Iterative Development and Changing Requirements: Drivers of Variability in an Industrial System for Veterinary Anesthesia

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

Jacob Krüger
Ruhr-University Bochum &
Otto-von-Guericke-University
Magdeburg, Germany
jacob.krueger@rub.de

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

ABSTRACT

Developing a safety-critical embedded system poses a high risk, since such systems must usually comply with (potentially changing) rigorous standards set by customers and legal authorities. To reduce risk and cope with changing requirements, manufacturers of embedded devices increasingly use iterative development processes and prototyping both for hard- and firmware. However, hard- and firmware development are difficult to align in a common process, because hardware development cycles are typically longer and more expensive. Thus, seamlessly transitioning software to new hardware revisions and reusing old hardware revisions can be problematic. In this paper, we describe an industrial case study for veterinary anesthesia in which we also faced this problem. To solve it, we introduced preprocessor-based variability to create a small configurable system that could flexibly adapt to our needs. We discuss our solution, alternative solutions for hardware evolution, as well as their pros and cons. Our experiences generalize an interesting evolution scenario for systems that are planned and delivered as a single system, but exhibited variability to cope with problems during agile development processes.

CCS CONCEPTS

• **Software and its engineering → Embedded software**; **Requirements analysis**; **Reusability**.

KEYWORDS

Case Study, Configurable System, Driver of Variability, Evolution

ACM Reference Format:
Elias Kuiter, Jacob Krüger, and Gunter Saake. 2021. Iterative Development and Changing Requirements: Drivers of Variability in an Industrial System for Veterinary Anesthesia. In *25th ACM International Systems and Software Product Line Conference - Volume B (SPLC '21), September 6–11, 2021, Leicester, United Kingdom.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3461002.3473950

1 INTRODUCTION

Embedded systems, such as vehicles, domestic appliances, and internet-of-things devices are ubiquitous in our everyday life [37, 44]. Especially in life sciences and medical domains, we depend on the safety and correct operation of such systems, which must usually satisfy rigorous legal regulations. So, developing safety-critical embedded systems is a risky endeavor for manufacturers [13].

In practice, software for embedded systems (i.e., *firmware* [48]) is developed with rudimentary development processes like *burn-and-pray* [42]. Such processes hamper the verification of the system, since they value quick fixes and hacks more than proper design and systematic testing. Furthermore, using traditional development processes (e.g., the waterfall model) hampers the validation of the system, since it is more challenging to take changes in customer requirements and legal regulations into account. To improve the capability for verifying and validating a system (thus reducing the financial risk), manufacturers increasingly adopt iterative (e.g., agile) development practices based on prototyping [14, 18, 23, 29].

However, adopting iterative development practices for embedded systems poses its own challenges [29]. One of these challenges is to align the different durations and costs of hard- and firmware development cycles. That is, developing a new hardware revision usually takes longer and is more expensive than developing a new firmware revision [14, 18, 42, 45]. Thus, a gap between hard- and firmware emerges, which must be bridged somehow to ensure a fast time-to-market and reduce the financial risk.

In this paper, we illustrate this problem based on a case study and discuss several possible solutions, their trade-offs, as well as to what extent they are applicable to other projects in the embedded domain. Our case study is about an industrial system for veterinary anesthesia (i.e., PigNap [31]), which is used by farmers all across Germany. Since this project was a high-risk endeavor, we utilized an iterative development process to adapt to changing requirements and legal regulations. We bridged the aforementioned gap between hard- and firmware by introducing variability into the system, a concept known from configurable systems and software product lines (SPLs) [3, 6, 24]. This allowed us to seamlessly transition to new (and reuse old) hardware revisions; which ultimately contributed to the success of the project.

In detail, we contribute the following in this paper:

- We describe the development of an industrial embedded system, and how variability emerged during its development.
- We discuss the nature of the variability, its drivers, and how it could have been avoided with other solutions.
- We publish our development repository (i.e., code, commits) and analysis results on GitHub to support our findings and enable future research.[1]

SPLC '21, September 6–11, 2021, Leicester, United Kingdom

ACM ISBN 978-1-4503-8470-4/21/09.
https://doi.org/10.1145/3461002.3473950

[1] https://github.com/ekuiter/pignap-case-study

In summary, we contribute towards understanding and resolving the evolutionary gap between hard- and firmware in embedded systems. Notably, our findings are generalizable to many projects in the embedded domain that employ agile practices.

2 CONTEXT

Next, we explain the context of and problem addressed by PigNap based on a brief domain analysis. In the domain of pig farming, castration of male piglets (i.e., newborn pigs) is widely practiced all across the globe [7, 50]. For example, in Germany, there are about 6,800 piglet breeding farms that castrate about 25 million piglets in total every year [43, 49]. This procedure is done, because in 10–75 % of cases the meat of uncastrated boars (i.e., male pigs) may develop an unpleasant smell (known as *boar taint*) that makes the meat inedible and therefore unfit for sale [8]. Currently, it is impossible to remove boar taint or detect it in advance, so the only choice is to prevent it. One widely practiced solution to this problem is the castration of male piglets soon after birth [7]. Besides preventing boar taint, this practice reduces dominant behavior and cannibalism among the animals [50]. As most piglet breeding farms aim for profit maximization, they usually perform the castration procedure without anesthesia, if the local law permits.

In recent years, animal protection movements in Germany have urged politicians to ban piglet castration without anesthesia [49]. Due to these protests, the German federal ministry of food and agriculture (BMEL) initiated a change of the *Tierschutzgesetz* (animal protection act) in 2020 [9]. Thus, since January 2021, piglet castration in Germany is only allowed via vaccination, injection anesthesia, or inhalation anesthesia [17, 46, 51]. From an economic point, inhalation anesthesia tends to be most profitable, since the castration treatment can be performed by the farmers themselves; while the other options require the presence of a veterinarian.

The change of the animal protection act was controversially discussed by politicians, animal rights proponents, farmers, and veterinarians. In particular, the technical feasibility of inhalation anesthesia was questioned, since there were no anesthesia machines on the market that satisfied the strict requirements prescribed by the first draft of the *Ferkelbetäubungssachkundeverordnung* (piglet anesthesia enactment) [10]. However, a compromise was found in 2019: The manufacturers of anesthesia machines had time until 2021 to build prototypes, which were then iteratively reviewed in close cooperation with the *German Agricultural Society* (DLG) [30].

These reviews initially focused on safety requirements, such as workplace safety for the farmers, well-being of the treated piglets, and environmental pollution by the anesthetic. In later iterations, non-functional requirements (e.g., ease of use and tamper-proof history) were also considered [21, 30]. The results of the DLG reviews drove the development of improved prototypes and showed the BMEL which requirements were realistic and which were not. So, a feedback loop between manufacturers, the DLG, and the BMEL was established, which imposed an incremental development process on the manufacturers.

As a result of this process, five piglet anesthesia machines were successfully certified by the DLG. One of these machines is PigNap, for which we developed the firmware in cooperation with our industry partners *HCP-Technology* (responsible for development and assembly) and *BEG Schulze Bremer* (responsible for marketing and training) over the course of one year [31]. Other stakeholders involved in the development of PigNap include the BMEL (legislator), DLG (certificate authority), and customers (i.e., farmers and veterinarians)—each with their own specific requirements and demands for PigNap, which were only partially known in advance.

3 FIRMWARE DEVELOPMENT

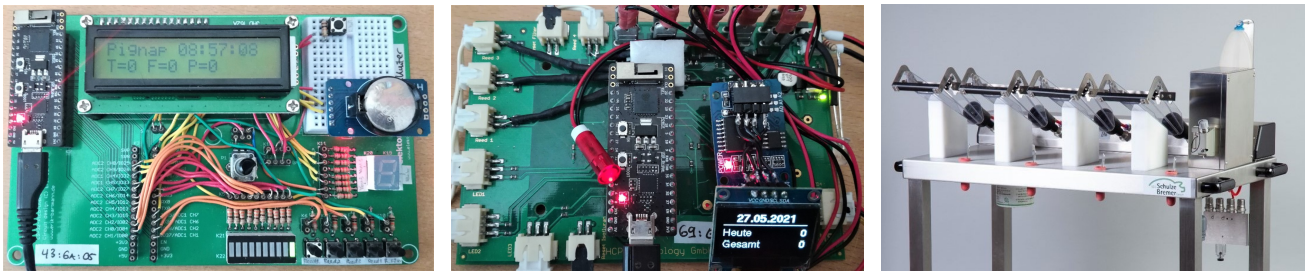
We describe our initial requirements analysis and development of PigNap, focusing on stakeholder interests and emerging variability.

3.1 Initial Requirements Analysis

Our industry partner HCP-Technology is a small German company that designs and manufactures thermoelectric systems for the agricultural and medical domain. In May 2019, HCP-Technology developed an initial concept for a piglet anesthesia machine, which was anticipated to have complex firmware requirements. Due to their lack of expertise in software engineering and our success in previous collaborations [28], they asked the first author of this paper to aid in the firmware design and implementation. Initially, we had an extensive exchange of information, ideas, and clarifications with HCP-Technology. This included familiarizing with the domain (cf. Section 2) and a discussion of the most critical design issues. We describe two discussion topics in more detail, which later proved to have unanticipated impact on variability (cf. Section 4).

Pin Mapping. While eliciting requirements, we noticed that hard- and firmware requirements for an anesthesia machine were clearly separated: The hardware was primarily required to provide a safe, fast, and resource-efficient narcosis; that is, its requirements were mostly concerned with engineering issues regarding pneumatics, valves, and respiratory masks [51]. In contrast, the firmware was required for reliably controlling the narcosis process; that is, its requirements focused on business logic, such as detecting when a piglet is clamped in the device, managing the state of narcosis, and thread safety (as the device should allow parallel treatments of several piglets) [30]. Thus, we agreed on establishing an interface between hardware (which solves engineering issues) and firmware (which implements business logic) by means of a *pin mapping*. A pin mapping determines which input/output pin (i.e., connection ports of the embedded CPU) is connected to which piece of hardware [5, 42, 48]. For PigNap, we intended to use this approach to abstract away low-level details of the narcosis process, so that (as far as the firmware is concerned) the narcosis can be started/stopped simply by en-/disabling some pin. The pin mapping approach allowed us to develop hard- and firmware mostly independently of each other, as long as both complied with the same interface. Thus, we aimed to minimize the risk of bugs, as engineering issues and business logic had minimal potential for unexpected interactions. However, whenever the interface was subject to unexpected change later on (e.g., by rewiring or substituting a hardware device), we had to change hard- and firmware accordingly. Although we were aware of this issue, we adapted the pin mapping approach early in the project, because it enabled us to develop hard- and firmware in parallel and minimize the risk of interaction bugs.

Interpretation of Enactment. Another discussion topic was the first draft of the BMEL's piglet anesthesia enactment [10], which



(a) Initial prototype on PCB_1 with LCD display. (b) PCB_2 with OLED display and $FRAM_1$. (c) Final device with PCB_4, OLED, and $FRAM_2$.

Figure 1: Different device prototypes used throughout the development of the PigNap firmware.

mandated several strict requirements for piglet anesthesia machines. However, these requirements were vaguely phrased, which is why we had to translate them into concrete requirements for PigNap. For example, the enactment required that "anesthesia machines must be suitable on a technical and constructional level to ensure a sufficient depth of narcosis and avoid piglet suffering as much as possible,"[2] without a further definition of "suitable," "sufficient," or "as much as possible" [10]. In theory, the certification authority DLG was responsible for interpreting the enactment and setting concrete standards (e.g., how long piglets should be exposed to the anesthetic) [30]. However, in practice most of these standards were only established by the feedback loop between manufacturers and the DLG described in Section 2. So, for our initial requirements analysis, we had to predict requirements based on domain knowledge from farmers and veterinaries. Thus, it was clear from the beginning that we would probably need to adjust our predicted requirements during development.

Taking the pin mapping as well as enactment interpretation into account, we developed a preliminary and informal list of firmware requirements for our anesthesia machine, which served as the basis for developing our first prototype.

3.2 Development Process

To develop PigNap, we had to choose a development process that fits the context and requirements explained above. For the hard- as well as firmware, we opted for an iterative development process based on prototyping [32, 38, 40]. That is, we repeatedly cycled through several phases (requirements analysis → development → review), where each cycle yielded a functional prototype of the hard- or firmware, respectively. After flashing the firmware onto the hardware, the combined prototype could be reviewed by several stakeholders: only HCP-Technology in the beginning and farmers, veterinarians, as well as the DLG later on. These reviews typically shed some light on several issues and enactment misinterpretations, which we incorporated into the next development phase.

We chose a prototype-based iterative process mainly because the animal protection act prescribes that "the BMEL reports at least every six months to the German parliament about the current state of anesthesia machines" [9]. As we and our customers had considerable interest in taking the enactment into a more practically feasible direction, it was essential that we participated in this feedback loop, which was facilitated by our prototype-based process. In addition, our discussions regarding the pin mapping and enactment interpretation (cf. Section 3.1) fitted well with an iterative process.

3.3 Development of First Prototype

In June 2019, we began to develop the first PigNap prototype, which we show in Figure 1a. Regarding the embedded CPU, we quickly settled for the ESP32 microcontroller [35], which had enough computation power and input/output pins to adapt to new and changing requirements. Thus, we aimed to minimize the risk of needing CPU upgrades, which would have required us to rewrite large parts of the codebase. Regarding external components, we mostly focused on the pins in our pin mapping that were directly related to the business logic of the anesthesia treatment process. Due to the pin mapping interface (cf. Section 3.1), we were able to simulate this treatment process with simple buttons and LED lights as shown in Figure 1a (i.e., a button press indicates that a piglet is clamped in the device and the LEDs indicate the state of narcosis). This allowed us to develop a first version of the firmware while the first hardware prototype was still in its inception.

We implemented the firmware in C with the ESP32 SDK. This SDK is applicable in many scenarios and systems and uses Kconfig to handle CPU-internal variability [47]. We did not use this mechanism, since the default options worked fine for our purposes. In later development cycles, we used the `#define` and `#ifdef` directives of the C preprocessor and simple build scripts to manage emerging variability (cf. Section 3.4). We developed the first firmware version for the prototype shown in Figure 1a in 38 days, with 5,327 lines of code in 48 commits on 42 files. Then, we evolved that version over the course of one year, leading to a final version with 7,081 lines of code in 138 commits over 59 files (including all variability).

3.4 Emerging Variation Points

To discuss later development cycles of PigNap with respect to stakeholder interests and emerging variability, we performed an in-depth analysis of our development repository (i.e., all 138 commits, June 2019 to June 2020). During our analysis, we manually inspected each commit with respect to three questions:

(1) Which firmware requirements does this commit address (e.g., introducing some desired functionality)?
(2) Which hardware requirements does this commit address (e.g., changing the pin mapping)?
(3) Which stakeholders hold interests in these requirements (e.g., who requested the change and why)?

[2] All excerpts from German law are loosely translated by the authors.

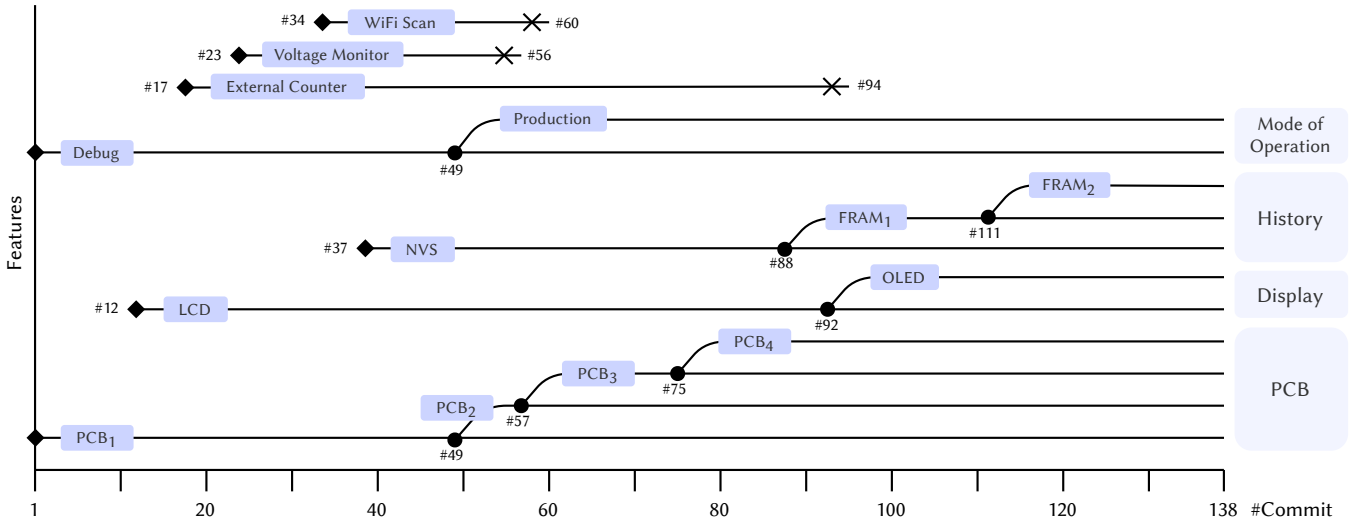


Figure 2: Timeline for the development of the PigNap firmware over the course of one year.

To answer these questions, we cross-referenced the committed code, mail correspondence, and memory of the project lead at HCP-Technology as well as the first author. We provide the detailed results of our analysis in our repository. Due to corporate interests, we cannot publish all implementation files; however, all variability-related code artifacts are available for analysis.

Based on our analysis, we identified *features* (i.e., user-visible functionalities [3]) and variation points that emerged during the development of the PigNap firmware. In Figure 2, we visualize these features and variation points in a timeline. On the x-axis, we show the number of each commit in our development repository. On the y-axis, we show which features are actively supported in the firmware revision corresponding to the given commit. In the timeline, we emphasize important events like the inception (⧫) and removal (×) of features as well as variation points (•). For each commit on the x-axis, we can derive a feature model for the corresponding firmware revision from the features and variation points on the y-axis. That is, our timeline is equivalent to a temporal feature model [39], which annotates each feature with the date of its inception (and, possibly, removal).

In Figure 3, we show the feature model for the final firmware version at commit #138. This feature model comprises four subtrees with alternative features, where each subtree represents some variation points that emerged after the development of the first prototype (i.e., starting at commit #49). In the following, we describe each feature subtree in Figure 3 (corresponding to the variation points highlighted identically in Figure 2) in more detail.

Printed Circuit Board. We can see in Figure 1, that the ESP32 microcontroller that runs our firmware is mounted on a printed circuit board (PCB) along with resistors, capacitors, and other hardware accessories (e.g., a real-time clock with a battery). As described, we developed hard- and firmware simultaneously by following the initial pin mapping we described in Section 3.1, which corresponds to PCB_1 in Figure 1a (commit #1). However, several times during development, we had to change the pin mapping for various reasons, namely genuine **mistakes** (mostly due to misinterpreting the ESP32 documentation) and **changing requirements** expressed by stakeholders during reviews (e.g., we removed an external treatment counter in favor of a better display at the request of customers).

Since the pin mapping represents the interface between hard- and firmware, **each change necessitated the design and production of a new PCB revision** (e.g., PCB_2). This process was expensive and time-consuming, because new PCB revisions were produced only every few months and only in larger quantities by an external company. Thus, we could not afford to produce a new PCB for each single change of the pin mapping. Instead, we kept a record of all outstanding pin-mapping changes, which we forwarded to the external company shortly before the next PCB would be produced. This happened three times during development, namely in commits #49 (PCB_2 in Figure 1b), #57 (PCB_3), and #75 (PCB_4, used in the final system in Figure 1c), which we encode as features in Figure 3. Batching the pin-mapping changes together was advantageous on the hardware side, reducing logistic and production costs. However, this method also had the disadvantage that pin-mapping changes were reflected in the hardware only with a month-long delay (i.e., the *transition phase*), during which the firmware was required to be compatible with two PCB revisions at the same time: First, we had to support the old PCB revision at least until the new PCB was produced, which allowed us to use the old revision for testing firmware changes. Second, we also had to support the new PCB before it was produced, enabling us to seamlessly transition to the device prototypes that would use the new PCB. So, during the transition phase between two PCB revisions, our firmware needed to support both revisions. We embraced this transitional variability by introducing corresponding variation points, which we implemented with preprocessor directives (e.g., `#ifdef BOARD_VERSION >= 2`).

In addition, we decided to keep the support for old PCB revisions even after the transition phase was over; that is, we never phased out any PCB revisions. We opted for this practice for three reasons: First, it allowed us to keep using old PCB revisions for development and testing. This was convenient, since the revisions PCB_3 and PCB_4 were intended to be used only in device prototypes, and we had no access to such device prototypes for developing and testing

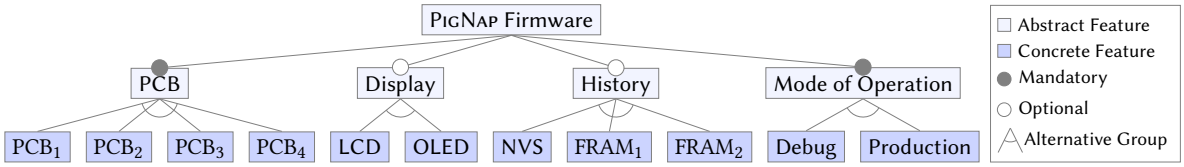


Figure 3: Final feature model for the PigNap firmware, corresponding to commit #138 in Figure 2.

the firmware. Second, the piglet anesthesia enactment explicitly allowed that "anesthesia machines that were built before this enactment comes into effect may still be used even when they do not satisfy the requirements listed above" [10]. So, we were allowed to reuse old prototypes for lab testing at the DLG, in-field testing on farms, or in educational institutions. This *hardware reuse* was convenient, because it allowed us to efficiently make use of the prototypes that we built throughout development (and not waste the amount of work and expensive components put into building these prototypes). However, to leverage this hardware reuse, we had to keep developing the firmware for old PCB revisions as well (e.g., so that old devices would still receive up-to-date bugfixes). Finally, the costs of keeping variability in the firmware instead of discarding it after the transition phase was negligible. This was because there was almost no maintenance effort for a feature after it was initially developed.

Display. The second feature subtree concerns the display that is mounted on PigNap and is visible to the customer. Early on (commit #12), we found that having a display with status information may be useful to convey information to customers, such as the number of treatments or critical status messages (e.g., low battery warning). We initially chose an off-the-shelf LCD display (cf. Figure 1a) that we used to report such information to the customer. However, we later received **criticism from customers** regarding the readability of the information on the display. Thus, we decided to switch to a more expensive OLED display (cf. Figure 1b), which could display more information in a more organized and readable way. However, for similar reasons as described above for the PCB subtree, we still had to support the LCD display during the transition phase; that is, until we integrated the OLED display into the hardware design. After we implemented support for both displays (commit #92), we kept the support for LCD displays in the firmware to facilitate hardware reuse as described above.

History. Among the requirements mandated by the first draft of the piglet anesthesia enactment is that "anesthesia machines must store a tamper-safe history of the number of treatments and their date for at least one year" [10]. This was necessary to ensure that anesthesia machines are actually used by the farmers and no illegal castration without anesthesia takes place. Of all requirements, this was the most challenging to implement, because it necessitated many new requirements for the hard- and firmware: On a hardware level, we needed a persistent storage to store the history while the device is switched off; to track the current date, a real-time clock was required; and for the clock, a battery was needed. On a firmware level, this implied that we needed a corruption-resilient data structure to store log data, a Wifi-based user interface for accessing log data, and a tamper-proof mechanism for switching the battery. Since these issues required complex solutions, the implementation of the history subtree accounts for half of the firmware codebase.

Due to the complexity of the history requirement, we decided to **relax this requirement** for our first prototype to demonstrate the feasibility of our solution. This means that we initially used the built-in non-volatile storage (NVS) of the ESP32 microcontroller instead of a dedicated storage chip (commit #37). Using the built-in NVS had the advantage that building the first prototype was faster, since we did not need to integrate an external device driver. On the downside, the built-in NVS is not durable enough for the targeted system lifespan of 20 years. So, this was only a temporary solution and we later enhanced it with a dedicated 2KiB ferroelectric RAM storage chip in commit #88 (feature $FRAM_1$). To allow for hardware reuse and bridge the transition phase, we again decided to keep the NVS support for old prototypes. This was possible, because storing a history is not relevant in educational institutions, where the old prototypes were used.

A second change became necessary when, in February 2020, the **enactment draft was changed** by the BMEL to prescribe storing a history "for at least three years" [10]. Unfortunately, we could not address this unanticipated change solely by updating the firmware, because $FRAM_1$ did not have enough storage to save three years of history. We therefore had to select a larger, more expensive 8KiB storage chip (feature $FRAM_2$). This chip had a different device driver, so we introduced another variation point in commit #111.

Mode of Operation. When we developed our initial prototype (cf. Figure 1a), we were not aware of the optimal parameters for narcosis (e.g., how long piglets should be exposed to the anesthetic or how many piglets can be treated with one vial of the anesthetic). We therefore used arbitrary values in the beginning (commit #1), which were **unsuitable for in-field usage**. Consequently, we introduced a variation point in commit #49 that distinguishes realistic values (*Production*) from dummy values (*Debug*). By extensive in-field testing, we successively refined the values used in production mode. Nonetheless, we kept the dummy values in the feature model, because they were useful for testing during the development phase.

3.5 Mandatory and Removed Features

For some features (i.e., mandatory, removed), we did no implement variability, which we briefly discuss next.

Mandatory Features. We omit a detailed discussion of mandatory features (e.g., narcosis state management, filter reset switch) that were neither removed nor led to variability. In Figure 3, these features can be considered to be integrated in the root feature *PigNap firmware*. Note that not all requirement changes necessitated variation points: Whenever possible, we integrated changes seamlessly without changing the interface between hard- and firmware (e.g., when we rebranded the software to BEG Schulze Bremer). We did this to keep the number of variation points to a minimum.

Removed Features. Since it was not always possible to correctly predict customer needs and standards set by the DLG, we also

had to remove some features during the development process. For example, we developed a voltage monitor that ensured that the device would always have some time to commit history changes whenever it was turned off (commit #23). However, this setup did not work correctly in some edge cases, so we abandoned it in favor of a pure software solution (commit #56). Other examples include an external counter (#17–#94) and a WiFi access point scan (#34–#60), which we dropped because they provided little additional value.

3.6 Final Configuration Space

We concluded the development of PigNap in June 2020 at commit #138 with a last adjustment of the parameters for narcosis in the feature *Production*. At this point, we could build and flash the firmware for all variants admitted by the feature model in Figure 3, some of which corresponded to old prototypes. For instance, we show in Figure 1a and Figure 1b setups for the configurations $\{PCB_1, LCD, Debug\}$ and $\{PCB_2, OLED, FRAM_1, Debug\}$, respectively (omitting abstract features). For the final device (cf. Figure 1c) that we delivered to customers, we used the configuration $\{PCB_4, OLED, FRAM_2, Production\}$. This configuration satisfied all (possibly changed) requirements that we elicited over the course of the project, therefore it can be considered the canonical configuration of the PigNap firmware. Between the finalization of the canonical configuration in June 2020 and the commencement of the enactment in January 2021, we successfully produced, flashed, and sold hundreds of PigNap devices to farmers in Germany. In a recent survey, PigNap was estimated to have a market share of 33% [21]. In summary, we consider this project a success story, with two contributing factors to this success being our iterative development process and our handling of emergent variability.

4 DISCUSSION

In this section, we discuss the bigger picture of variability in PigNap and similar projects. We first clarify whether PigNap is an SPL and the nature of the variability that emerged throughout the system's evolution. Then, we point out the driving factors of this variability and alternative scenarios based on which the variability could have been avoided. Finally, we discuss the implications and tradeoffs of these scenarios with regard to our case study and other projects in the embedded domain.

4.1 Is PigNap a Software Product Line?

An SPL is "a set of software-intensive systems that share a common, managed set of features satisfying the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way" [11]. Although PigNap has a feature model with a "common, managed set of features," it only partially satisfies this definition: The variability in the PigNap feature model does not serve "the *specific* needs of a particular market segment or mission." As we explained in Section 3.6, PigNap is only sold in one canonical configuration, and thus has no external variability (i.e., the ability for customers to choose between different variants) [41]. Instead, all variability in Figure 3 is internal (i.e., hidden from customers). Since PigNap was not intended to be an SPL and almost the entire revenue is generated by a single product, we do not consider it a classical SPL.

Table 1: Impact of variability drivers in PigNap.

Driver \ Feature subtree	PCB	Display	History	Mode
Iterative development				
of hardware	●	●	●	◐
of firmware	○	○	○	●
Changing requirements				
from customers	◐	●	○	○
from BMEL and DLG	◐	○	●	◐

● high impact ◐ medium impact ○ no impact

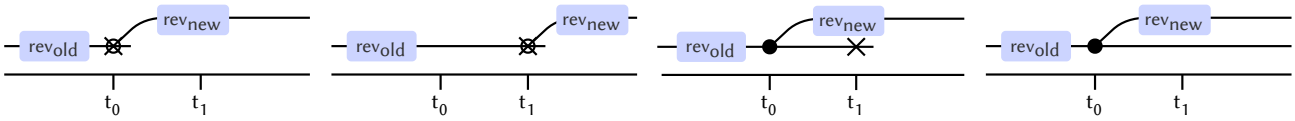
4.2 Domain-Intrinsic Variability

As the development of PigNap started and concluded with a single system, we could argue that the variability that emerged (cf. Section 3.4) was only "accidental" and perhaps could have been avoided. To examine this claim in more detail, we classify features in the feature model as (not) intrinsic to the modeled domain. That is, a feature is domain-intrinsic if it arises naturally from the domain knowledge elicited from experts. In contrast, a feature is not domain-intrinsic when it is only introduced to correct a previous mistake, adapt to changing requirements, or when it is only necessitated by the development process. Based on this distinction, we can classify all concrete features in Figure 3 as not domain-intrinsic (or internal variability due to technical reasons according to Pohl et al. [41]). Actually, none of these features emerged naturally from the domain of anesthesia machines. Instead, they can be considered "byproducts" of the specific circumstances given by our design and development choices. Thus, all variability in PigNap could have been avoided by making different design decisions in the beginning.

4.3 Drivers of Variability

If all variability could have been avoided by making different choices, the question arises, which choices precisely promoted variability and why? We refer to these choices and circumstances as *drivers of variability*, which were primarily responsible for the emergent variability. Precisely, we identified iterative development and changing requirements as the primary drivers of variability in PigNap. In Table 1, we show their respective impact on the feature subtrees in Figure 3, and discuss them in the following.

Iterative Development of Hardware. In Section 3.2, we described and justified our iterative development process based on prototyping. Notably, we chose to develop the hard- and firmware of PigNap *in parallel* by means of the pin-mapping interface we described in Section 3.1. Hardware, however, is fundamentally different from firmware with regard to the design and engineering process: To produce hardware (e.g., PCB, respiratory masks), components must be obtained and assembled in a lengthy process. This was further complicated in our case study, because we outsourced the PCB production to an external company. In addition, the COVID-19 pandemic began to have a negative impact on the component supply chain in early 2020 [20, 21]. Thus, producing a new prototype of the PigNap device took weeks to months, while changes to the firmware could usually be adapted quickly—that is, the hardware "lagged behind" the firmware. The resulting gap between hard- and firmware was a high-impact driver of variability in the *PCB*, *Display*,



(a) Early Shift (ES) (b) Late Shift (LS) (c) Transition Phase (TP) (d) Hardware Reuse (HR)

Figure 4: Scenarios for evolving hardware revisions in an iterative development process with changing requirements.

and *History* subtrees. Without this gap, we would not have needed to introduce variability in these subtrees at all.

Iterative Development of Firmware. The variability in the *Mode of Operation* subtree was only partially driven by the gap between hard- and firmware. Instead, it mostly emerged from our development practices for the firmware: Usually, after each change to the firmware, we would test it immediately on our simulation-based prototypes (cf. Figure 1). Thus, we could quickly identify and solve obvious problems introduced by our changes. Because our work on the firmware was mostly remote from the site where the hardware was produced (especially during the pandemic), this practice allowed us to save time and money for remote maintenance. However, these quick implementation and test cycles also required an efficient way to simulate the device with dummy values, which drove the introduction of the feature *Debug*.

Changing Requirements. In Section 3.1, we mentioned how one reason for choosing an iterative development process was that we anticipated certain requirements to change during development. These can mostly be classified into customer requirements (which were requested by farmers and veterinarians) and requirements from the BMEL and DLG (which related to the piglet anesthesia enactment). As we described in Section 3.4, introducing the *Display* subtree was mostly driven by customers, and the *History* subtree by changes to the enactment. In addition, both kinds of requirement changes also had some impact on the *PCB* subtree, because the display and FRAM components affected the pin mapping. In retrospect, some of the introduced variability (e.g., regarding the display) could have been avoided with a more detailed initial requirements analysis. However, in the beginning it was more important to quickly develop a first prototype than to analyze customer requirements.

4.4 Scenarios for Hardware Evolution

In Section 4.2, we noted that none of the variability in PigNap is domain-intrinsic, and thus could have been avoided by making different design decisions. With our knowledge about drivers of variability from Section 4.3, we now discuss concrete evolution scenarios with which we could have avoided variability in PigNap.

The natural solution for avoiding variability is to avoid all drivers of variability. If we had not chosen an iterative development process for PigNap and knew all requirements in advance, no variability would have emerged. For example, we could have used the waterfall model instead of an iterative model [4]. In this case, the hardware would have been developed first; and only afterwards we would have developed the firmware, which would have avoided the main driver of variability (i.e., the parallel development of hard- and firmware). However, in our case study, this scenario clearly violates our initial requirements analysis (cf. Section 3.1). First, we had to rely on the feedback loop with the BMEL, DLG, and customers to successively refine our device, because we did not know all requirements in advance. Second, the idea to completely decouple hard- and firmware development in an embedded system is unrealistic, they simply interact too much to be considered independently [42].

A more realistic solution to avoid variability may be to not completely avoid all drivers of variability, but relax some of the desired properties of iterative development that we described in Section 3.4. In Figure 4, we show four scenarios for hardware evolution, which are all intended to bridge the gap that arises when hard- and firmware are developed in parallel. We show time and variability on the x- and y-axis, respectively; and the time period between the design (t_0) and production (t_1) of a new hardware revision rev_{new} is marked on the x-axis. The four scenarios we show have different (dis-)advantages in terms of the properties they relax, which we summarize in Table 2. We discuss all scenarios in detail and explain their applicability in our case study.

Early Shift. In the first scenario (cf. Figure 4a), we begin to develop the firmware for a new hardware revision rev_{new} as soon as rev_{new} is fully designed (t_0). In our case study, this would have meant to fully shift the development to a new PCB revision, display, or storage chip as soon as the pin mapping was changed or the new component selected. The old firmware revision (with support for the old hardware revision rev_{old}) is immediately discontinued in this scenario. Thus, we avoid introducing any variability. However, old hardware revisions like rev_{old} are neither supported during the transition ($t \in [t_0, t_1]$) nor after the transition ($t > t_1$).

Late Shift. The second scenario is similar to the first one; it also involves a full shift from rev_{old} to rev_{new}. However, in this scenario, we shift development to rev_{new} only when it is fully produced (t_1). Thus, we still support rev_{old} during the transition, although we may not be able to seamlessly shift to the new firmware version in time.

In both scenarios, early and late shift, we forcibly synchronize hard- and firmware development. For early shifting, the developed firmware must wait for the hardware production to catch up; while for late shifting, the produced hardware must wait for the firmware development to catch up. Thus, old revisions are only partially supported and a seamless shift to a new firmware revision is hampered.

Transition Phase. The third scenario addresses these problems as follows: During the transition, both rev_{old} and rev_{new} are temporarily encoded as two firmware variants, which exist at the same time and are developed in parallel. Thus, we deliberately mix revisions (i.e., variability in time) and variants (i.e., variability in space) in this scenario [2, 41]. This scenario has the advantage that we support both the old and new hardware revisions during the transition and enable a seamless shift to rev_{new} as well. However, it also comes at the expense of temporarily introducing variability, which must be implemented and maintained during the transition phase. Therefore, this scenario should only be adopted when developers have sufficient expertise in implementing software variability.

Table 2: Properties of scenarios for hardware evolution.

Property \ Scenario	ES	LS	TP	HR
Supports seamless shift to new revision	●	○	●	●
Supports old revisions during transition	○	●	●	●
Supports old revisions after transition	○	○	○	●
Avoids variability	●	●	◐	○

● yes ◐ partially ○ no

Hardware Reuse. The fourth and final scenario generalizes the third: We completely embrace the variability that was introduced in the transition phase. That is, variability is *not* temporary in this scenario, meaning that all old hardware revisions are indefinitely supported. Depending on the project, this may incur maintenance costs or variability bugs. In our case study, this scenario was attractive, because the fast time-to-market and economic constraints of our project made it necessary that we used the available time and hardware as well as we could. Compared to a single PigNap system, which costs over 9,000 €, the implementation and maintenance costs of software variability [26] were negligible for us. In addition, our expertise in implementing variable software systems [28] was helpful to contain the complexity of our variable source code.

4.5 Generalizing the Lessons Learned

We discussed the drivers of variability in PigNap and which evolution scenario was best for our project to deal with the gap between hard- and firmware. We argue that our experiences can be generalized to other projects in the embedded domain. First, when developing a real-world embedded system, it is hard to completely separate the development of hard- and firmware. Typically, firmware is intended to build on and leverage the specific properties of the hardware to fulfill its task. However, hard- and firmware will usually be developed simultaneously, with one guiding the design of the other to achieve a suitable balance between hard- and firmware design [14, 23, 29]. Second, real-world systems require verification (to examine functional correctness of the system) and validation (to ensure that the system satisfies customer needs). By incrementally developing and testing a series of prototypes, the risk of fundamental verification and validation issues can be reduced; and if a problem occurs, the project can still be steered into the right direction. Thus, an iterative development process in which hard- and firmware are developed simultaneously is natural for developing a new system in the embedded domain [29, 45]. In addition, customer needs (and in our case, laws and regulations) frequently change and drive the development of new hardware and firmware revisions [18, 42]. So, iterative development and changing requirements are not only drivers of variability in PigNap, but can be expected to affect the development of many systems in the embedded domain in general.

In particular, this means that all embedded systems with iterative development and changing requirements will inevitably face emerging variability, due to the gap between hard- and firmware. As guidance for deciding how to deal with this gap, project managers may consider our scenarios for hardware evolution and their tradeoffs. Precisely, they may choose one scenario over the others, depending on the specific circumstances and desired properties of the system. If, for instance, the hardware is cheap and can be quickly produced, the early or late shift strategies may be more appropriate. Such tradeoffs also depend on the developers' experience with SPLs: If said experience is low, it may be too risky to introduce long-term variability.

5 RELATED WORK

We are not aware of a similar study on variability in embedded systems that is not domain-intrinsic (i.e., that emerges and evolves during the development process). There are numerous case studies that report on an iterative or re-engineering-based adoption of SPLs [1, 12, 16, 19, 22, 25, 27, 28, 36, 52, 53] as well as empirical studies on variability that partly consider not domain-intrinsic features [15, 33, 34]. Still, none of these works focuses on the evolution of such variability during development or provides a dataset annotated by the original developer. Consequently, we provide novel contributions regarding an interesting aspect of variability evolution.

Considering agile development practices, Könnölä et al. [29] analyze three industrial case studies with respect to the potential and challenges of agile methods in embedded systems development. They acknowledge the gap between hard- and firmware development we mentioned as a primary concern for developing new embedded systems. Several other studies [14, 18, 42, 45] and a literature review of Kaisti et al. [23] also argue that the co-development of hard- and firmware implies challenges for agile development practices. However, these works neither analyze variability in their case studies, nor do they suggest a solution for the gap between hard- and firmware. So, our contributions advance upon these works.

6 CONCLUSION

Developing embedded systems in safety-critical domains (e.g., veterinary anesthesia) comes with high risks, since customer needs and legal regulations are likely to change and subject to interpretation by certification authorities. An incremental development process based on prototyping is crucial to reduce this risk. However, the emerging gap between hard- and firmware development poses a challenge, because hard- and firmware fundamentally differ in their design and engineering processes. In this paper, we described and discussed an industrial case study in which we faced this challenge. Our solution was to introduce preprocessor-based variability into our system, which yielded a small configurable system that could flexibly adapt to all hardware revisions produced (i.e., hardware reuse). This was the best course of action in our project, considering that the costs for producing hardware exceeded the costs for introducing variability significantly. However, other scenarios for hardware evolution are also plausible (i.e., early shift, late shift, or transition phase), each with their individual strengths and weaknesses. With our work, we aim to motivate further research on this issue and provide recommendations for project managers to decide which scenario may best fit their project.

In future work, we aim to further analyze our scenarios for hardware evolution on other case studies, allowing us to improve our current recommendations. Also, it seems promising to develop (semi-)automated tool support, for instance, for removing temporary variability in the transition-phase scenario.

Acknowledgments. Partly supported by the DFG (SA 465/49-3).

REFERENCES

[1] Jonas Åkesson, Sebastian Nilsson, Jacob Krüger, and Thorsten Berger. 2019. Migrating the Android Apo-Games into an Annotation-Based Software Product Line. In *International Systems and Software Product Line Conference (SPLC)*. ACM, 103–107. https://doi.org/10.1145/3336294.3342362

[2] Sofia Ananieva, Sandra Greiner, Thomas Kühn, Jacob Krüger, Lukas Linsbauer, Sten Grüner, Timo Kehrer, Heiko Klare, Anne Koziolek, Henrik Lönn, Sebastian Krieter, Christoph Seidl, S. Ramesh, Ralf Reussner, and Bernhard Westfechtel. 2020. A Conceptual Model for Unifying Variability in Space and Time. In *International Systems and Software Product Line Conference (SPLC)*. ACM, 15:1–12. https://doi.org/10.1145/3382025.3414955

[3] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. 2013. *Feature-Oriented Software Product Lines*. Springer. https://doi.org/10.1007/978-3-642-37521-7

[4] S. Balaji and M. Sundararajan Murugaiyan. 2012. Waterfall vs. V-Model vs. Agile: A Comparative Study on SDLC. *International Journal of Information Technology and Business Management* 2, 1 (2012), 26–30.

[5] Michael Barr. 1999. *Programming Embedded Systems in C and C++*. O'Reilly.

[6] 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. https://doi.org/10.1145/2430502.2430513

[7] Regina Binder, Werner Hagmüller, Peter Hofbauer, Christine Iben, U. S. Scala, Christoph Winckler, and Johannes Baumgartner. 2004. Aktuelle Aspekte der Kastration männlicher Ferkel. 1. Mitteilung: Tierschutzrechtliche Aspekte der Ferkelkastration sowie Verfahren zur Schmerzausschaltung bei der chirurgischen Kastration. *Wiener Tierärztliche Monatsschrift* 91 (2004), 178–183.

[8] Michel Bonneau. 1998. Use of Entire Males for Pig Meat in the European Union. *Meat Science* 49, Supplement 1 (1998), 257–272. https://doi.org/10.1016/S0309-1740(98)90053-5

[9] Bundesgesetzblatt. 2020. Tierschutzgesetz (TierSchG).

[10] Bundesgesetzblatt. 2020. Verordnung zur Durchführung der Betäubungmit Isofluran bei der Ferkelkastration durch sachkundige Personen (FerkBetSachkV).

[11] Paul Clements and Linda Northrop. 2002. *Software Product Lines: Practices and Patterns*.

[12] Jamel Debbiche, Oskar Lignell, Jacob Krüger, and Thorsten Berger. 2019. Migrating Java-Based Apo-Games into a Composition-Based Software Product Line. In *International Systems and Software Product Line Conference (SPLC)*. ACM, 98–102. https://doi.org/10.1145/3336294.3342361

[13] William R. Dunn. 2003. Designing Safety-Critical Computer Systems. *Computer* 36, 11 (2003), 40–46. https://doi.org/10.1109/MC.2003.1244533

[14] Ulrik Eklund, Helena Holmström Olsson, and Niels J. Strøm. 2014. Industrial Challenges of Scaling Agile in Mass-Produced Embedded Systems. In *International Conference on Agile Software Development (XP)*. Springer, 30–42. https://doi.org/10.1007/978-3-319-14358-3_4

[15] Wolfram Fenske, Sandro Schulze, and Gunter Saake. 2017. How Preprocessor Annotations (Do Not) Affect Maintainability: A Case Study on Change-Proneness. In *International Conference on Generative Programming: Concepts and Experiences (GPCE)*. ACM, 77–90. https://doi.org/10.1145/3136040.3136059

[16] Thomas S. Fogdal, Helene Scherrebeck, Juha Kuusela, Martin Becker, and Bo Zhang. 2016. Ten Years of Product Line Engineering at Danfoss: Lessons Learned and Way Ahead. In *International Systems and Software Product Line Conference (SPLC)*. ACM, 252–261. https://doi.org/10.1145/2934466.2934491

[17] Bundesanstalt für Landwirtschaft und Ernährung. 2020. *Alternativen zur betäubungslosen Ferkelkastration*. Brochure 2001.

[18] Bill Greene. 2004. Agile Methods Applied to Embedded Firmware Development. In *International Conference on Agile Software Development (XP)*. IEEE, 71–77. https://doi.org/10.1109/ADEVC.2004.3

[19] Sten Grüner, Andreas Burger, Tuomas Kantonen, and Julius Rückert. 2020. Incremental Migration to Software Product Line Engineering. In *International Systems and Software Product Line Conference (SPLC)*. ACM, 5:1–11. https://doi.org/10.1145/3382025.3414956

[20] Dabo Guan, Daoping Wang, Stephane Hallegatte, Steven J. Davis, Jingwen Huo, Shuping Li, Yangchun Bai, Tianyang Lei, Qianyu Xue, D'Maris Coffman, Danyang Cheng, Peipei Chen, Xi Liang, Bing Xu, Xiaosheng Lu, Shouyang Wang, Klaus Hubacek, and Peng Gong. 2020. Global Supply-Chain Effects of COVID-19 Control Measures. *Nature Human Behaviour* 4 (2020), 577–587. https://doi.org/10.1038/s41562-020-0896-8

[21] Susanne Gäckler, Sophie Gumbert, Jürgen Harlizius, Wilfried Hopp, and Frederik Löwenstein. 2021. Isofluran-Narkose: Vieles läuft noch nicht rund. *Top Agrar* 7 (2021), 18–22.

[22] Takahiro Iida, Masahiro Matsubara, Kentaro Yoshimura, Hideyuki Kojima, and Kimio Nishino. 2016. PLE for Automotive Braking System with Management of Impacts from Equipment Interactions. In *International Systems and Software Product Line Conference (SPLC)*. ACM, 232–241. https://doi.org/10.1145/2934466.2934490

[23] Matti Kaisti, Ville Rantala, Tapio Mujunen, Sami Hyrynsalmi, Kaisa Könnölä, Tuomas Mäkilä, and Teijo Lehtonen. 2013. Agile Methods for Embedded Systems Development–A Literature Review and a Mapping Study. *EURASIP Journal on Embedded Systems* 2013, 15 (2013), 1–16. https://doi.org/10.1186/1687-3963-2013-15

[24] Kyo C. Kang, Sholom G. Cohen, James A. Hess, William E. Novak, and A. Spencer Peterson. 1990. *Feature-Oriented Domain Analysis (FODA) Feasibility Study*. Technical Report CMU/SEI-90-TR-21. Carnegie Mellon University.

[25] Jacob Krüger and Thorsten Berger. 2020. Activities and Costs of Re-Engineering Cloned Variants Into an Integrated Platform. In *International Working Conference on Variability Modelling of Software-Intensive Systems (VaMoS)*. ACM, 21:1–10. https://doi.org/10.1145/3377024.3377044

[26] Jacob Krüger and Thorsten Berger. 2020. An Empirical Analysis of the Costs of Clone- and Platform-Oriented Software Reuse. In *Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM, 432–444. https://doi.org/10.1145/3368089.3409684

[27] Jacob Krüger, Wardah Mahmood, and Thorsten Berger. 2020. Promote-pl: A Round-Trip Engineering Process Model for Adopting and Evolving Product Lines. In *International Systems and Software Product Line Conference (SPLC)*. ACM, 2:1–12. https://doi.org/10.1145/3382025.3414970

[28] 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 *International Systems and Software Product Line Conference (SPLC)*. ACM, 179–189. https://doi.org/10.1145/3233027.3233050

[29] Kaisa Könnölä, Samuli Suomi, Tuomas Mäkilä, Tero Jokela, Ville Rantala, and Teijo Lehtonen. 2016. Agile Methods in Embedded System Development: Multiple-Case Study of Three Industrial Cases. *Journal of Systems and Software* 118 (2016), 134–150. https://doi.org/10.1016/j.jss.2016.05.001

[30] DLG-Fachzentrum Landwirtschaft. 2019. *Prüfrahmen DLG-TH 10:2019-12, Version 1: Narkosegeräte für die Ferkelkastration*. Standard.

[31] DLG-Fachzentrum Landwirtschaft. 2020. *DLG-Prüfbericht 7081: BEG Schulze Bremer GmbH – Isofluran-Narkosegerät PigNap 4.0*. Audit Report.

[32] Craig Larman. 2004. *Agile and Iterative Development: A Manager's Guide*. Addison-Wesley.

[33] Jörg Liebig, Sven Apel, Christian Lengauer, Christian Kästner, and Michael Schulze. 2010. An Analysis of the Variability in Forty Preprocessor-Based Software Product Lines. In *International Conference on Software Engineering (ICSE)*. ACM, 105–114. https://doi.org/10.1145/1806799.1806819

[34] Kai Ludwig, Jacob Krüger, and Thomas Leich. 2019. Covert and Phantom Features in Annotations: Do They Impact Variability Analysis?. In *International Systems and Software Product Line Conference (SPLC)*. ACM, 218–230. https://doi.org/10.1145/3336294.3336296

[35] Alexander Maier, Andrew Sharp, and Yuriy Vagapov. 2017. Comparative Analysis and Practical Implementation of the ESP32 Microcontroller Module for the Internet of Things. In *International Conference on Internet Technologies and Applications (ITA)*. IEEE, 143–148. https://doi.org/10.1109/ITECHA.2017.8101926

[36] 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. https://doi.org/10.1145/3109729.3109748

[37] Peter Marwedel. 2021. *Embedded System Design: Embedded Systems Foundations of Cyber-Physical Systems, and the Internet of Things*. Springer. https://doi.org/10.1007/978-3-030-60910-8

[38] Justus D. Naumann and A. Milton Jenkins. 1982. Prototyping: The New Paradigm for Systems Development. *MIS Quarterly* 6, 3 (1982), 29–44. https://doi.org/10.2307/248654

[39] Michael Nieke, Christoph Seidl, and Sven Schuster. 2016. Guaranteeing Configuration Validity in Evolving Software Product Lines. In *International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS)*. ACM, 73–80. https://doi.org/10.1145/2866614.2866625

[40] Nienke Nieveen. 1999. Prototyping to Reach Product Quality. In *Design Approaches and Tools in Education and Training*. Springer, 125–135.

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

[42] Timo Punkka. 2005. Agile Methods and Firmware Development. *SoberIT* (2005), 1–21.

[43] Christa Rohlmann, Mandes Verhaagh, and Josef Efken. 2019. *Steckbriefe zur Tierhaltung in Deutschland: Ferkelerzeugung und Schweinemast*. Technical Report. Johann Heinrich von Thünen-Institut. Bundesforschungsinstitut für Ländliche Räume, Wald und Fischerei.

[44] Farzad Samie, Lars Bauer, and Jörg Henkel. 2016. IoT Technologies for Embedded Computing: A Survey. In *International Conference on Hardware/Software Codesign and System Synthesis (CODES)*. ACM, 8:1–10. https://doi.org/10.1145/2968456.2974004

[45] Takanori Sasaki, Nobukazu Yoshioka, Yasuyuki Tahara, and Akihiko Ohsuga. 2014. Evaluation of Flexibility to Changes Focusing on the Variable Structures in Legacy Software. In *Joint Conference on Knowledge-Based Software Engineering (JCKBSE)*. Springer, 252–269. https://doi.org/10.1007/978-3-319-11854-3_22

[46] Cornelia Schwennen. 2015. *Untersuchungen zur Anwendbarkeit der Isofluran-narkose bei der Ferkelkastration sowie deren Auswirkung auf Produktionsparameter in der Ferkelerzeugung unter konventionellen Produktionsbedingungen.* Ph.D. Dissertation. Tierärztliche Hochschule Hannover.

[47] Steven She and Thorsten Berger. 2010. *Formal Semantics of the Kconfig Language.* Technical Report. University of Waterloo.

[48] Gary Stringham. 2009. *Hardware/Firmware Interface Design: Best Practices for Improving Embedded Systems Development.* Newnes.

[49] Deutscher Tierschutzbund. 2019. *Verbot der betäubungslosen Kastration von männlichen Saugferkeln: Bewertung der aktuell diskutierten Alternativen aus Tierschutzsicht.* Position Paper.

[50] Eberhard von Borell, M. Oliver, B. Fredriksen, Sandra Edwards, and Michel Bonneau. 2008. Standpunkte, Praktiken und Kenntnisstand zur Ferkelkastration in Europa (PIGCAS): Projektziele und erste Ergebnisse. *Journal für Verbraucherschutz und Lebensmittelsicherheit* 3, 2 (2008), 216–220.

[51] B. Walker, N. Jäggin, M. Doherr, and U. Schatzmann. 2004. Inhalation Anaesthesia for Castration of Newborn Piglets: Experiences with Isoflurane and Isoflurane/N_2O. *Journal of Veterinary Medicine Series A* 51, 3 (2004), 150–154. https://doi.org/10.1111/j.1439-0442.2004.00617.x

[52] Jens H. Weber, Anita Katahoire, and Morgan Price. 2015. Uncovering Variability Models for Software Ecosystems from Multi-Repository Structures. In *International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS).* ACM, 103–108. https://doi.org/10.1145/2701319.2701333

[53] Gang Zhang, Liwei Shen, Xin Peng, Zhenchang Xing, and Wenyun Zhao. 2011. Incremental and Iterative Reengineering towards Software Product Line: An Industrial Case Study. In *International Conference on Software Maintenance (ICSM).* IEEE, 418–427. https://doi.org/10.1109/icsm.2011.6080809