Getting Rid of Clone-And-Own: Moving to a Software Product Line for Temperature Monitoring

Elias Kuiter
Otto-von-Guericke-University
Magdeburg, Germany
kuiter@ovgu.de

Jacob Krüger
Harz University of Applied Sciences
Otto-von-Guericke-University
Wernigerode & Magdeburg, Germany
jkrueger@ovgu.de

Sebastian Krieter
Harz University of Applied Sciences
Otto-von-Guericke-University
Wernigerode & Magdeburg, Germany
skrieter@hs-harz.de

Thomas Leich
Harz University of Applied Sciences
METOP GmbH
Wernigerode & Magdeburg, Germany
tleich@hs-harz.de

Gunter Saake
Otto-von-Guericke-University
Magdeburg, Germany
gunter.saake@ovgu.de

ABSTRACT

Due to its fast and simple applicability, clone-and-own is widely used in industry to develop software variants. In cooperation with different companies for thermoelectric products, we implemented multiple variants of a heat monitoring tool based on clone-and-own. After encountering redundancy-related problems during development and maintenance, we decided to migrate towards a software product line. Within this paper, we describe this case study of migrating cloned variants to a software product line based on the extractive approach. The resulting software product line encapsulates variability on several levels, including the underlying hardware systems, interfaces, and use cases. Currently, we support monitoring hardware from three different companies that use the same core system and provide a configurable front-end. We share our experiences and encountered problems with cloning and migration towards a software product line—focusing on feature extraction and modeling in particular. Furthermore, we provide a lightweight, web-based tool for modeling, configuring, and implementing software product lines, which we use to migrate and manage features. Besides this experience report, we contribute most of the created artifacts as open-source and freely available for the research community.

CCS CONCEPTS

• **Software and its engineering → Software product lines**; *Software configuration management and version control systems*; *Software reverse engineering*;

KEYWORDS

Software Product Line, Case Study, Feature Modeling, Extraction

SPLC '18, September 10–14, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-6464-5/18/09…$15.00
https://doi.org/10.1145/3233027.3233050

ACM Reference Format:
Elias Kuiter, Jacob Krüger, Sebastian Krieter, Thomas Leich, and Gunter Saake. 2018. Getting Rid of Clone-And-Own: Moving to a Software Product Line for Temperature Monitoring. In *SPLC '18: 22nd International Systems and Software Product Line Conference, September 10–14, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3233027.3233050

1 INTRODUCTION

Software product lines are a systematic approach to reuse and manage software artifacts [2, 36]. These artifacts correspond to *features* – user-visible functionalities of a set of variants – that are modeled within variability models [15, 42] to define their dependencies. A selection of features that fulfills all these dependencies is a *valid* configuration. Based on such a configuration, a tool can automatically instantiate a variant from the implemented artifacts.

Using software product lines promises several benefits, for instance, reduced costs for development and maintenance, faster time-to-market, and improved quality [2, 23, 47]. Nonetheless, developing a software product line requires higher initial investment and careful investigation of whether it is suitable for the task at hand [13, 28, 43]. Thus, many organizations start with a single system instead, which is then cloned and adapted to new customer requirements—the clone-and-own approach [16, 18]. As this approach creates separated software variants, it can quickly become expensive to maintain, due to the necessary change propagation for updates [16, 35]. For this reason, organizations often decide later on to migrate these cloned variants towards a more systematic approach; adopting, for example, software product lines [9, 36, 43]—which is called *extractive approach* [26].

In this paper, we describe our experiences with implementing a set of similar variants in the temperature monitoring domain. We started to implement these variants to address personal needs, but in the process attracted different organizations to adopt and extend the variants for distributing them to their own customers. Due to the resulting adaptations, the initially used clone-and-own approach was not feasible anymore and we decided to extract a software product line. To this end, we also implemented our own tooling to facilitate development and maintenance for our purpose. Consequently, with this paper we contribute the following:

- We describe our experiences of developing a set of variants in the temperature monitoring domain. This includes the development based on the clone-and-own approach (cf. Section 2) and the problems we faced. Furthermore, we discuss the benefits we hoped to achieve with a software product line as well as the migration process (cf. Section 3).
- We describe and provide lightweight tooling especially for web-based software-product-line engineering (cf. Section 5). To this end, we briefly discuss the reasons for implementing new tools instead of relying on existing ones, such as, FeatureIDE [34], pure::variants [10], and Gears [27]. Our tooling is inspired by FeatureIDE, which, despite its improved reusability [25], did not align with our specific requirements.
- We provide most of the implemented artifacts, comprising the tools and the software product line, to the research community. These artifacts are available in different repositories containing the legacy systems and the extracted software product line.[1] As we cooperated with different organizations, we cannot fully provide all details and omit implementation details for some features. Thus, we ensure that the organizations' critical information are secure while still providing almost full insights into the migration process and artifacts to the research community. In particular, we provide a complete, real-world feature model that we will continue to refine in the future.

Overall, our contributions provide insights into the extraction of software product lines from cloned variants in an industrial domain. To facilitate further analyses and allow for more detailed insights, we provide most of the artifacts to the research community. Thus, we hope to encourage the development of new approaches that support the industrial application of software product lines.

2 PHASE 1: CLONING

Initially, we developed a single application to monitor heating devices for private usage. Soon afterwards, we had several requests by other users of such systems if they could also use our solution and if we could implement slight customizations. This demand finally resulted in several organizations asking us to implement solutions based on their requirements and for their specific use cases. In this section, we describe this initial phase during which we relied on clone-and-own. To this end, we report details on two specific variants: Uvr2web and TempLog, their implementation, and practical impact for users and the organizations. We briefly sketch some *further variants* that we developed to show their commonalities and differences.

2.1 Uvr2web

Industrial Background. The Austrian company *Technische Alternative*[2] sells heating control systems (UVRs). Such control systems are widely used in private households to manage solar plants as well as wood and oil heating systems. During runtime, sensor data is logged to verify correct behavior and can be transferred via SD card or CAN bus. This data is also interesting in different application scenarios, for instance:

- To determine when to fuel a heating system with additional wood or oil;
- To evaluate whether an installed solar plant pays off; and
- To debug heating control settings in complex environments.

For these applications, a monitoring software is necessary to present the sensor data in a comprehensible manner. Based on such a monitoring solution, customers can analyze the aforementioned scenarios and make their decisions, for example, installing additional solar plants or searching for faults.

As of 2013, there was only a single open-source software for monitoring this type of heating control systems (*UVR1611 Data Logger Pro*[3]), requiring expensive dedicated hardware. By now, *Technische Alternative* has developed an official solution, the *Control Monitoring Interface* that also requires expensive hardware. Due to these limitations, we started with developing an open-source solution that does require little to no additional hardware.

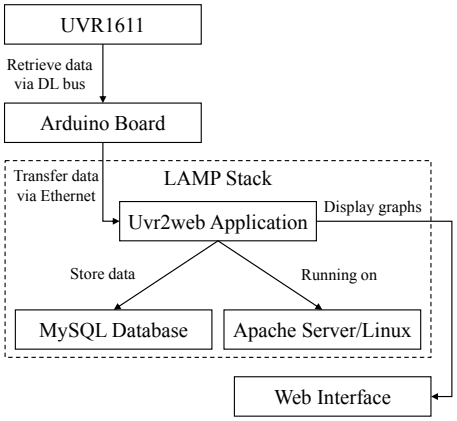


Figure 1: Structure of the application uvr2web.

Implementation. At first, we implemented a single monitoring web-application that aimed to provide a cheap solution for private users: Uvr2web[4]. This application only supports one type of control systems, the *UVR1611*. To access the sensor data, we had the options to use the CAN bus, which is also common in automotive systems, or the company's alternatively provided DL bus. We decided to use the DL bus for uvr2web, due to its simplicity facilitating the design of our application.

For retrieving the actual data, we opted for an *Arduino*-based approach [4]. Arduino provides an open platform for physical computing that comprises a micro-controller board and its accompanying software. While the hardware is relatively affordable for private households (around $20), it is still sufficient for our purpose. By further providing our web-application as open-source, our solution is cheaper and more accessible than the previous ones.

As we show in Figure 1, the Arduino board retrieves and decodes the data from the DL bus. Then, it transfers the data via Ethernet to a server. For the back-end software, we chose the popular and

[1] uvr2web: https://github.com/ekuiter/uvr2web
uvr2web SPL: https://github.com/ekuiter/uvr2web-spl/tree/master/spl/artifacts
[2] https://www.ta.co.at/
[3] https://github.com/berwinter/uvr1611
[4] https://uvr2web.de/

widely used *LAMP* stack [31], comprising a Linux operating system, Apache web server, MySQL database, and PHP programming environment. Using the LAMP stack ensures that our solution is compatible with typical web hosters. A PHP application processes the arriving sensor data and inserts it into the MySQL database. Then, users can access the data with a front-end web page that visualizes the measured data. To further improve the monitoring capabilities of our solution, we added other features, for example, to view live data, to send e-mail notifications, and to summarize the data flow. Overall, our solution comprises two main components: Firstly, the Arduino (C++) code is used to decode DL bus data and forward it to the PHP application. Secondly, the UVR2WEB application manages the database and generates sensor data graphs.

Impact. Due to its open-source availability, we have no precise data on our application's usage. However, UVR2WEB had 34 unique visitors from January 15th to January 28th 2018 and since its inception in 2013, we have been contacted by more than 20 people interested in or already using it. While running the software for around 5 years on one personal system, we had no issues.

2.2 TempLog

Industrial Background. Also in 2013, the German cooler vendor *HCP-Technology*[5] approached us with another project on temperature monitoring. This company focuses on coolers and different temperature control solutions. In this regard, they asked for similar requirements we described as useful before, namely checking the temperature and monitoring it over time (e.g., based on a graph). Up to this point, such capabilities were significantly limited.

Implementation. Again, we sketch a rough structure of the system in Figure 2. In contrast to the heating control systems, the considered coolers are equipped with sensors and Bluetooth transmitters. Such a transmitter sends the measured temperature to a nearby computer or smartphone—currently supporting Android and Windows devices. The application can be used to quickly visualize the data on the device as a graph. Optionally, the application can send the data to a back-end server, which can store and also visualize the data for online retrieval. In total, HCP TEMPLOG[6] comprises four components (cf. Figure 2):

- A HCP module with BASCOM-AVR code that decodes temperature data and transmits it via Bluetooth.
- An Android/Java application visualizes incoming temperature data on smartphones.
- A Windows (C#) application that does the same as the Android version.
- An adapted PHP component manages customers as well as the database and generates sensor data graphs.

For the last component, UVR2WEB was the starting point from which we cloned and adapted the TEMPLOG variant. Consequently, some obsolete features have been removed – for example, reading data from different sensors – while others have been added – for example, customer management and regular database cleanup. From this variant, we also created a second clone that we called DOMETIC

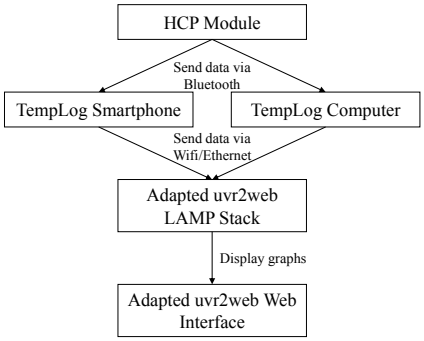


Figure 2: Structure of the application HCP TEMPLOG.

TEMPLOG. We have released this clone for the Swedish appliance manufacturer *Dometic Group*[7].

Impact. Regarding the two variants of TEMPLOG, we have more detailed insights of their usage. Firstly, they have been requested by two organizations (while UVR2WEB was initiated by us). Secondly, both variants together have been installed 70 times for Android and downloaded 250 times for Windows. Finally, 30 customers are using the available online database.

2.3 Further Variants

As mentioned, the previous variants appeared to be quite successful and to provide functionality requested by organizations and customers alike. Over time, several people contacted us and asked especially whether the UVR2WEB application was available for other heating controls than the UVR1611. At this point, we did not start with a software product line, yet—due to a lack of awareness for variability management and no obvious problems with the clone-and-own approach until then. Consequently, besides some adaptations on our side, other developers implemented variants for the *Raspberry Pi*[8] or *ESP8266*[9] by forking our repository. Usually, these variants are even more specific to the users' requirements and there has been no effort to consolidate and merge all these variants.

3 PHASE 2: TOWARDS A SOFTWARE PRODUCT LINE

Over time, we faced several issues for which we decided to switch to more systematic reuse, namely a software product line. In this section, we report our experiences regarding *encountered problems*, *reasons for a software product line*, and *barriers of extracting features* that we addressed.

3.1 Encountered Problems

The first issue we noticed after some clones had been created, was the problem of **introducing new features**. While cloning was a fast approach to get an adapted variant working, it was rather problematic in the long run. Due to the cloning, emerging variants have already multiple features in common and, thus, high potential for a software product line. Still, there was already some variability

[5] http://hcp-technology.com/
[6] http://log.hcp-technology.com/
[7] https://www.dometic.com/
[8] https://github.com/martinkropf/UVR31_RF24
[9] https://github.com/Buster01/UVR2MQTT

implemented, for example, support for different devices (i.e., desktop computer and smartphone) and optional features that can be activated at runtime (i.e., online database). Some of those features provide unique functionality potentially useful for other variants (e.g., mail notifications). Also, the TempLog variants already use simple build-system like scripts to reuse code. Nonetheless, these mechanisms were neither uniform nor managed in a systematic way, causing problems when introducing the same feature in multiple variants. For instance, we had to add glue code in each variant and analyze if any unique dependencies may cause errors.

Second, we experienced that **fixing bugs** required significantly increasing effort, as we needed to propagate updates and consider side effects of different implementation details in each variant. For example, we identified a security-related bug and fixed it in the main branches of each variant. Despite fixing parts of it in some other variants, the same bug still appeared in these, due to the diverging source code. Consequently, while sharing common code to a large extent, we had to apply fine-grained changes and extensions that made it impossible to apply the same fix in each variant. Instead, we had to identify potential side effects and customize the bug fixes for all variants individually.

Third, each of the variants had potential to be extended and distributed to a broader market, but our approach of cloning seemed to make it **impossible to advance all variants simultaneously and in a consistent way**. More precisely, uvr2web implemented a number of features using *runtime variability* and integrated these in a fitting workflow. Still, this variability was not modeled explicitly in the source code, making it challenging to extend the variants and potentially introducing unintended feature interactions. Moreover, using *version control systems* to branch versions seemed to be a good way to quickly introduce new features. However, the disadvantages of diverging branches – also including those of third-party developers – pose new challenges. For instance, an additional mail notification feature was developed in a branch that is by now 35 commits behind the master branch. At this point, it seems too costly to merge these branches. Our final approach to rely on *build systems* worked quite well for a subset of our variants with more similarities. Here, the problem appeared that the build scripts grew overly complex and that keeping the local executables in sync with the server was error-prone.

Overall, we saw that a centralized system and implementation technique to manage variability had been missing. As a result, we were quite reluctant to improve and extend the software, basically following the suggestion of *if it ain't broken don't fix it*. Still, as most likely any developer experiences at some point, this did not work for us anymore when we faced the aforementioned limitations, which called for a more systematic approach to manage our software.

3.2 Why a Software Product Line?

To tackle the described problems, we thought that using a variability-specific implementation technique to consolidate the variants may be the best course of action. Thus, we investigated the extraction from our legacy variants towards a software product line in more detail. As we faced concrete problems, we did not perform a potential analysis [20], but defined concrete goals we wanted to fulfill. Finally, we aimed to extract features from our variants to:

- Remove code duplication and, thus, facilitate maintenance and extensibility;
- Derive a feature model as ground truth for the variability in our applications;
- Avoid branching by using an implementation technique that supports a variability mechanism;
- Improve our tooling to further automate the build process for our variants; and
- Extend and simplify our capabilities to adopt variants to different environments.

Fulfilling these goals would allow us to facilitate the development of new variants and their adoption to user-specific requirements, for instance, to support other kinds of temperature control units, motherboards (e.g., Arduino, Raspberry Pi), and back-end servers or components (e.g., Apache server, Emoncms[10], MySQL).

3.3 Barriers of Feature Extraction

Considering the extraction of features, we faced some problems, which we are summarizing in the following.

Deciding which behavior comprises a feature is a well-known challenge in software-product-line engineering that we faced, too. This problem arises, as developers have different notions of features [8, 12, 29, 30], which challenges the introduction of a uniform understanding of a system and hampers feature location tasks. Nonetheless, it is necessary to solve this problem to decide which features to extract from the variants. We addressed this by letting the main developer of the variants decide how to decompose these legacy systems into features.

There was a mismatch between the intended and actual variability that we could implement by only extracting features [24, 32, 45]. We faced this problem, due to several components requiring specific settings and, thus, imposing restrictions we did not intend at the beginning. Thus, we needed to decide for such features if we wanted to further restrict the variability or apply refactorings to achieve the intended variability. This was decided on a case-by-case basis, depending on the complexity of the required refactoring and the gained variability.

Extracting variable feature code was the main challenge of migrating the variants towards a software product line. At first, we had to decide which variability mechanism we wanted to use for C++ and PHP, the two primary languages of the applications. We decided to rely on the C preprocessor [21] for C++ and a combination of *runtime variability*, *plug-ins*, and a *build system* for PHP. Additionally, we had to refactor several code parts to make them reusable as features and resolve code smells [19]. However, here we sometimes had to decide which code should be refactored – potentially risking the introduction of new bugs – and which should be kept the way it was – potentially missing variability. In most cases, we kept the existing code where variability was of no concern, only fixing obvious bugs, because there was no systematic unit or integration testing in place. However, for some key features regarding variability, larger refactorings had to be performed carefully (e.g., to support different heating controls).

Initializing configuration management and product derivation was mainly concerned with deciding for the tooling we wanted

[10] https://emoncms.org/

to install. These tools should support a suitable configuration menu and fully automatically provide the configured variant. Here, we found that existing tools [10, 27, 34] – while supporting these functionalities greatly – had some limitations that restricted their usage for our use case. Thus, we decided to implement our own tooling that also allows us to rely on different programming languages and variability mechanisms at the same time. In Section 5, we further discuss the reasons for this decision and briefly describe the tools that we developed for the extraction process.

Managing the source code was a problem we were not aware of beforehand, but that quickly arose when our cooperating organizations asked for variants. In particular, our software product line suddenly comprised open-source as well as closed-source features. Consequently, we had to enforce a privacy policy for the extracted features. To do this, we identified all features and code artifacts which are private to the organizations. Due to the nature of our tool, it is easy to separate these features into a private repository. Still, the feature model is entirely open-source, because it does not contain sensitive information.

4 PHASE 3: DESIGNING A FEATURE MODEL

Within this section, we describe the design of an appropriate feature model that covers all existing variants, but also allows for new combinations of features—increasing the possible configuration space significantly. In particular, the feature model must ensure that variants can only be instantiated if they fulfill the requirements of the underlying hardware and must forbid unintended feature interactions. Considering our extracted software product line, we have to account for multiple different device types for capturing (e.g., HCP module and DL bus) and representing (e.g., smartphones and web interfaces) data, as well as varying data visualizations.

We display the resulting feature model in Figure 3. For the feature model's hierarchy structure, we decided to use a functional separation of concerns. Thus, below the root feature *uvr2web*, we divided the feature model into three mandatory subtrees, *data capture*, *data transfer*, and *data visualization*. All features within the data-capture subtree are related to retrieving (i.e., capturing) data from a device. The data-transfer subtree refers to all features that are related to physically transferring the captured data within the system. Lastly, the features within the data-visualization subtree are related to processing, storing, and visualizing the captured data. As these subtrees already indicate, the features in our model reflect the sequential data flow of the variants, as we depict in Figure 1 and Figure 2. This facilitates the configuration process, as the same order appears there and follows the user-visible data-flow.

As the principle *tyranny of the dominant decomposition* indicates, we cannot represent all dependencies between features within a feature tree [44]. For instance, our focus on a functional tree hierarchy makes it harder to express that some existing software artifacts, such as, the Android and Windows applications, only work under specific circumstances (e.g., when data is transferred via Bluetooth). To represent these dependencies within our feature model, we additionally introduced cross-tree constraints to scope the number of valid variants to those that are actually useful. Alternatively, we could have refactored the existing artifacts to make them more general. However, because of the very specific nature of the local visualization by the Android and Windows applications, a refactoring is beyond the scope of this paper and also not needed for the canonical use case (i.e., logging UVR sensor data).

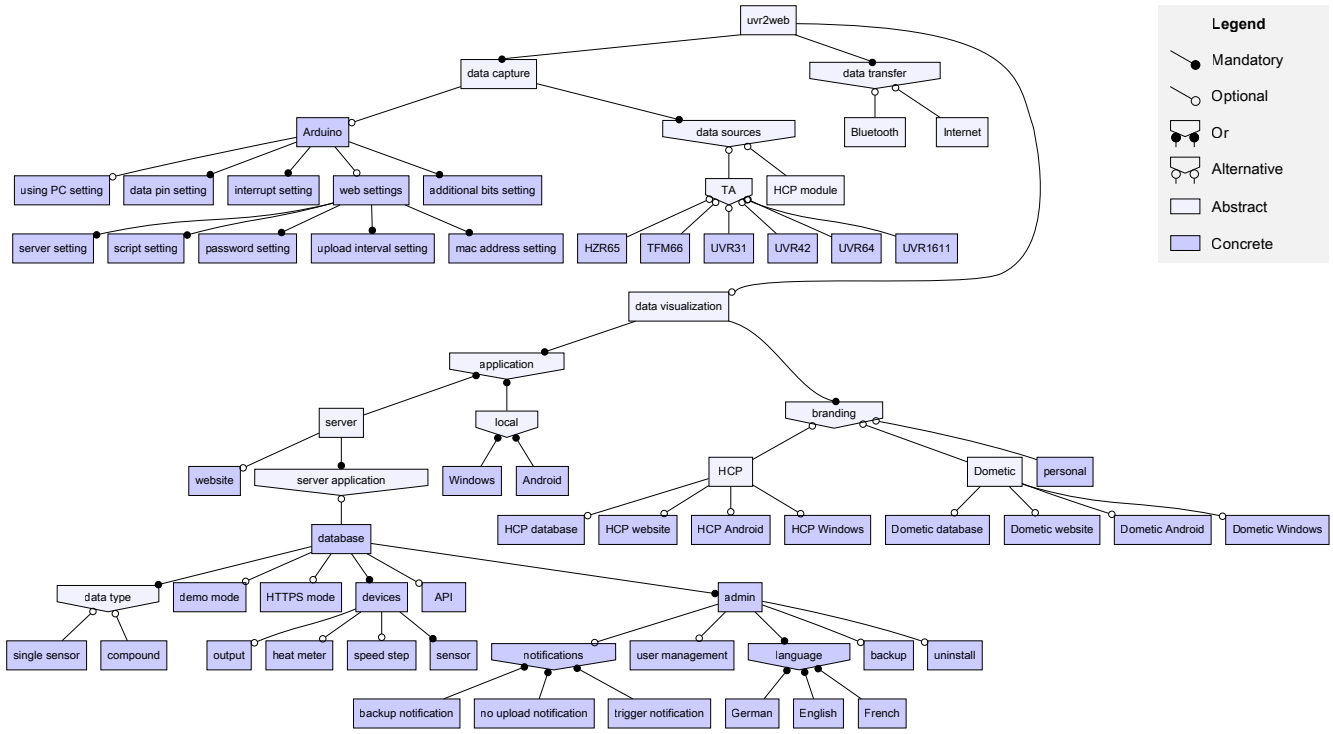
In the following we describe each of the relevant subtrees – *data capture*, *data transfer*, and *data visualization* – in more detail. Precisely, we reason about their necessity and provide some implementation details. Finally, we describe the *branding* and *database* features as part of the data visualization subtree – which are critical for the cooperating companies – and explain why specific *cross-tree constraints* have been introduced.

Data Capture. Within the subtree *data capture*, we support two different types of devices that are used within different variants. First, variants supporting devices from the UVR product line by *Technische Alternative (TA)* use the features within the subtree *TA*. The child features below *TA* implement a variety of different data types (e.g., sensors, outputs, heat meters, and speed steps) and are represented as an alternative group within the feature tree (i.e., exactly one device must be chosen). Note that of the UVR controls present in the feature model, only the *UVR1611* feature is actually implemented in the original *uvr2web* variant (cf. Figure 1). The other controls have been implemented after migrating towards the software product line.

Second, the feature *HCP module* is used by variants supporting Bluetooth devices from *HCP-Technology*. This affects the use case for *HCP-Technology* and *Dometic Group*, as we describe in Section 2.2. The feature *HCP module* exposes exactly one temperature value and, thus, has no other features below itself. However, only a single temperature value is captured in the corresponding use cases—which is already implemented in other features (i.e., *Bluetooth* and *single sensor*) that are required, due to cross-tree constraints. Consequently, *HCP module* comprises no implementation and is abstract, only facilitating the configuration process and avoiding faulty variants.

The *Arduino* feature provides the base code for capturing UVR control data. Still, there are some functionalities which may need to be customized according to the user's needs, such as, the server to upload sensor data or a password to secure the access. These functionalities are added as mandatory and optional features, depending on their necessity. Some of the setting features are mandatory, meaning that they do not provide additional variability on the surface, but, in fact, they do broaden the number of possible variants: They hold a string value that configures their behavior, instead of being only selected or not—resembling attributed features in extended feature models [2, 7]. These features allow the developer to pass further configuration options. As examples, we can specify the data pin that connects the Arduino board with the UVR control and define the connection to the web server to which the data is uploaded. In contrast to the traditional toggling of Boolean features, this allows us to reduce the size and complexity of the feature model without losing too much information on first glance.

Data Transfer. We introduced the subtree *data transfer* and both of its child features, *Bluetooth* and *Internet*, to further structure the software product line. The subtree mainly serves two purposes: Simplifying cross-tree constraints related to Bluetooth devices and helping users to understand how their decisions in the other two subtrees influence the data transfer within the system. Currently, there is no implementation for these transfer channels, as we rely



$$HCP \wedge database \Rightarrow single\ sensor \qquad TA \wedge database \Rightarrow (compound \wedge sensor \wedge output)$$
$$TA \Rightarrow Internet \qquad single\ sensor \Rightarrow (sensor \wedge \neg output \wedge \neg heat\ meter \wedge \neg speedstep)$$
$$UVR1611 \wedge database \Rightarrow (heat\ meter \wedge speedstep) \qquad TA \Leftrightarrow Arduino$$
$$uninstall \Rightarrow backup \qquad backup\ notification \Rightarrow backup$$
$$local \Rightarrow HCP\ module \qquad HCP\ module \Leftrightarrow (HCP \vee Dometic)$$
$$database \Leftrightarrow (personal \vee HCP\ database \vee Dometic\ database) \qquad Android \Leftrightarrow (HCP\ Android \vee Dometic\ Android)$$
$$Windows \Leftrightarrow (HCP\ Windows \vee Dometic\ Windows) \qquad website \Leftrightarrow (HCP\ website \vee Dometic\ website)$$
$$website \Rightarrow (HCP\ module \wedge database \wedge Android \wedge Windows)$$
$$HCP\ module \Rightarrow (Bluetooth \wedge \neg demo\ mode \wedge \neg user\ management \wedge \neg backup \wedge \neg uninstall \wedge \neg API)$$
$$(UVR31 \vee UVR42 \vee UVR64 \vee HZR65 \vee TFM66) \wedge database \Rightarrow (\neg heat\ meter \wedge \neg speedstep)$$

Figure 3: Feature diagram of the software product line UVR2WEB.

on them through the hardware and their APIs. Consequently, we do not require implementation for these features for now and, thus, all of them are abstract. The resulting structure facilitates further evolution of the software product line to provide more options for data transfer (e.g., radio transmission).

Data Visualization. Each variant of the software product line supports one or both of the currently existing two visualization approaches. First, *local* visualization refers to applications that retrieve sensor data via Bluetooth. Their job is to provide a first impression of the data (e.g., drawing a graph on a smartphone) and, optionally, forward it to a server application. The user may select the Android application as well as the Windows application, both provide the same insights into the captured data. In contrast to the feature combination of *Arduino* and *database* (the UVR2WEB use case in Section 2), the local visualization cannot be customized further, because it is only used within *HCP-Technology*'s use case (cf. page 3). Thus, the customizations for visualizations of these variants are defined by the company.

Second, *server* visualization refers to applications running on the server that retrieves sensor data via Internet. The original *uvr2web* variant already provides a complete, but complex, database application. In contrast to local visualization applications, the database has to be customizable to a great extent. This is because the database may be part of every variant (i.e., for every supported UVR control as well as the single-sensor Bluetooth device)—each requiring customizations to the used data formats. Similar to the additional UVR controls, an evolution of the software product line may support other server visualization applications as well, for example, Emoncms or openHAB. These may not be as tailored as the included database application, but they are very well-suited and popular for smart-home monitoring and home automation. Finally, for *HCP-Technology*'s use case, the *website* feature comprises an installation of the database for each customer as well as a web

portal. For personal use, this is not needed and we therefore disable this combination with cross-tree constraints.

Branding. To clarify the corporate branding of a variant, we added an additional *branding* feature. This way, we are able to customize and distinguish variants generated for *HCP-Technology*, *Dometic Group*, and personal use. As these are the most critical features of the software product line, we omit their parts of the implementation in the repository, but will explain some details in the following. In particular, we describe how we achieve the branding of different features and illustrate the problems we faced with feature interactions and granularity, which we resolved by combining design-time (i.e., C preprocessor, build system) and runtime (i.e., plug-ins) variability mechanisms.

Regarding branding, we encountered a problem: The features *Windows*, *Android*, *website*, and *database* need to be branded individually, meaning that for each of these features there are some artifacts specific to *HCP-Technology*, *Dometic Group*, and other customers. Which of these artifacts we have to include depends on the selected branding. Thus, only when both, *Windows* and *HCP*, are selected, the HCP-specific Windows artifacts are delivered, otherwise they are not. This requirement gives rise to a number of feature interactions and glue-code between the aforementioned features and the features *HCP* and *Dometic*. We decided to resolve these feature interactions by extracting them into their own features (i.e., *HCP database*, *HCP website*, and so on). These features are added below *HCP* and *Dometic* and connected through cross-tree constraints to the platform features (e.g., *Windows*).

Database. At last, the *database* feature defines the primary visualization software for the captured data. Depending on the device in the *data capture* subtree, either *compound* or *single sensor* are selected automatically to enable some use-case specific adjustments inside the database. The *demo mode* and *HTTPS mode* features implement minor and fine-grained adjustments of the database using runtime variability, whereas the *API* feature – a larger feature implementing a JSON-based API – makes use of a build system for its coarse-grained refinements.

Just as the *data type* subtree, the *devices* subtree is directly tied to the selected device features. Every device exposes at least one sensor, so *sensor* is mandatory, while the other features in this subtree depend on the device. We implemented these features by using a combination of a build system with a plug-in loader at runtime—with the build system deciding which artifacts to copy and, at runtime, the database loading all available artifacts.

The *admin* subtree contains various optional features for administrative usage, the only exception being the *language* feature. This feature determines which languages can be chosen at runtime. To guarantee that at least one language is available, *language* is an or-group. Similarly, the *notifications* subtree is an or-group that determines the e-mail notifications selectable at runtime. The *backup notification* requires the *backup* feature to be selected, which is modeled with a cross-tree constraint. We employ a similar implementation strategy as before, using a plug-in loader. Extracting all these optional features in the *admin* subtree was necessary because the *HCP-Technology* use case limits the access to administrative functions. Consequently, features like *backup* and *uninstall* are not available in these configurations.

Cross-Tree Constraints. Most cross-tree constraints originate from the legacy use cases and their separated hardware and software solutions (cf. Section 2). Consequently, the constraints resemble a separation between the corresponding features, but also map dependencies within the legacy systems. Considering the legacy *uvr2web* variant, the approach for capturing data is by connecting an Arduino board to the UVR control and to the Internet. Thus, selecting an UVR control feature automatically selects the *Arduino* and *Internet* features in the extracted software product line.

In contrast, the *HCP-Technology* hardware requires a Bluetooth connection and only exposes a single sensor value. Thus, the *Bluetooth* and *single sensor* features have to be selected for HCP. Because the Bluetooth device is already flashed with a fixed software, no further variability is needed for the device and the *HCP module* feature is abstract. Moreover, the *HCP module* and *Arduino* features are mutually exclusive, which currently does not allow merging the two legacy variants at this point.

5 PHASE 4: DEVELOPING THE TOOLING

Within this section, we explain why we implemented new tools for managing our software product line. Afterwards, we introduce our tools and their concepts in more detail.

While considering existing tools for managing software product lines [10, 27, 34], we had some additional requirements: For instance, because UVR2WEB is mostly an open-source project anyone can participate in, its tooling should also be free and open-source to facilitate contribution. This rules out pure::variants and Gears because they are commercial products.

Additionally, for encouraging contribution, setting up the tooling should be easy and familiar for new developers (i.e., running a single command). Finally, our goal was to set up a simple web environment where end-users may configure and generate a product tailored to their needs. Using this environment should not require any understanding of software-product-line engineering, making it more user-friendly than other tools. While FeatureIDE is open-source, it is also tightly coupled to the Eclipse platform and does not provide a web interface yet, which was our main reason for not using FeatureIDE. Thus, we decided to implement new tooling that specifically addresses the above requirements.

In order to manage our software product line, we provide several specialized tools for certain tasks such as *visualization*, *configuration*, and *analysis*. We describe our tools based on their role in the software-product-line engineering process [2, 36] that we depict in Figure 4. Here, the activities we support with our tools are framed with dashed boxes. Each name within a box refers to one tool that we explain in the following.

As one of our main goals was to make the software product line easily accessible to its potential users, we decided to (1) use a common format for feature models (i.e., the XML format of the open-source tool FeatureIDE) and (2) develop a web-based API for each tool. This further facilitates accessibility by:

- Making tooling and source code of a software product line available via regular Internet connection;
- Requiring no additional installations or advanced knowledge about software product lines; and
- Supporting multiple kinds of devices.

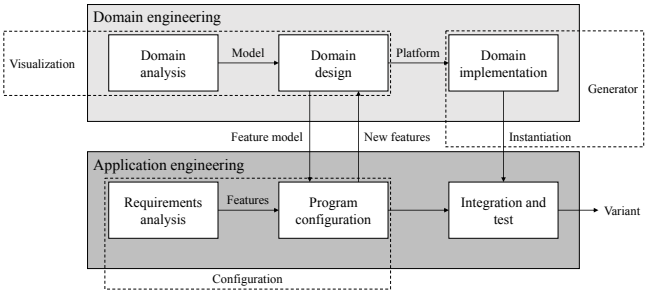


Figure 4: Software-product-line engineering activities (based on Krüger et al. [28]). Activities supported by our tools are framed with dashed boxes.

Domain Analysis. Specifying the relevant features of a software product line and defining their dependencies in a feature model is usually one of the first steps in software-product-line engineering. While fully-fledged IDEs, such as FeatureIDE, provide a powerful feature model editor, we focus only on lightweight feature model visualizations inside a web browser to:

(1) Quickly assess a feature model without installing additional and unnecessary tools;
(2) Discuss and collaborate on the feature model's design; and
(3) Display a configuration of selected and deselected features.

Our visualization tool[11] takes a feature model as input and displays it as a graph (cf. Figure 3). While it currently only shows feature models (i.e., read-only), we see potential for collaborative editing (e.g., enabling domain experts to simultaneously work on the same feature model over the web). As mentioned, the tool is completely web-based and can be accessed via most modern browsers. The implementation uses the *GraphViz* library to generate an SVG image, which is embedded into the web document.

Requirements Analysis. Configuring a software product line (i.e., selecting and deselecting features to specify a variant) is a central process in software-product-line engineering. We support this process with our web-based configuration tool[12] that:

(1) Provides an intuitive, lightweight user-interface for selecting features defined in the provided model; and
(2) Applies decision propagation to automatically select implied features according to the underlying feature model.

Our configuration tool is based on JavaScript and provides a user interface based on *tri-state checkboxes* (i.e., selected, deselected, and indeterminate states). In Figure 5, we show the user interface, which provides a tree-structured view corresponding to the feature model. For decision propagation, it uses a satisfiability solver and the algorithm outlined by Apel et al. [2].

Domain Implementation & Product Derivation. Deriving a variant from a valid configuration and the corresponding software artifacts is the main step in software-product-line engineering. For us, using established tools such as FeatureHouse or AHEAD [5] was not an option, as due to technical limitations it is not possible to

[11] https://github.com/ekuiter/feature-model-viz
[12] https://github.com/ekuiter/feature-configurator

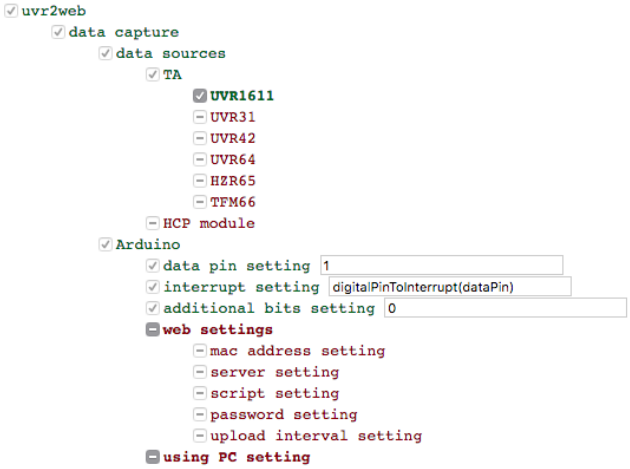


Figure 5: Web-based configuration interface.

run these tools on shared hosting servers. In addition, our server infrastructure regarding UVR2WEB and HCP-TECHNOLOGY has limited capabilities. Thus, we developed a lightweight generator tool with modest server requirements serving our needs. Our generator tool[13] can be used to:

(1) Analyze and validate feature models and configurations;
(2) Implement and trace features using various variability mechanisms at the same time; and
(3) Generate products and export them.

In our UVR2WEB software product line, features are implemented with different variability mechanisms, mainly *runtime variability*, a *build system*, and a *preprocessor*. Thus, our generator tool supports all of these variability mechanisms in parallel. Additionally, it provides experimental support for feature-oriented programming [37] and aspect-oriented programming [22]. Similar to our other tools, the generator tool is web-based. The implementation is based on PHP and requires a PHP environment, which should be available on most shared hosting services.

Summary. Though we developed the tools with our particular use case in mind, we focused on accessibility and generalizability in the tool's APIs. Thus, we claim that each of them can be used independently and for arbitrary feature models and software product lines. For example, our configuration interface may be coupled with a dedicated build server running established software product line tooling. This way, existing software product lines can become *web-ready* without much additional development effort.

6 PHASE 5: THE SOFTWARE PRODUCT LINE

Finally, we implemented the features we show in the feature model in Figure 3 with the described tooling. Most of our source code is open source and available on GitHub[1], excluding few features that are critical for the companies we cooperated with. In this section, we discuss the benefits of the extracted variability from an implementation perspective.

[13] https://github.com/ekuiter/feature-php

Table 1: Code metrics before feature extraction.

Variant	SLOC per component				SLOC
	C++	Android	Windows	PHP	
uvr2web	736	0	0	4,412	5,148
TempLog	0	1,662	1,563	3,612	6,837
				Total	11,985

While reviewing the variability we modeled in the feature model, we see an expected general division into two types of valid variants: First, those that monitor UVR controls (including *TA*, *Arduino*, *compound*, and *Internet*). Such variants closely resemble the original UVR2WEB variant. Second, those for monitoring Bluetooth devices (including *HCP Module*, *single sensor*, *Bluetooth*, and *local*). These variants correspond to the TEMPLOG use cases. We aimed to have such a clear separation of variability that resembles the previous variants to facilitate the migration process by allowing us to focus on variants that are demanded by the companies. Thus, we introduced several cross-tree constraints that may be superfluous, but would require an unfeasible amount of additional effort to resolve. Consequently, we would require more time for development and testing, slowing down time-to-market. Nonetheless, instead of having maybe a dozen cloned variants, we can now instantiate 4,017 variants, a significant increase in diversity but also complexity.

As we are just in the beginning of using this software product line, four questions arise for us:

(1) Is all of the modeled variability useful?
(2) Can all modeled feature interactions be instantiated?
(3) Should the software product line be split to reflect the origin and use cases of features?
(4) Will the software product line be beneficial in the future?

Usefulness of Variability. Concerning the first question, we argue that the introduced variability is useful, despite increasing the complexity for three reasons. First, as described, we added constraints to facilitate the feature extraction which may be removed in the future to allow for new variants. The main benefit in our eyes is that we may reduce the variability and number of variants, but we resemble the original systems more closely. As these are currently in our and the companies' focus, this is a clear advantage over allowing all potential variability.

Second, the database subtree increases the variability quite a lot and could, to some extent, simply be represented as a single database feature. However, this feature and its children are used by all variants to store data, requiring some adaptations in the stored data types, APIs, and user management. Consequently, we decided to add the full subtree, even if the variability is increased, to model more fine-grained features we have implemented and aim to introduce—and for which a single feature is too coarse.

Finally, the main purpose of introducing a software product line was to unify the code-base and provide a simple configuration process, not a large solution space. In our assessment, we achieved this by limiting the variability with constraints to resemble the legacy systems. By now, these have a more unified code and we can use our software product line to instantiate them.

Table 2: Code metrics after feature extraction.

Feature	SLOC
Windows	1,533
Android	1,478
website	1,013
database	890
devices	659
Arduino	623
Other features	4,388
Total	10,584

Feature Interactions. Feature interactions are a well-known issue in software-product-line engineering, especially for testing [2, 6, 11, 38, 46]. Concerning the extraction of features from legacy systems, we experienced some issues that require a more detailed analysis from our side. In particular, we are currently unaware whether there exist any feature interactions that may change the behavior of any variants in an unintended way. Furthermore, testing all possible variants and feature interactions is unfeasible for us, due to our focus on a subset of the available solution space. Thus, we may encounter problems with valid configurations of our software product line, due to feature interactions we are unaware of. Our current testing strategy consists of testing each variant on its own when we instantiate it. However, we plan to run larger tests on the whole software product line in the future.

Splitting the Software Product Line. After creating an initial version of the feature model, we thought about dividing the software product line into a multi product line [39, 40]. The main reason for this are the numerous cross-tree constraints and the architecture of the model—already structuring it into subtrees for *capturing*, *transferring*, and *visualizing* data. In addition, the database subtree is rather independent of the remaining features. Thus, it seemed that dividing our software product line into a multi product line could be a good approach.

Despite such reasons, we decided to keep and improve the feature model, mainly for two reasons: First, we would not be able to use the implemented configuration process without further significant adaptations. Second, splitting the features would re-introduce some of the problems we aimed to resolve by moving from clone-and-own to a software product line, for example: No unified model, code duplication, complicated configuration, and nonuniform evolution. Thus, we decided that keeping all features in a single model and code-base – improving the structure and scoping in the future – would be the best course of action.

Future Benefits. Because of the now unified code-base, there are several perspectives for evolving the software product line:

- Already, we implemented several additional heating controls other than the original UVR1611 control. Using our tooling, we can provide a simplified configuration interface for quickly instantiating predefined variants. More heating controls are planned and simple to implement, due to the new software-product-line approach.

- For the UVR2WEB use case, we plan to implement additional server application, such as Emoncms, to improve smart home integration. This is only a matter of adding features to the *server application* subtree in the feature model. Previously, we would have had to create additional cloned variants.
- *HCP-Technology* plans to introduce a new way to transfer data from the *HCP module* to the online database, not involving Bluetooth. Using the extracted software product line, this may be achieved by adjusting some cross-tree constraints and a few implementation details.

Due to the already described benefits we aimed for and have achieved, these extensions should be easier to implement than in the legacy clone-and-own approach. Besides facilitated processes for updating, adding, and fixing features, we also experienced that managing variability has become far easier. In particular, we do not need to implement every feature multiple times for each variant and identify which may be the best to start with for a new customer. Thus, despite the necessity for more glue-code, the software product line requires approximately 88.3% of the legacy systems source code. We show the corresponding values before and after extraction in Table 1 and Table 2, respectively. Overall, we see several benefits of the software product line approach for our variants, especially concerning its future evolution.

7 RELATED WORK

Extractive adoption of software product lines is the most common approach in practice [9, 36, 43]. Consequently, there has been a lot of research on this topic, for instance, on feature location approaches for extracting features [3, 30]. Martinez et al. [33] collect case studies with different scopes, settings, and approaches, providing a detailed overview on scientific and practice reports on similar works. In the following, we discuss a set of papers we are aware of and that describe industrial pros and cons of migrating towards a software product line or the process itself.

Dubinsky et al. [16] empirically investigate how cloning of software is used in industrial settings. To this end, they analyze six software families that are based on different cloning granularities, reaching from single methods to complete products. Some of the pros and cons reported in the interviews resemble those that we describe, namely, time and cost savings of cloning, problematic change propagation, and issues with introducing new adaptations. However, this study has been conducted in the context of already used software product lines. Our development and report differs greatly in several details: We provide additional insights into the reasoning of using a sole clone-and-own approach and the alter extraction process. Thus, we describe also other issues from a different perspective, complementing this empirical study. Furthermore, we contribute most artifacts of our industrial systems.

Yoshimura et al. [48] describe a case study with a company that aimed to merge multiple clones into a software product line. Here, the focus is on assessing the potential for this extraction based on code-clone detection [41]. While the focus of this work is on predicting costs and benefits, several discussed steps and tasks are similar or even identical to those we performed. Still, we provide detailed insights into the actual process and, thus, complement such case studies that solely focus on estimating costs in advance.

Couto et al. [14] extract a preprocessor-based software product line of the ArgoUML system and provide it as open-source. While this closely resembles our case study, it also differs heavily in multiple points. First, ArgoUML may be larger than our systems, but did not originate from industry. Second, ArgoUML has been a single product with multiple components and not a set of clones that share, but also differ, in their features. Third, the feature model is small and not nearly as complex as ours. Finally, we discuss the actual extraction process in detail, including pros and cons. Thus, we argue that ArgoUML is a valid and well-designed case study, but our contributions complement this significantly by providing a real-world software product line.

Ebert and Smouts [17] report their experiences with extracting software product lines from existing variants at a company. As for aforementioned works, some of the reported pros and cons are similar to ours. However, the focus of this paper lies more on business aspects and management decisions to sell and coordinate the software product line. This has not been our focus, as we only distribute variants to customers on their request and have not a large development team. While we focus less on this part, we describe the actual extraction process in great detail, which is out of scope of the work of Ebert and Smouts [17].

Alves et al. [1] describe a case study during which they combined the extractive and reactive approach to adopt an aspect-oriented software product line. They explain the corresponding refactorings and their method in great detail. In contrast to our work, the authors do not describe the pros and cons of extracting a software product line for their use case. Additionally, it seems unclear if the used subject systems have been developed in an industrial context and if the artifacts are available. Thus, our work significantly differs from this one by Alves et al. [1].

8 CONCLUSION

In this paper, we described a migration process from cloned systems towards a software product line in an industrial setting. We provided a holistic view, starting from the development of clones, over the migration process, to software-product-line engineering and also include tooling and reasoning for each step. Furthermore, we contribute almost all details of the study and tools as open-source artifacts to the research community. Additionally, we discussed pros and cons of the different steps we performed and approaches we used, showing some potential for improvement in the future.

For future work, we will continue to develop the software product line described in this paper as well as our tooling. The open-source nature of our artifacts allows other researchers to also analyze the systems and to potentially support or extend our work. A particular focus for us is the improvement of the feature model and the extracted features. This poses the chance to investigate necessary refactorings, testing, evolution, and quality assurance of the extractive approach in detail.

ACKNOWLEDGMENTS

This work is supported by the German Research Foundation (DFG) under grants LE 3382/2-1, LE 3382/3-1, and SA 465/49-1, and Volkswagen Financial Services AG.

REFERENCES

[1] Vander Alves, Pedro Matos, Leonardo Cole, Paulo Borba, and Geber Ramalho. 2005. Extracting and Evolving Mobile Games Product Lines. In *International Systems and Software Product Line Conference (SPLC)*. Springer, 70–81.

[2] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. 2013. *Feature-Oriented Software Product Lines*. Springer.

[3] Wesley Klewerton Guez Assunção and Silvia Regina Vergilio. 2014. Feature Location for Software Product Line Migration: A Mapping Study. In *International Systems and Software Product Line Conference (SPLC)*. ACM, 52–59.

[4] Massimo Banzi and Michael Shiloh. 2014. *Getting Started with Arduino: The Open Source Electronics Prototyping Platform*. Maker Media.

[5] Don Batory. 2004. Feature-Oriented Programming and the AHEAD Tool Suite. In *International Conference on Software Engineering (ICSE)*. IEEE, 702–703.

[6] Don Batory, Peter Höfner, Bernhard Möller, and Andreas Zeland. 2013. Features, Modularity, and Variation Points. In *International Workshop on Feature-Oriented Software Development (FOSD)*. ACM, 9–16.

[7] David Benavides, Sergio Segura, and Antonio Ruiz-Cortés. 2010. Automated Analysis of Feature Models 20 Years Later: A Literature Review. *Information Systems* 35, 6 (2010), 615–636.

[8] Thorsten Berger, Daniela Lettner, Julia Rubin, Paul Grünbacher, Adeline Silva, Martin Becker, Marsha Chechik, and Krzysztof Czarnecki. 2015. What is a Feature? A Qualitative Study of Features in Industrial Software Product Lines. In *International Conference on Software Product Line (SPLC)*. ACM, 16–25.

[9] Thorsten Berger, Ralf Rublack, Divya Nair, Joanne M. Atlee, Martin Becker, Krzysztof Czarnecki, and Andrzej Wąsowski. 2013. A Survey of Variability Modeling in Industrial Practice. In *International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS)*. ACM, 1–8.

[10] Danilo Beuche. 2008. Modeling and Building Software Product Lines with Pure::Variants. In *International Systems and Software Product Line Conference (SPLC)*. IEEE, 358–358.

[11] Muffy Calder, Mario Kolberg, Evan H Magill, and Stephan Reiff-Marganiec. 2003. Feature Interaction: A Critical Review and Considered Forecast. *Computer Networks* 41, 1 (2003), 115–141.

[12] Andreas Classen, Patrick Heymans, and Pierre-yves Schobbens. 2008. What's in a Feature: A Requirements Engineering Perspective. In *International Conference on Fundamental Approaches to Software Engineering (FASE)*. Springer, 16–30.

[13] Paul C. Clements and Charles W. Krueger. 2002. Point/Counterpoint: Being Proactive Pays Off/Eliminating the Adoption Barrier. *IEEE Software* 19, 4 (2002), 28–30.

[14] Marcus Vinicius Couto, Marco Tulio Valente, and Eduardo Figueiredo. 2011. Extracting Software Product Lines: A Case Study Using Conditional Compilation. In *European Conference on Software Maintenance and Reengineering (CSMR)*. IEEE, 191–200.

[15] Krzysztof Czarnecki, Paul Grünbacher, Rick Rabiser, Klaus Schmid, and Andrzej Wąsowski. 2012. Cool Features and Tough Decisions: A Comparison of Variability Modeling Approaches. In *International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS)*. ACM, 173–182.

[16] Yael Dubinsky, Julia Rubin, Thorsten Berger, Slawomir Duszynski, Martin Becker, and Krzysztof Czarnecki. 2013. An Exploratory Study of Cloning in Industrial Software Product Lines. In *European Conference on Software Maintenance and Reengineering (CSMR)*. IEEE, 25–34.

[17] Christof Ebert and Michel Smouts. 2003. Tricks and Traps of Initiating a Product Line Concept in Existing Products. In *International Conference on Software Engineering (ICSE)*. IEEE, 520–525.

[18] Stefan Fischer, Lukas Linsbauer, Roberto Erick Lopez-Herrejon, and Alexander Egyed. 2014. Enhancing Clone-and-Own with Systematic Reuse for Developing Software Variants. In *International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 391–400.

[19] Martin Fowler and Kent Beck. 1999. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley.

[20] Claudia Fritsch and Ralf Hahn. 2004. Product Line Potential Analysis. In *International Systems and Software Product Line Conference (SPLC)*. Springer, 228–237.

[21] Brian W. Kernighan and Dennis M. Ritchie. 1978. *The C Programming Language*. Prentice-Hall.

[22] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. 1997. Aspect-Oriented Programming. In *European Conference on Object-Oriented Programming (ECOOP)*. Springer, 220–242.

[23] Peter Knauber, Jesus Bermejo, Günter Böckle, Julio Cesar Sampaio Do Prado Leite, Frank van der Linden, Linda M. Northrop, Michael Stark, and David M. Weiss. 2002. Quantifying Product Line Benefits. In *International Workshop on Product-Family Engineering (PFE)*. Springer, 155–163.

[24] Sebastian Krieter, Jacob Krüger, and Thomas Leich. 2018. Don't Worry About it: Managing Variability On-The-Fly. In *International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS)*. ACM, 19–26.

[25] Sebastian Krieter, Marcus Pinnecke, Jacob Krüger, Joshua Sprey, Christopher Sontag, Thomas Thüm, Thomas Leich, and Gunter Saake. 2017. FeatureIDE: Empowering Third-Party Developers. In *International Systems and Software Product Line Conference (SPLC)*. ACM, 42–45.

[26] Charles W. Krueger. 2002. Easing the Transition to Software Mass Customization. In *International Workshop on Software Product-Family Engineering (PFE)*. Springer, 282–293.

[27] Charles W. Krueger. 2007. BigLever Software Gears and the 3-Tiered SPL Methodology. In *ACM SIGPLAN Conference on Object Oriented Programming Systems, Languages, and Applications (OOPSLA)*. ACM, 844–845.

[28] Jacob Krüger, Wolfram Fenske, Jens Meinicke, Thomas Leich, and Gunter Saake. 2016. Extracting Software Product Lines: A Cost Estimation Perspective. In *International Systems and Software Product Line Conference (SPLC)*. ACM, 354–361.

[29] Jacob Krüger, Wanzi Gu, Hui Shen, Mukelabai Mukelabai, Regina Hebig, and Thorsten Berger. 2018. Towards a Better Understanding of Software Features and Their Characteristics: A Case Study of Marlin. In *International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS)*. ACM, 105–112.

[30] Jacob Krüger, Louis Nell, Wolfram Fenske, Gunter Saake, and Thomas Leich. 2017. Finding Lost Features in Cloned Systems. In *International Systems and Software Product Line Conference (SPLC)*. ACM, 65–72.

[31] George Lawton. 2005. LAMP Lights Enterprise Development Efforts. *Computer* 38, 9 (2005), 18–20.

[32] Duc Minh Le, Hyesun Lee, Kyo Chul Kang, and Lee Keun. 2013. Validating Consistency Between a Feature Model and Its Implementation. In *International Conference on Software Reuse (ICSR)*. Springer, 1–16.

[33] Jabier Martinez, Wesley K. G. Assunção, and Tewfik Ziadi. 2017. ESPLA: A Catalog of Extractive SPL Adoption Case Studies. In *International Systems and Software Product Line Conference (SPLC)*. ACM, 38–41.

[34] Jens Meinicke, Thomas Thüm, Reimar Schröter, Fabian Benduhn, Thomas Leich, and Gunter Saake. 2017. *Mastering Software Variability with FeatureIDE*. Springer.

[35] Tristan Pfofe, Thomas Thüm, Sandro Schulze, Wolfram Fenske, and Ina Schaefer. 2016. Synchronizing Software Variants with VariantSync. In *International Systems and Software Product Line Conference (SPLC)*. ACM, 329–332.

[36] Klaus Pohl, Günter Böckle, and Frank van der Linden. 2005. *Software Product Line Engineering: Foundations, Principles, and Techniques*. Springer.

[37] Christian Prehofer. 1997. Feature-Oriented Programming: A Fresh Look at Objects. In *European Conference on Object-Oriented Programming (ECOOP)*. Springer, 419–443.

[38] Iran Rodrigues, Márcio Ribeiro, Flávio Medeiros, Paulo Borba, Baldoino Fonseca, and Rohit Gheyi. 2016. Assessing Fine-Grained Feature Dependencies. *Information and Software Technology* 78 (2016), 27–52.

[39] Marko Rosenmüller and Norbert Siegmund. 2010. Automating the Configuration of Multi Software Product Lines. In *International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS)*. 123–130.

[40] Marko Rosenmüller, Norbert Siegmund, Christian Kästner, and Syed Saif ur Rahman. 2008. Modeling Dependent Software Product Lines. In *Workshop on Modularization, Composition and Generative Techniques for Product Line Engineering (McGPLE)*. 13–18.

[41] Chanchal K. Roy, James R. Cordy, and Rainer Koschke. 2009. Comparison and Evaluation of Code Clone Detection Techniques and Tools: A Qualitative Approach. *Science of Computer Programming* 74, 7 (2009), 470–495.

[42] Ina Schaefer, Rick Rabiser, Dave Clarke, Lorenzo Bettini, David Benavides, Goetz Botterweck, Animesh Pathak, Salvador Trujillo, and Karina Villela. 2012. Software Diversity: State of the Art and Perspectives. *International Journal on Software Tools for Technology Transfer* 14, 5 (2012), 477–495.

[43] Klaus Schmid and Martin Verlage. 2002. The Economic Impact of Product Line Adoption and Evolution. *IEEE Software* 19, 4 (2002), 50–57.

[44] Peri L. Tarr, Harold Ossher, William H. Harrison, and Stanley M. Sutton Jr. 1999. *N* Degrees of Separation: Multi-Dimensional Separation of Concerns. In *International Conference on Software Engineering (ICSE)*. 107–119.

[45] Reinhard Tartler, Daniel Lohmann, Julio Sincero, and Wolfgang Schröder-Preikschat. 2011. Feature Consistency in Compile-Time–Configurable System Software. In *International Conference on Computer Systems (EuroSys)*. ACM, 47–60.

[46] Thomas Thüm, Sven Apel, Christian Kästner, Ina Schaefer, and Gunter Saake. 2014. A Classification and Survey of Analysis Strategies for Software Product Lines. *ACM Computing Surveys* 47, 1 (2014), 1–45.

[47] Frank van der Linden, Klaus Schmid, and Eelco Rommes. 2007. *Software Product Lines in Action*. Springer.

[48] Kentaro Yoshimura, Dharmalingam Ganesan, and Dirk Muthig. 2006. Defining a Strategy to Introduce a Software Product Line Using Existing Embedded Systems. In *International Conference on Embedded Software (EMSOFT)*. ACM, 63–72.