion noticeably led to higher maintenance efforts at the end phase of the development and brought up errors late during the product lifecycle. For all the teams working in these projects, with the problems mentioned above, the idea of migrating to another platform including a new tool chain, not only sounded like a mammoth task but unrealistic too, having in mind the tightly planned release schedules.

With our migration to SPL we try to tackle as many as possible of the mentioned challenges. The migration is only partially done and still ongoing, but we did not detect major issues so far. At the same time, we managed to:

- migrate 'clone-and-own' sources to our SPL,
- add variant handling capability to the build system,
- replace the outdated SCM system by Git and Bitbucket,
- introduce a CI/CD system,
- introduce a unit test framework as integral part of the development,
- automate the setup of the build environment and reduce the repository size drastically,
- stay compliant with all rules and norms,
- train the team and
- still keep release plans.

We will not write about all solution aspects in this paper. The focus will be on the source code migration process and less on the tools, processes and performed trainings. Our build system implementation is freely available on Github.com [5].

3 SOLUTION

Our goal with the migration was to switch the mindset from 'my project' to 'one of our product variants' by enabling the software teams to collaborate in a single Git repository and be able to use all variants in a single IDE instance (in our case Visual Studio Code) to build any target of any product variant.

Our SPL implementation uses Visual Studio Code plus CMake Tools extension as IDE, CMake and Ninja for building all required targets, Powershell and Python as build wrapper and CI/CD test runners.

With the help of our SPL implementation we follow a migration strategy that consists of three steps as depicted in Figure 1.

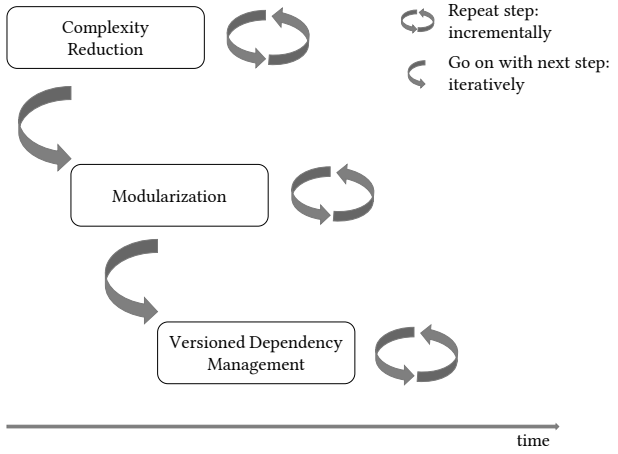

Figure 1: Migration Flow Overview

All three steps will be explained in detail in the following subsections. It is important to know though, that each step can be done incrementally:

- *Complexity Reduction*: variant by variant.
- *Modularization*: component by component.
- *Versioned Dependency Management*: component by component.

And as soon as at least one increment is done on the current step, the migration can iteratively go to the next migration step.

The content of the SPL repository will change during the different migration steps. As shown in Figure 2 the size of legacy code will first increase with every legacy project added to the codebase, then legacy code will be exchanged with configurable sources over time, and finally even configurable sources will decrease, as they are moved to a separate repository and just configured as external build dependencies in several product families.

3.1 Complexity Reduction

Our first step towards SPL consisted of importing several active sensor projects into the SPL, i.e., adding Dimension repository product variants into one structured Git repository. To support further incremental imports of additional sensor variants we implemented a tool called Transformer [4] that highly automates the necessary import steps. The Transformer was implemented in Python and could be applied to each variant separately, therefore making the transformation incremental and independent of already existing or upcoming variants. The transformation of a legacy project into a new product variant is divided into four parts:

(1) **Copy sources**: The variant's original source tree is copied into a variant-specific legacy tree without any further changes or adaptions.
(2) **Create build configuration**: The original project build system (GNU Make) is used to extract all variant-specific configuration and to transform it into a CMake configuration and part list compatible with the SPL build system.
(3) **Create IDE configuration**: For seamless integration of the SPL repository and its variant concept into Visual Studio Code the CMake Tools extension is used. This extension introduces the concept of CMake Variants, therefore we create a configuration file with build settings for each variant.
(4) **Establish CI/CD**: In order to simplify the switch to the new tool environment, introduce a CI/CD system to increase automation and validation for every source code change.

Because switching from variant-specific repositories in Dimensions to a single Git repository in Bitbucket and introducing an SPL build system, as well as a new IDE caused a high workflow impact for the developers, we decided not to change the production source code so that the software developers could continue to work in their well-known code basis.

The DevOps Handbook [21] calls this a monolithic approach. They positively emphasize the simplicity and resource efficiency in small scale, but mention the drawback of overall poor scaling and weak modularization capabilities. We are aware of those drawbacks and are going to tackle them within the third migration step.

The structure after the transformation looks like this:

```
.vscode/
  cmake-kits.json
  cmake-variants.json
  settings.json
legacy/
  variant-a/
    component-1/
    component-2/
  variant-b/
    component-1/
src/
variants/
  variant-a/
    config.cmake
    parts.cmake
  variant-b/
    config.cmake
    parts.cmake
CMakeLists.txt
```

The 'legacy' directory contains all the projects' unmodified source code directories as variant-specific directories. The 'src' directory is empty at first, it will contain the configurable sources. And the 'variants' directory contains the variant configuration in terms of compilation and feature model.

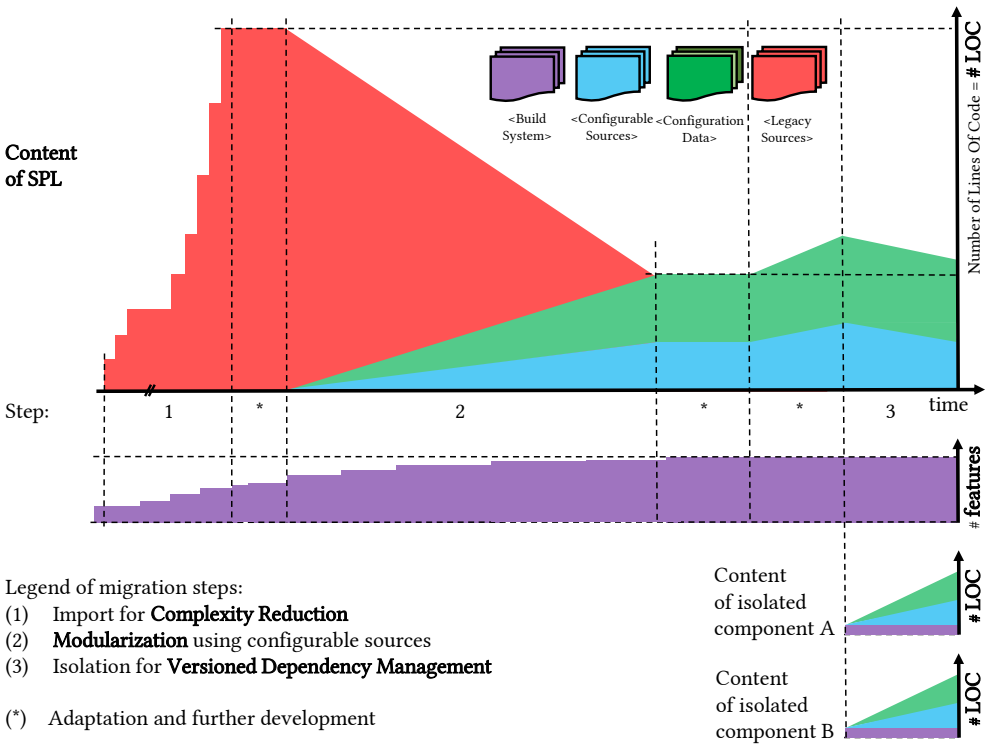

Figure 2: Three Migration Steps to SPL

Although the build system and tool environment is reused already for all variants, for source code there is no reuse in place, so scaling will become problematic. Still there are a lot of advantages with the monolithic way, like centralized deployment with our CI/CD system, start learning to use Git as source code management system and reducing complexity by reducing the number of systems: polyrepo vs. monorepo with a single build system and tool environment.

3.2 Modularization

As soon as all developers, software project managers and integrators are used to the outcome of the first step of the SPL migration, the *Modularization* step can begin. With the following three stages of *Modularization* it is possible to switch from a project to a product perspective. Thinking in terms of products rather than projects helps to reduce the amount of code and maximizes the reuse across all variants. However, in order to properly build up software components, consisting of core assets and customer-specific functions, a feature model definition is required, allowing a configuration of the different software component characteristics. The goal is to merge the different legacy software components with similar functionality into configurable sources, keeping customer-specific functionality, to maximize reuse and reduce maintenance. Although CMake is capable of handling very simple feature models with variables and template generators pretty well, a more complex feature model might require another tool to manage it. Our recommendation for complex feature model configurations is KConfig, 'a domain specific language designed specifically for coding the Linux kernel variability model' [3, page 3]. KConfig is able to manage the complex design of the Linux kernel and it has enough functionality to work with most embedded applications. Additionally it is open-source and free to use for anyone. Alternative solutions might also solve the configuration purpose. Prominent examples are pure:variants [23] from pure systems or Gears from BigLever [18]. Both tools are known to be highly flexible and easy to integrate into existing projects [15].

Independent of the feature model tool, the modularization process should be the same. All stages of the *Modularization* step are visible in Figure 3 and will be further described in detail:

Migrate from legacy components into product components: After the first step *Complexity Reduction*, all source code files are located in a dedicated legacy directory. This legacy directory contains a folder for each variant, holding all software components separately. Using a component in multiple variants is possible, but does not make much sense. There is no reason to share the variant component variant-a/component-1 also in variant-b. The components do not belong to a variant, but to the entire product. In this step, components are just moved into a generic source directory. The folder structure will change from:

```
legacy/
  variant-a/
    component-1/
    component-2/
  variant-b/
    component-1/
```

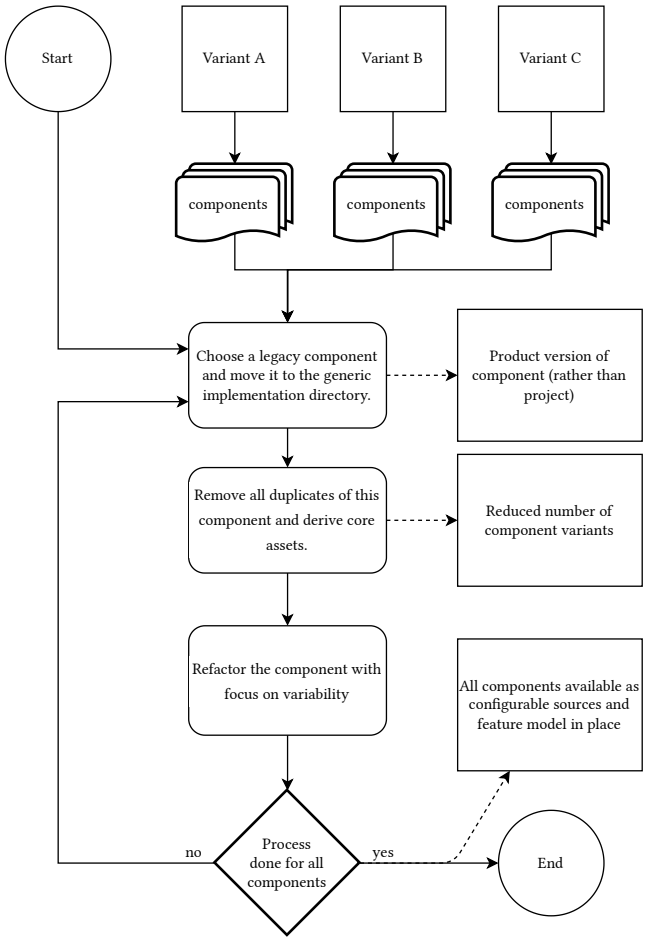

Figure 3: Detailed stages of the *Modularization* step.

to:

```
src/
  component-1/
    variant-a/
    variant-b/
  component-2/
    variant-a/
```

By doing this, all variants of one component, with the identical or very similar functionality, are moved to the same directory. Variants of a component residing next to each other are easier to compare and to handle. This will enable reuse already, but without the following stages it has no positive effect on source code reduction or maintenance effort.

Remove duplicates and create core assets: As a first step within this new structure, we recommend to clean up duplicates. Components are considered as duplicates if they are identical and no configuration is required. This is a fast and easy step for reuse and will not require additional knowledge on a feature model nor on the source code. Removing duplicates is followed by building up core assets and variant-specific functionality with a decorator pattern. The separation can be done on a function level. This approach is enabled by the build system and requires static function interface definitions. This way the core assets and the decorators can be developed without any need of conditional compiler directives. The concept is to create a:

- core asset C file, which contains generic code across all variants, considered to be static,
- a header file with interface descriptions of core assets and decorators,
- a decorator C file for every variant, which contains a variant-specific implementation of a function; the function signature must be identical for all variants and shall be defined in the according header file.

Like this it is possible to implement variant-specific differences in an object-oriented manner in C. The idea is to provide a common core asset, which is meant to stay unchanged, and additionally one interface with multiple decorators that extend the shared functionality. The feature model's responsibility is to take control of the underlying build system and configure the correct decorators. If a separation into core assets and decorators in individual files is not possible or not useful for any reason, another option would be to use a feature configuration with conditional preprocessor directives or runtime configuration. Toggle points in the code make it possible to select different implementations within the same C file. The feature model's job is to configure features and control the execution flow.

Refactor with focus on variability: Last but not least, the code should be refactored according to variability. This is the first refactoring step which has a functional influence on the code. Previous steps only changed when/how the source code was compiled. This step is also important because of the mindset shift. One must stop thinking about project specific features and define component specific features relevant for the product family. Most of the legacy components did not support variant handling and some might require substantial refactorings to have a meaningful feature set.

We highly encourage to implement unit tests before the refactoring starts, if there are no unit tests available already. During this step the availability of fast unit tests covering all core assets and features will ensure that no functionality was broken and will reduce time in finding errors at a later development phase. Even the non-functional refactorings can have huge affects on the compiled binaries and change the behavior, even though it was not intended by the developer. Using features will generate many possibilities to configure the software. Written unit tests should use the same features to enable/disable tests or expectations in order to cover all possible execution paths.

3.3 Versioned Dependency Management

In an SPL we talk about at least two variation dimensions: space, time [20] and sometimes also maturity [17]. Our first migration step *Complexity Reduction* is merely a step to prepare these dimensions, to bring code together that belongs together within a new SPL-capable build system. The second step *Modularization* establishes the space dimension by building up configurable sources or variants of software components to reduce lines of code and maximize reuse.

With variation in space we are able to create feature variants of our software and its components and this is the beginning of the SPL idea. This section will highlight the time dimension and how the same approach can be used for cross-product component reuse.

Time dimension: By using Git it is already possible to get a variation in time. At any created commit, it is possible to create a new branch. This branch can be the active development branch with all the latest changes, or it can be based on a former commit to represent a baseline of the SPL that is going to be released or was released already. If bug fixes are required after a release happened, they will be done on the same release branch. With this strategy, releases can be done on a stable code basis without the influence of new feature developments of other branches. Additionally the documentation of release relevant changes becomes simple. The release branch will stay unchanged and usually not get any feature updates, but only fixes. But release branches in a monorepo come with some drawbacks. It is complicated to get different versions of different components. A baseline must be done on the entire software. An integration of mixed versions is possible but requires lots of manual effort and will likely have a strong negative impact on maintenance. Adding a fix in a component of a specific version in a release branch will not become available in the same version of the same component in any other branch. All commits and branches are independent of each other, so no automatic reuse of code between them is available. Therefore using the same component and version provides no reuse in the space dimension if we introduce the time dimension like this. And finally fixing a bug in a new version, that was spotted in an earlier version, is even more complicated, especially across all branches. Our proposal requires some more changes to the build system and infrastructure. It is required to split up the repository again. With the modularization being done, the split will not be done per project anymore, but per software component. Each modularized software component will reside in its own repository and is maintained separately. This allows us to work with different developers or development teams on different software components independently. To enable the development of a software component in its own repository the following requirements must be fulfilled:

- component specific requirements specifications,
- mocked required interfaces,
- unit tests for all component features in all possible configurations,
- software integration tests,
- Software-In-the-Loop (SIL) tests and
- a software test plan must exist.

Although this step introduces a polyrepo workflow it still follows the SPL concept by dividing products into smaller pieces still developed in SPL repositories. The SPL repositories of the products do not contain the sources of the isolated components anymore but each SPL repository has dependencies to specific versions (baselines) and variants of the isolated components. These dependencies might be references to commits of a Git repository or to released component package artifacts, e.g., with JFrog Conan.io. This step can be done incrementally again, component by component. The SPL repository's build system will then do the integration job. With a proper dependency management, the build system will integrate requested software components from external repositories of any kind and configure them according to the feature model of the selected variant. The benefits of this approach are the following:

- Components are being developed and tested independently to get a faster release loop.
- A fix of a buggy version is implemented only once in the component's repository.
- Buggy versions can be marked in the components' repositories. This way variant builds containing a buggy version can print a warning or error. The integrator can then manually update the version of the dependency or justify the warning in regards to their project setup. An automatic update would also be possible.

Since all components are being developed independently, one package version might not be compatible with another component's package. With Conan.io 'compatibility can even be configured and customized on a per-package basis' [8]. This is why we recommend Conan.io over Git submodules or similar techniques. The development team together with their integrators can configure this compatibility check. An AUTOSAR architecture might make this check easier. An application software component should only have interfaces to the Run-Time Environment (RTE), so based on the RTE configuration a compatibility between components should be clearly defined.

Cross-product component reuse: Independent of an SPL introduction, it is most likely that within the product portfolio of an automotive supplier like Marquardt with a wide range of different products, those products share similar components and solutions. When such a company applies SPL engineering for one of its products, sooner or later the concept is taken over for other products of other business units, too. In case the second step of SPL migration *Modularization* is finished in at least two SPL repositories with similar core assets and step three has been performed on a few components, the probability, that the need to share those components between the different SPL repositories will arise, is rather high. Therefore the third step of our migration concept provides an additional useful feature: isolation of configurable sources for *Cross-product component reuse*. The idea is that the isolated software component and all its variants can be used in multiple other SPL repositories, not just in one. There are functionalities, like current measurement, that might be useful in a multitude of electronic products. This can exponentially increase reuse in the entire product portfolio.

4 CONCLUSION AND FUTURE WORK

This work provides an iterative and incremental migration strategy towards SPL for the software of a real-world product family of an automotive supplier. We proposed three steps to perform the migration of a project's software repository to an SPL variant: (1) *Complexity Reduction*, (2) *Modularization* and (3) *Versioned Dependency Management*. It was shown that our proposal can be applied to the repositories of a product family's software at any time of the development phase due to the fact that any of the three steps can be applied incrementally and each transformed variant can iteratively continue with the next migration step independent of the others. With our proposal we:

- established a stable SPL with variants buildable at any time,
- reduced workload of developers by reducing the number of repositories,
- increased collaboration beyond project borders and
- integrated all software sources of a product family into one git repository.

We implemented a build system with modern CMake fully supporting the concept of SPL variants.

At the submission date's point of time we managed to proof about 50% of our concept. The build system prototype required two persons full-time for six weeks, the implementation for migration step one and two required two persons full-time for three more months. Over a time frame of two months we incrementally added ten variants, while each import to the SPL was really fast due to the automated importing mechanism. We already started with *Modularization* of software components, but only converted a few components and begun with removing duplicates. Recently we focussed more on automation than on the SPL migration, as we saw more advantages there in the long run. The automation for CI/CD is helpful for the migration as well. Nevertheless, the build system requires some more features to be able to handle our migration step three and we plan to implement them in the coming months, in parallel to the *Modularization* step of the software components.

Additionally to the build system implementation and the actual migration of the embedded source code, it is required for us to qualify tools which have an impact on the created binaries or the processes to build them. This tool qualification is requested by the ISO 26262 standard for safety reasons. We must take care that CMake, Ninja and KConfig are properly qualified. On the other hand, proprietary software might come with safety evaluations already. This should be kept in mind, when deciding for your tool landscape.

Besides all the discussed technical challenges, there were also social obstacles on the way to SPL. Fear was probably the strongest emotion we faced. At the beginning we were certain that all required build system features were implemented and we always had a backup plan to go back to the original repositories. There was no real threat and still no one had enough confidence to start the migration. Afterwards we do not see a reason for hesitation, but sometimes it simply needs someone brave in the team or someone pushing you. Our conclusion is: Just do it! And if you break it, make sure you can fix it fast!

REFERENCES

[1] 2000. Software Product Lines; Experiences and Research Directions, Proceedings of the First International Conference. In *SPLC*, Patrick Donohoe (Ed.).

[2] Muhammad Abbas, Robbert Jongeling, Claes Lindskog, Eduard Paul Enoiu, Mehrdad Saadatmand, and Daniel Sundmark. 2020. Product line adoption in industry: an experience report from the railway domain. In *SPLC '20: 24th ACM International Systems and Software Product Line Conference, Montreal, Quebec, Canada, October 19-23, 2020, Volume A*, Roberto Erick Lopez-Herrejon (Ed.). ACM.

[3] I. Maria Attarian, Kacper Bak, and Leonardo Passos. 2011. Software Product Line Evolution: the Linux Kernel.

[4] Avengineers. 2022. *Python based tool to bring legacy GNU Make projects into SPL structure with CMake.* https://github.com/avengineers/SPLTransformer

[5] Avengineers. 2022. *Software Product Line implementation for C development based on PowerShell, CMake and Python.* https://github.com/avengineers/SPL

[6] Maider Azanza, Leticia Montalvillo, and Oscar Díaz. 2021. 20 years of industrial experience at SPLC: a systematic mapping study. In *SPLC '21: 25th ACM International Systems and Software Product Line Conference, Leicester, United Kingdom, September 6-11, 2021, Volume A*, Mohammad Mousavi and Pierre-Yves Schobbens (Eds.). ACM.

[7] Paul Clements and Linda Northrop. 2002. *Software Product Lines: Practices and Patterns.* Addison-Wesley.

[8] JFrog Conan.io. 2022. *Welcome to Conan C/C++ Package Manager Documentation.* https://docs.conan.io/en/1.43/index.html

[9] Yanja Dajsuren and Mark van den Brand. 2019. Automotive Software Engineering: Past, Present, and Future. In *Automotive Systems and Software Engineering.* Springer.

[10] Slawomir Duszynski, Saura Jyoti Dhar, and Tobias Beichter. 2019. Using relation graphs for improved understanding of feature models in software product lines. In *Proceedings of the 23rd International Systems and Software Product Line Conference, SPLC 2019, Volume A, Paris, France, September 9-13, 2019*, Thorsten Berger, Philippe Collet, Laurence Duchien, Thomas Fogdal, Patrick Heymans, Timo Kehrer, Jabier Martinez, Raúl Mazo, Leticia Montalvillo, Camille Salinesi, Xhevahire Tërnava, Thomas Thüm, and Tewfik Ziadi (Eds.). ACM.

[11] Thomas Eisenbarth and Daniel Simon. 2001. Guiding Feature Asset Mining for Software Product Line Development. In *Proceedings of the First International Workshop on Product Line Engineering: The Early Steps: Planning, Modeling and Managing (PLEES'01), Erfurt, Germany, September 2001 (Artikel in Tagungsband).* Fraunhofer IESE. ftp://ftp.informatik.uni-stuttgart.de/pub/library/ncstrl.ustuttgart_fi/INPROC-2001-85/INPROC-2001-85.pdf

[12] Claudia Fritsch, Richard Abt, and Burkhardt Renz. 2020. The benefits of a feature model in banking. In *SPLC '20: 24th ACM International Systems and Software Product Line Conference, Montreal, Quebec, Canada, October 19-23, 2020, Volume A*, Roberto Erick Lopez-Herrejon (Ed.). ACM.

[13] Klaus Grimm. 2003. Software technology in an automotive company: major challenges. IEEE Computer Society, Piscataway, NJ.

[14] Sten Grüner, Andreas Burger, Tuomas Kantonen, and Julius Rückert. 2020. Incremental migration to software product line engineering. In *SPLC '20: 24th ACM International Systems and Software Product Line Conference, Montreal, Quebec, Canada, October 19-23, 2020, Volume A*, Roberto Erick Lopez-Herrejon (Ed.). ACM.

[15] Sten Grüner, Andreas Burger, Tuomas Kantonen, and Julius Rückert. 2020. Incremental Migration to Software Product Line Engineering.

[16] William A. Hetrick, Charles W. Krueger, and Joseph G. Moore. 2006. Incremental return on incremental investment: Engenio's transition to software product line practice. In *Companion to the 21th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2006, October 22-26, 2006, Portland, Oregon, USA*, Peri L. Tarr and William R. Cook (Eds.). ACM.

[17] BigLever Software Inc. 2020. The Systems and Software Product Line Engineering Lifecycle Framework. Whitepaper.

[18] BigLever Software Inc. 2022. *Gears Product Line Engineering Tool and Lifecycle Framework.* https://biglever.com/solution/gears/

[19] Hans Peter Jepsen, Jan Gaardsted Dall, and Danilo Beuche. 2007. Minimally Invasive Migration to Software Product Lines. In *Proceedings of the 11th International Software Product Line Conference.* IEEE Computer Society. http://doi.ieeecomputersociety.org/10.1109/SPLINE.2007.30

[20] Kyo C. Kang, Vijayan Sugumaran, and Sooyong Park (Eds.). 2009. *Applied Software Product Line Engineering.* CRC Press, Taylor and Francis Group.

[21] Gene Kim, Jez Humble, Patrick Debois, and John Willis. 2016. *THE DEVOPS HANDBOOK.* IT Revolution Press, LLC.

[22] Charles W. Krueger. 2001. Easing the Transition to Software Mass Customization. In *Proceedings of the Fourth International Workshop on Product Family Engineering (PFE-4)*, European Software Institute (Ed.). Bilbao, Spain.

[23] pure systems. 2022. *Systematic variant management with pure::variants.* https://www.pure-systems.com/purevariants

[24] Clément Quinton, Michael Vierhauser, Rick Rabiser, Luciano Baresi, Paul Grünbacher, and Christian Schuhmayer. 2021. Evolution in dynamic software product lines. *J. Softw. Evol. Process* (2021).

[25] Julia Rubin, Krzysztof Czarnecki, and Marsha Chechik. 2013. Managing cloned variants: a framework and experience. In *17th International Software Product Line Conference, SPLC 2013, Tokyo, Japan - August 26 - 30, 2013*, Tomoji Kishi, Stan Jarzabek, and Stefania Gnesi (Eds.). ACM. http://dl.acm.org/citation.cfm?id=2491627

[26] Andreas Schäfer, Martin Becker, Markus Andres, Tim Kistenfeger, and Florian Rohlf. 2021. Variability realization in model-based system engineering using software product line techniques: an industrial perspective. In *SPLC '21: 25th ACM International Systems and Software Product Line Conference, Leicester, United Kingdom, September 6-11, 2021, Volume A*, Mohammad Mousavi and Pierre-Yves Schobbens (Eds.). ACM.

[27] Daniel Simon and Thomas Eisenbarth. 2002. Evolutionary Introduction of Software Product Lines. In *SPLC 2: Proceedings of the Second International Conference on Software Product Lines.* Springer-Verlag, London, UK.

[28] Mikael Svahnberg and Jan Bosch. 1999. Characterizing Evolution in Product Line Architectures. In *Proceedings of the 3rd annual IASTED International Conference on Software Engineering and Applications*, N. Debnath and R. Lee (Eds.). Anaheim, CA.

[29] Mikael Svahnberg and Jan Bosch. 1999. Evolution in Software Product Lines: Two Cases. *Journal of Software Maintenance: Research and Practice* 11, 6 (1999).

[30] Oleksandr Tomashchuk, Dimitri Van Landuyt, and Wouter Joosen. 2021. The architectural divergence problem in security and privacy of eHealth IoT product lines. In *SPLC '21: 25th ACM International Systems and Software Product Line Conference, Leicester, United Kingdom, September 6-11, 2021, Volume A*, Mohammad Mousavi and Pierre-Yves Schobbens (Eds.). ACM.

[31] Christian Wolschke, Martin Becker, Sören Schneickert, Rasmus Adler, and John MacGregor. 2019. Industrial perspective on reuse of safety artifacts in software product lines. In *Proceedings of the 23rd International Systems and Software Product Line Conference, SPLC 2019, Volume A, Paris, France, September 9-13, 2019*, Thorsten Berger, Philippe Collet, Laurence Duchien, Thomas Fogdal, Patrick Heymans, Timo Kehrer, Jabier Martinez, Raúl Mazo, Leticia Montalvillo, Camille Salinesi, Xhevahire Tërnava, Thomas Thüm, and Tewfik Ziadi (Eds.). ACM.