Delft University of Technology
Software Engineering Research Group
Technical Report Series

Safe Evolution Patterns for Software Product Lines

Nicolas Dintzner

Report TUD-SERG-2015-002

TU Delft

SERG

TUD-SERG-2015-002

Published, produced and distributed by:

Software Engineering Research Group
Department of Software Technology
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology
Mekelweg 4
2628 CD Delft
The Netherlands

ISSN 1872-5392

Software Engineering Research Group Technical Reports:
http://www.se.ewi.tudelft.nl/techreports/

For more information about the Software Engineering Research Group:
http://www.se.ewi.tudelft.nl/

Note: Accepted for publication at the International Conference on Software Engineering for its Doctoral Symposium (ICSE 2015)

Safe Evolution Patterns for Software Product Lines

Nicolas Dintzner
Software Engineering Research Group,
Delft University of Technology, Netherlands
Email: N.J.R.Dintzner@tudelft.nl - Web page: http://swerl.tudelft.nl/bin/view/NicolasDintzner/WebHome

***Abstract*—Despite a global recognition of the problem, and massive investment from researchers and practitioners, the evolution of complex software systems is still a major challenge for today's architects and developers. In the context of product lines, or highly configurable systems, variability in the implementation and design makes many of the pre-existing challenges even more difficult to tackle. Many approaches and tools have been designed, but developers still miss the tools and methods enabling safe evolution of complex, variable systems.**

In this paper, we present our research plans toward this goal: making the evolution of software product lines safer. We show, by use of two concrete examples of changes that occurred in Linux, that simple heuristics can be applied to facilitate change comprehension and avoid common mistakes, without relying on heavy tooling. Based on those observations, we present the steps we intend to take to build a framework to regroup and classify changes, run simple checks, and eventually increase the quality of code deliveries affecting the variability model, mapping and implementation of software product lines.

I. Problem Statement and Objective

Software evolution has been extensively studied over the past decades, highlighting that the process remains, even today, a source of concerns for architects and designers. In the context of software product lines (SPLs), additional constraints emerging from the required variability are imposed on components or subsystems, adding one more layer of complexity to the problem at hand and forcing engineering practices to be adapted [1]–[3]. Even with this additional complexity, product line engineering practices are the approaches of choice when aiming for efficient re-use and increased quality of families of related products [4].

To assist engineers in dealing with the increased complexity and variability related concerns when evolving SPLs, specific approaches have been devised. Several researchers focused on the study of the evolution of feature models (FMs), a formalisation of features (units of variability) and their constraints, and the validation of feature changes in the context of the FM itself [5]–[7]. Others focused on the validation of the implementation, and its correctness, by extending known techniques and adapt them to the product-line context, yielding variability-aware or family-based analysis methods [8], [9]. Finally, some approaches focused on the extraction of variability information from implementation artefacts, enabling the verification of the consistency between the feature constraints as stated in the FM and its implementation [10], [11]. Recently, Passos et al. [12] described common evolution patterns observed in the Linux kernel, a large scale, open source software product line, showing that change operations often affect the Linux feature model, the mapping between feature and implementation, and the implementation simultaneously. While tools focusing on FMs or implementation are necessary, each on its own is not sufficient as changes affect all of the three variability spaces (variability model, mapping, and implementation). Moreover, such tools and techniques are computationally and memory intensive, and despite their level of sophistication, there are known corner cases and some errors can remain undetected.

The main objective of this PhD project is to devise a tool supported approach facilitating the integration of artefact deliveries affecting the FM, the mapping, and the implementation of a SPL. We start this paper by provide a glimpse of what can be achieved in terms of change synthesis and change verification by use of two concrete examples of changes that were performed on the Linux kernel. Then, we describe how we intend to reduce the amount of faulty patches delivered during the development process of SPLs by enabling lightweight and targeted validation of complex changes. We continue by providing an outline of our methodology to reach this goal with its intermediate steps. Finally, we conclude with an overview of the current status and highlight the expected contributions of this project, ending in 1.5 years.

II. Motivating Examples

We present here two evolution scenarios that took place during the maintenance of the Linux kernel. We use the first scenario to illustrate how claims made by a developer can be validated by inspecting the changes contained in a commit. With the second scenario, we show that simple heuristics could have avoided a bug. The two scenarios describe a common type of change, namely the addition of features to the Linux FM and their implementation.

A. Opportunity for Change Synthesis

Let us consider the Scenario 1, shown on the left hand side of Figure 1, the addition of a non-modular feature that was studied during the search for pattern in [12]. A new optional feature is added to the Linux kernel FM (CMDLINE_BOOL, highlighted in green). This feature is non-modular, it is not associated with any specific compilation unit, and the mapping between features and code (in Makefiles), has not been changed. When CMDLINE_BOOL is selected, the behaviour encoded in the file `setup.c` will be modified. Note that `setup.c` is unconditionally compiled if the X86 architecture is chosen. The mapping defined in Makefiles ties

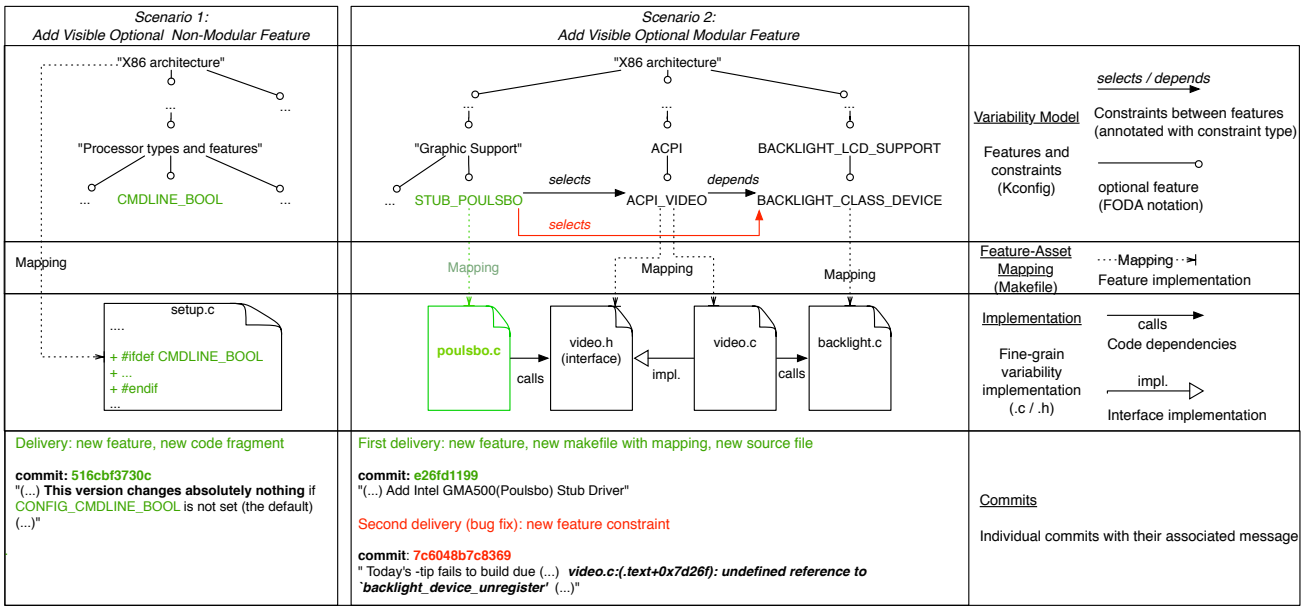

Fig. 1: Illustration of the co-evolution of the Linux FM, the mapping between its features and compilation units, and its implementation, by example of the addition of a non-modular feature (scenario 1) and a modular feature (scenario 2) to the kernel.

the X86 feature to the `setup.o` compilation unit. Finally, the developer claims that the added feature does not modify the pre-existing behaviour when not selected.

Manually analysing `setup.c`, we find in the implementation a new #ifdef statement imposing a compilation condition on the presence of the new feature. We also notice that no #elseif statement nor conditions on the absence of the new feature can be found. Finally, the only lines of code that have been modified are guarded by #ifdef statements. We infer from this that the claim made by the developer is true: if the new feature is not selected, the behaviour is preserved. We believe that this can be checked automatically.

For such a commit, we can generate a high-level view of the change, summarising deliveries touching the Linux kernel FM. We can classify this change using the patterns proposed by Passos et al. [12], by determining which spaces are modified and obtaining basic information about the use of the new feature in each space. For Scenario 1, we can say that the change is the addition of a non-modular feature, it modifies the implementation of a behaviour of the feature X86 (`setup.c`). Using the heuristic presented above, we can add that the change is neutral when the new feature is not selected. This last information is important to understand the nature of the change and when choosing configurations to test.

B. Feature Dependencies

In Scenario 2, depicted on the right hand side of Figure 1, a new driver for a media graphic accelerator is added. This addition later caused a bug, which was captured in the variability bug database created by Alba et. al [13]. The driver can be included in the kernel during the compilation process by selecting the feature STUB_POULSBO, mapped to the file `poulsbo.c` by a new Makefile. The implementation of the driver uses generic libraries associated with the feature ACPI_VIDEO. This dependency is shown in the Linux FM, by the "select" constraint between STUB_POULSBO and ACPI_VIDEO, and by the "call" relationship between the files `poulsbo.c` and `video.h`. The ACPI_VIDEO feature itself depends on the feature BACKLIGHT_CLASS_DEVICE (BCD for short), and this dependency, described in the FM by the "depends" constraints between them, matches a code dependency (a function call) between the implementation of ACPI_VIDEO (`video.c`) and the implementation of BCD.

In the Linux kernel FM, a "depends" statement guarantees that a feature cannot be selected as long as its dependencies are not satisfied. Opposed to this, the "select" statement guarantees that when this feature is selected, all targeted features are included as well. However a "select" statement does not guarantee that the dependencies of the selected target are satisfied. Here, the selection of STUB_POULSBO properly includes its own implementation and the interfaces it relies on. But the code mapped with the dependencies of ACPI_VIDEO is **not** included, and the file `video.c` does **not** compile (hence the bug fix included in the second commit, highlighted in red in Figure 1). The second commit corrects this error by adding a second "select" statement on STUB_POULSBO targeting BCD, forcing the inclusion of its implementation and satisfying the dependencies of ACPI_VIDEO.

By looking at the hierarchy of the new feature, and the hierarchy of the selected target, one can see that this situation is problematic, and this without introspecting the entire model. If we were to generate a change report for the initial commit, we would be able to provide the following information. This change can be classified as the addition of a new modular feature (STUB_POULSBO is associated with a new compilation unit). We include a warning regarding the unsafe usage of the

select statement. Finally, we can suggest a partial configuration of the kernel which could reveal the potential error that we detected: a configuration containing STUB_POULSBO and ACPI_VIDEO but not BCD (or one of the 4 features on which ACPI_VIDEO depends). Despite the apparent simplicity of this bug, it is interesting to note that errors caused by faulty dependencies among features are frequent [13].

III. Methodology

In the two examples presented above, the information extraction and the verification of dependencies was done using partial information about the Linux FM and its implementation. The modified implementation artefacts guided us both in deciding what to verify and how to perform the verification. The analyses done in those examples are experience-based, so probably incomplete, but are sufficient to create additional knowledge that could have been useful to the developers at the time. Those three aspects, partial information, local information, and heuristics-based verifications are key elements in the efficient analysis of changes performed in SPLs.

During this PhD project, we will determine the extent to which such reasoning can be automated, and help in making each patch safe with respect to known bugs. We aim at leveraging their apparent simplicity to create holistic views of complex changes. The target is to build change reports that will support developers with targeted implementation verifications of known pitfalls, either when delivering new code or when integrating patches. We know that changes are not done in a single commit, so given a set of commits, we need to regroup related commits based on the feature(s) that they impact. This defines which artefact changes should investigated. Then, we match each commit group with a known evolution pattern, giving us an abstract description of the change and several of its expected characteristics. Finally, we can choose the information to extract and verification heuristics to apply.

By using heuristics, we aim at lowering the cost of validation imposed by formal approaches. Additionally, we will rely as much as possible on local information - local with respect to the changes that occurred, guaranteeing that the information we use is familiar to the developer performing the changes; facilitating the interpretation of our results. Finally, the solution should impose as little overhead as possible to developers during their development process. To reach this goal, several research challenges must first be met.

Gathering changes. Our first step will be to regroup related commits participating in the evolution of a SPL. But this is non-trivial. How does one regroup changes affecting heterogeneous implementation artefacts? In the case of Linux, the feature model is described in the Kconfig format, the Makefiles contain shell scripts and the source code is mostly in C. Moreover, Linux is hosted in a Git repository. How does one define a relevant commit window in which to search for related commits, in a system heavily relying on parallel branches, and where time is a relative thing [14]? By answering those questions, we obtain the means to regroup related commits, and gather knowledge about the delivery habits of developers working on SPLs.

From changes to patterns. Once we obtained an efficient and reliable way of regrouping commits into consistent changes, we can classify them according to known co-evolution patterns. In this context, no such approach exists yet. We need to answer the following questions: how should each pattern be describe so that we can identify their instances given a set of modified files? how can this be applied systematically over large sets of commit groups? The representation of the pattern that we will use here should be sufficient for us to derive rules that can be implemented in a tool enabling automatic identification of pattern instances. This representation might itself be useful for further automation, or to describe changes occurring in different contexts.

Checking pattern properties. The core of work will be done in this last step, consisting in identifying the heuristics to use with each pattern. Our initial choice of checks will target common bugs occurring in SPLs [13]. The challenges that we will face here can be summarised by the following questions: What characteristics of a change should be checked in a given context? How do the results of an heuristic-based approach compare to existing techniques? By answering those questions, we refine our knowledge on the relevant properties of change operations in SPLs, and obtain a basis to implement validation heuristics. For this step, we will consider adapting existing tools, tuning them to work with partial and local information.

To answer those questions, we will conduct several case studies on open-source, and possibly industrial, SPLs. We will build our approach first for the Linux kernel: regrouping commits, identifying instances of frequently used patterns, and performing a minimal yet relevant set of checks inspired by know variability related bugs [13], and bug fixes captured during the manual analysis performed during the discovery of co-evolution patterns [15]. Then, we will refine our approach to match more patterns and run checks for more types of bugs. While we focus on SPLs, our approach should be applicable to any highly variable system with an explicit variability model (such as aXTLS or SeaBios). Each case study will be performed on multiple systems, to guarantee the generalizability of our results. Each step will give us the opportunity to advance toward our objective, and allow us to formalise detailed knowledge on SPL evolution which will be compiled as the last output of this project.

IV. Validation Strategy

Each step will require its own validation, but to prove that our approach is useful, we will build a tool automating the different steps described above. We will start by building a prototype and incrementally augmenting its capabilities as we progress through our research. We can compare the output of our tool with the manual analysis that was performed in the context of [12], both grouping commits, and validating the matching between changes and patterns. Additional manual analysis will be done, but we can rely on this information as a reference set. Once we start building heuristics, we will

compare our approach with existing tools; using Undertaker [11] to identify dead code block, and TypeChef [8] to check for dangling dependencies for instance. The last validation step of our work will consist in surveys and interviews to gather feedback from developers on the value of the information that we can provide them with.

V. Progress

My PhD research started in March 2012, under the supervision of Prof. Dr. Arie van Deursen and Prof. Dr. Martin Pinzger. We started by studying the evolution of SPLs from a feature perspective, specifically on the Linux kernel [16] and in the context of an industrial product line [7]. Through this work, we obtained a detailed understanding of feature changes and their impact in two different contexts, and means to capture and analyse them. We extended our work on the Linux kernel, obtaining further knowledge on the inner workings of its FM and its evolution [17]. More recently, I cooperated with Leonardo Passos on the extension of his work on co-evolution pattern identification [15]. My work so far resulted in the following three publications [7], [16], [17], and my participation in a fourth [15].

Our next concrete milestone is in early 2015 and is concerned with regrouping related commits affecting features of the Linux kernel. In the meantime, I will continue to collaborate with Leonardo Passos, to generalise the co-evolution patterns already identified. The next steps will be completed by the Spring of 2016 and will result in a thesis compiling our findings on the safe evolution of large scale SPLs.

VI. Expected Contributions

Our end goal is to facilitate the maintenance and evolution of complex, highly variable systems. By following the steps presented in this paper, we expect to make the following contributions: (1) A theory on the feature-oriented evolution of SPLs; (2) Tools and methods to extract information from implementation patches, both for implementation verification and change comprehension; (3) A repository mining methodology to regroup changes affecting heterogeneous implementation artefacts using domain level information (features in this case); (4) An approach to match evolution patterns to file changes and its application to the Linux kernel. Finally, we will make all tools and datasets built for this work publicly available, both for the reproducibility of our experiments and for researchers to use them as they see fit.

Despite our best effort, it will not be possible for us to cover all known co-evolution patterns nor all types of checks that would guarantee the safety of a feature related change. Yet, we believe that this work will provide to researchers and practitioners the tools and knowledge necessary to fully understand the potential benefits and limitations of a feature-oriented, pattern-based approach to support the evolution of complex highly variable systems.

Acknowledgement

This research is conducted under the supervision of Prof. Martin Pinzger (martin.pinzger@aau.at) and Prof. Arie van Deursen (Arie.vanDeursen@tudelft.nl). This work is supported by the Dutch national program COMMIT and carried out as part of the Allegio project under the responsibility of the Embedded Systems Innovation group of TNO in partnership with Philips Healthcare.

References

[1] H. Schirmeier and O. Spinczyk, "Challenges in software product line composition," in *Proc. of the 42nd Hawaii International Conference on System Sciences*, HICSS '09, IEEE, Jan. 2009.

[2] J. Bosch, *Design and Use of Software Architectures: Adopting and Evolving a Product-line Approach*. Pearson Education, 2000.

[3] J. Bosch, G. Florijn, D. Greefhorst, J. Kuusela, J. Obbink, and K. Pohl, "Variability issues in software product lines," in *Software Product-Family Engineering* (F. van der Linden, ed.), vol. 2290 of *Lecture Notes in Computer Science*, pp. 303–338, Springer Berlin / Heidelberg, 2002.

[4] P. Clements, *Software product lines: practices and patterns*. The SEI series in software engineering, Boston: Addison-Wesley, 2002.

[5] T. Thuem, D. Batory, and C. Kaestner, "Reasoning about edits to feature models," in *Proc. of the 31st International Conference on Software Engineering*, ICSE '09, pp. 254–264, IEEE Computer Society, 2009.

[6] W. Heider, M. Vierhauser, D. Lettner, and P. Grunbacher, "A case study on the evolution of a component-based product line," in *2012 Joint Working IEEE/IFIP Conference on Software Architecture (WICSA) and European Conference on Software Architecture (ECSA)*, pp. 1 –10, Aug. 2012.

[7] (Accepted), N. Dintzner, U. Kulesza, A. Van Deursen, and M. Pinzger, "Evaluating feature change impact on multi-product line configurations using partial information," in *Proceedings of the 14th International Conference on Software Reuse*, vol. 8919 of *Lecture Notes in Computer Science*, pp. 1–16, Springer International Publishing Switzerland, 2015.

[8] C. Kastner and S. Apel, "Type-checking software product lines - a formal approach," in *23rd IEEE/ACM International Conference on Automated Software Engineering, 2008. ASE 2008*, pp. 258 –267, Sept. 2008.

[9] T. Thüm, S. Apel, C. Kästner, I. Schaefer, and G. Saake, "A classification and survey of analysis strategies for software product lines," *ACM Comput. Surv.*, vol. 47, pp. 6:1–6:45, June 2014.

[10] S. Nadi and R. Holt, "Mining kbuild to detect variability anomalies in linux," in *2012 16th European Conference on Software Maintenance and Reengineering (CSMR)*, pp. 107–116, 2012.

[11] R. Tartler, J. Sincero, W. Schröder-Preikschat, and D. Lohmann, "Dead or alive: Finding zombie features in the linux kernel," in *Proceedings of the First International Workshop on Feature-Oriented Software Development*, pp. 81–86, 2009.

[12] L. Passos, J. Guo, L. Teixeira, K. Czarnecki, A. Wąsowski, and P. Borba, "Coevolution of variability models and related artifacts: a case study from the linux kernel," in *Proceedings of the 17th International Software Product Line Conference*, SPLC '13, pp. 91–100, ACM Press, 2013.

[13] I. Abal, C. Brabrand, and A. Wasowski, "42 variability bugs in the linux kernel: A qualitative analysis," in *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, (New York, NY, USA), pp. 421–432, ACM, 2014.

[14] C. Bird, P. Rigby, E. Barr, D. Hamilton, D. German, and P. Devanbu, "The promises and perils of mining git," in *6th IEEE International Working Conference on Mining Software Repositories, 2009. MSR '09*, pp. 1–10, 2009.

[15] L. Passos, L. Teixeria, N. Dintzner, S. Apel, A. Wąsowski, K. Czarnecki, P. Borba, and J. Guo, "Coevolution of Variability Models and Related Artifacts: A Fresh Look at Evolution Patterns in the Linux Kernel," *Empirical Software Engineering*, 2015. To appear.

[16] N. Dintzner, A. Van Deursen, and M. Pinzger, "Extracting feature model changes from the linux kernel using FMDiff," in *Proceedings of the Eighth International Workshop on Variability Modelling of Software-Intensive Systems*, VaMoS '14, ACM Press, 2013.

[17] (Submitted), N. Dintzner, A. Van Deursen, and M. Pinzger, "Analyzing the linux kernel feature model changes using FMDiff," *Software & Systems Modeling - special issue on Variability Management*.

TUD-SERG-2015-002
ISSN 1872-5392