A Study of Variability Spaces in Open Source Software

Sarah Nadi
David R. Cheriton School of Computer Science
University of Waterloo, ON, Canada
snadi@uwaterloo.ca
http://swag.uwaterloo.ca/~snadi

***Abstract*—Configurable software systems allow users to customize them according to their needs. Supporting such variability is commonly divided into three parts: configuration space, build space, and code space. In this research abstract, we describe our work in exploring what information these spaces contain in practice, and if this information is consistent. This involves investigating how these spaces work together to ensure that variability is correctly implemented, and to avoid any inconsistencies or anomalies. Our work identifies how variability is implemented in several configurable systems, and initially focuses on less studied parts such as the build system. Our goals include: 1) investigating what information each space provides, 2) quantifying the variability in the build system, 3) studying the effect of build system constraints on variability anomalies, and 4) analyzing how variability anomalies are introduced and fixed. Achieving these goals would help developers make informed decisions when designing variable software, and improve maintainability of existing configurable systems.**

***Index Terms*—Software Variability, Variability Anomalies, Linux, Mining Software Repositories**

I. Introduction

A configurable system (e.g., the Linux kernel) allows users to select the features they are interested in, and to compile the system with this specific *configuration*. Thus, different *variants* of the system can be generated from the same code base according to the user's selected configration. Such systems usually have some form of model that describes the features supported by the system, and the dependencies and constraints between these features. Some mapping scheme is then needed to map this feature selection to the desired areas in the code such that the corresponding variant can be correctly generated. Although other terms have been used to describe these parts of the system in related research areas (e.g., feature model [1], problem/solution space [2]), we use the terms *configuration space*, *build space*, and *code space* to describe the three information *spaces* that support variability in a system since they are more descriptive for our research purposes. The configuration space specifies the features supported by the system, and their dependencies. The build space controls the compilation process such that the user's feature selection is mapped into the correct source files which are then compiled and linked into the final product. The code space contains the source code implementing the system's supported functionality.

Each variability space contains dependencies and constraints that govern the variability in the system. The three spaces may be scattered across the system in different types of artifacts (e.g., Linux's C files and Makefiles) or may be combined together (e.g., eCos's CDL files). In either case, the constraints enforced by these three spaces must be clear and consistent. Previous work has typically studied one of these spaces in isolation [3], [4] or in terms of how two of the spaces co-evolve [5], [6]. However, there has not been an attempt to comprehensively study the variability of the three combined. Such a study is important to enable large scale variable systems to be easily maintainable while preserving their configurability. To comprehensively study software variability, we need to analyze the overlap between the spaces, as well as any conflicts between them. Our goal is to provide such a comprehensive view of variability in open source software.

In this work, we study existing configurable open-source software to analyze how variability is implemented in practice, and what information each of the three spaces actually provides. Such information can help in designing new configurable systems, and improving the maintainability of existing ones. Our study of variability spaces proposes the following:

1) Evaluating the commonality and differences in the variability information provided by the three spaces.
2) Studying variability in build systems since they play an important role in supporting variability, but have not been commonly studied in that context.
3) Combining variability constraints from all three spaces to detect anomalies in the system.
4) Performing an origin study of variability anomalies.

To achieve these goals, we choose the Linux kernel as our main case study since it is one of the biggest configurable open source software systems available with over 12,000 configurable features. Whenever applicable, we plan to study other configurable systems to verify our findings. In this research abstract, we describe our work in terms of the Linux kernel for simplicity and consistency.

We present our research as follows. Section II gives related background information about Linux. Section III discusses our approach, as well as our current progress. Section IV highlights key benefits and contributions of this work. Section V presents related research, and Section VI concludes this abstract.

978-1-4673-3076-3/13 © 2013 IEEE

ICSE 2013, San Francisco, CA, USA
Doctoral Symposium

Accepted for publication by IEEE. © 2013 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

II. Variability in the Linux Kernel

There are three artifacts that control variability in Linux which correspond to the three variability spaces described above: KCONFIG files (configuration space), KBUILD Makefiles (build space), and source code files (code space). KCONFIG files describe the various features of the Linux kernel. They specify configuration options and their interdependencies. Detailed description of the KCONFIG files, and how they describe Linux's feature model can be found in other work [5]. KBUILD Makefiles indicate which source files get compiled according to the user's selection. Through a special notation [7], [8], the Makefiles specify which KCONFIG feature(s) need to be selected for a particular file to compile. The source code in Linux (e.g., C files) implements the functionalities of the kernel, and uses C preprocessor (CPP) directives to provide conditional compilation according to the user's selection. That is, some parts of the code will only get compiled if certain kconfig features are chosen. More information about the Linux build process can be found in our previous work [7].

III. Approach and Research Questions

Figure 1 shows how we divide our work into four main parts depicted as research questions. For each question, we explain its goal, the approach followed, and our current status.

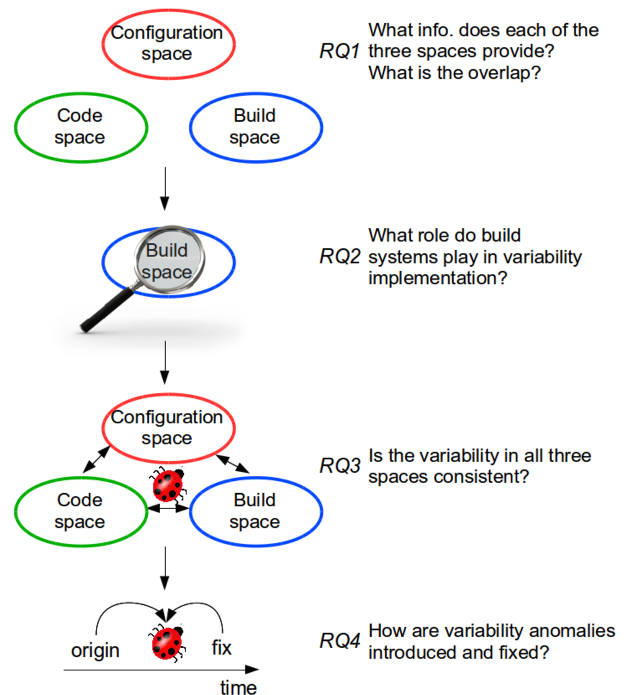

Fig. 1. Overview of our approach.

A. RQ1: What information does each of the three spaces provide, and what is the overlap between spaces?

Goal and Motivation. Each of the three variability spaces has constraints that control the possible variants generated by the system. However, it is not clear what is the overlap between the spaces. Is it enough to analyze the configuration space only? Is analyzing the code sufficient? Or do we always need to analyze all aspects of the system. We aim to explore where the constraints in the configuration space are reflected in the code and build files, and whether one is an over estimation of the other. We anticipate facing some challenges such as accurately extracting and calculating all variability constraints inthe three spaces including non-boolean constraints.

Approach. We plan to analyze several configurable software systems. We will first extract the constraints in each of the three spaces using existing tools (e.g., [4], [9]). We then plan to compare the constraints in the three spaces to find commonalities and overlaps, and identify common cases where the constraints in the variability model are reflected in the code or not. Such work will also uncover the limits of static analysis techniques to extract variability information from the code.

Status. This part of our work is currently in progress.

B. RQ2: What role do build systems play in variability implementation?

Goal and Motivation. During our initial investigation of software variability, we found that the build system contains important variability information, and yet its role has not been thoroughly studied. Our goal here is to understand how the build system contributes to variability support.

Approach. We choose the Linux kernel as a case study. We study Linux's documentation, and examine its build system (KBUILD) files. To understand the variability in KBUILD, we need to parse the Makefiles to extract the constraints that specify which feature(s) each source file depends on. This extraction also allows us to quantify the build system variability by counting the number of features used in the build system versus those used in the code. The size of the constraints in the build system also reflects the complexity of variability supported there. We plan to analyze all architectures of the Linux kernel, and perform this analysis on several releases.

Status. Statically extracting the variability constraints from KBUILD is challenging since it uses a complicated syntax, and a recursive build system. We have developed a Makefile constraint extractor that approximates the constraints controlling source file compilation in KBUILD [8]. We have performed a quantitative analysis of the variability in KBUILD [10] using a set of metrics adapted from Liebig et al. [11]. We found that on average, KBUILD uses 63% of the features supported in Linux, and that 46% of these features are exclusively used in KBUILD to control source file compilation which indicates that KBUILD is a major part of the variability implementation in Linux. However, we found that the constraints in KBUILD are not very complex, and that the compilation of each file is usually controlled by a single feature.

Future Work. We plan to do a deeper investigation of our results by exploring the patterns of feature usages. For example, this includes understanding the nature of features that are only used in the build space or only used in the code space. We also plan to quantify variability in other systems (e.g., ECOS).

C. RQ3: Is the variability in all three spaces consistent?

Goal and Motivation. When variability implementation is scattered over different parts of the system, inconsistencies are inevitable. Previous work by Tartler et al. [12] detected many dead and undead CPP guarded parts of the code due to inconsistencies between the constraints in the configuration and code spaces. However, the effect of the build system has not been studied. Our goal here is to include KBUILD in the anomaly detection process, and determine its effect.

Approach. We choose Linux as a case study. We first look at variability anomalies within the build system itself, and then look at how adding KBUILD constraints to the analysis affects the variability anomalies detected. We call the first category of anomalies, *syntactic variability anomalies* since they are more related to the way KBUILD has been setup, while we refer to the second category as *semantic variability anomalies* since they arise from conflicts in constraints. We develop heuristics to identify syntactic anomalies. We then extend Tartler et al.'s [12] work, which uses SAT solvers to detect inconsistencies, by including the build space constraints extracted from our Makefile constraint extractor in the previous step. Our approach includes studying several releases of the Linux kernel, and comparing the anomalies detected with and without considering the KBUILD constraints. We also track the evolution of these anomalies across the releases.

Status. We have developed heuristics to find three types of syntactic variability anomalies in Linux [7]. We performed a longitudinal study over several releases of the Linux kernel, and demonstrated that these anomalies get introduced and fixed over time. We have also extended the UNDERTAKER [12] tool to include the constraints we extracted from KBUILD Makefiles, and have shown that including the constraints from KBUILD allows more variability anomalies to be detected [8].

Future Work. We plan to further validate these findings on other configurable systems such as BusyBox, eCos etc.

D. RQ4: How are variability anomalies introduced and fixed?

Goal and Motivation. Addressing RQ3 establishes that variability anomalies do exist in Linux which indicates that it would be beneficial to have tools that automatically detect these anomalies and suggest fixes for them. It would even be more beneficial to have these tools be proactive such that they detect the change that introduces this anomaly and report it right away. To provide such intelligent maintenance support, we need to first understand how these anomalies get introduced in the first place, and how developers usually fix them.

Approach. We focus on the Linux kernel, and divide our approach into two parts. We first determine the patterns of causes and fixes to look for through an exploratory study of a set of existing patches that were submitted to Linux kernel developers to solve previously detected variability anomalies by Tartler et al. [12]. We focus on referential variability anomalies which are anomalies that arise due to a mismatch between the features used in the code in Linux, and those defined in KCONFIG. After determining these patterns, we need to perform a confirmatory case study on several releases of the Linux kernel to determine if the patterns we found for causes and fixes hold. We look for commits in Linux's GIT repository that may be possible causes and fixes of variability anomalies. Automatically determining potential causes and fixes is challenging. We develop heuristics to identify such commits that are based on the nature of the patch in the commit, as well as its temporal relation to the detected anomaly.

Status. We performed both the exploratory case study and the confirmatory one. Our results are currently under review. Our findings suggest that KCONFIG changes often have wide cross-cutting effects on the code that are not immediately detected, and must often be fixed by subsequent code changes.

Future Work. The study we have performed so far focused on referential variability anomalies (i.e., those caused by using undefined of incorrect KCONFIG features). Extending this work to include other types of anomalies that are directly caused by conflicts in constraints would be interesting.

IV. Expected Benefits

Answering the above research questions allows us to provide support to developers in designing, analyzing and maintaining configurable software. Specifically, we foresee the following benefits and contributions to the outcome of our work.

1) Guiding future analysis of variable software by determining the overlaps between different spaces in variability implementation.
2) Decreasing variability anomalies in software systems.
3) Improving the maintainability of variable software.
4) Guiding the design and implementation of software systems supporting variability.

V. Related Work

A. Software Variability and Variability Anomalies

The Linux kernel has been one of the main subjects of variability research due to its large size and number of supported features. The work on Linux (as well as other systems) has mainly focused on analyzing variability information in the source code [4], or in the configuration files [3], [5]. By analyzing the constraints from both the source code and KCONFIG, Tartler et al. [12] detected variability anomalies in Linux. They also provided patches to fix some of these detected variability anomalies. However, that work did not include the constraints from KBUILD in the analysis. Additionally, to the best of our knowledge, there has not been any comprehensive work done to understand how these variability anomalies get introduced, and how they eventually get fixed. Studying the origin of these variability anomalies is important in order to provide tools that support more proactive anomaly prevention.

B. Build System Variability

McIntosh et al. [13] and Adams et al. [6] studied build files and their evolution in Java systems and MAKE based systems. They analyzed the dependencies between build targets, but did not study the configuration features that appear in the build files, and how they contribute to the overall variability of the system. Part of their findings showed that the build

system's complexity grows over time in terms of its size, and the number of targets it supports. Such results suggest that studying build systems is important and that they consume a fair amount of maintenance effort. Since such work does not include variability information, and only focuses on static code dependencies between source files, further analysis of build systems from a variability perspective is needed.

It is our understanding that Berger et al. [14] were the first to discuss build system variability. They showed that the extraction of the presence conditions of source code files from Makefiles is feasible. When analyzing Linux, they only look at the x86 architecture, while the quantitative analysis we perform for KBUILD is based on all Linux CPU architectures over several releases, and not solely on the x86 architecture of one release. This provides a more thorough analysis of variability in KBUILD. We also show the effect of these constraints on the variability of the final compiled kernel image through the variability anomalies we detect.

To the best of our knowledge, our work [8] was the first to analyze the effect of variability in KBUILD on anomalies. Recently, the UNDERTAKER team analyzed KBUILD [15], and developed their own Makefile parser, GOLEM [16], which uses a dynamic probing mechanism. The goal of our work is not to determine the most accurate parsing for Makefiles, but rather to clarify the role of the build system in variability implementation so that it is recognized in future variability research.

C. Bug-Introducing Changes

It has been commonly believed that changes to the code often introduce bugs. Our finding that KCONFIG changes often cause variability anomalies aligns with this belief. Identifying *bug-introducing* changes is a well researched topic (e.g., [17], [18], [19]). Most of the techniques used are based on training models that are used to predict future bug introducing changes. We adopt a different approach since there has not been much work in determining the causes and fixes of variability anomalies. Thus, we first perform an exploratory study to determine criteria for finding causes and fixes of variability anomalies tailored to Linux's variability implementation. Our work is also different in that we do not only focus on source code changes, but rather we relate changes in one part of the system (KCONFIG) to anomalies in a different part (source code).

VI. CONCLUSION

Our proposed work aims to provide better support for designing and maintaining configurable software. By analyzing variability implementation from various angles, we provide better anomaly detection mechanisms, and facilitate future variability analysis by clarifying the origin and implications of variability constraints in different parts of the implementation. We do this through studying several configurable systems, with Linux being our main case study. Our work has so far helped in uncovering existing variability anomalies in Linux, and providing patterns for the causes and fixes of such anomalies.

ACKNOWLEDGMENTS

This work is under the supervision of Ric Holt.

REFERENCES

[1] K. Kang, S. Cohen, J. Hess, W. Novak, and A. Peterson, "Feature-oriented domain analysis (FODA) feasibility study," CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, Tech. Rep., 1990.

[2] K. Czarnecki and E. Ulrich, *Components and Generative Programming*, ser. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 1999, vol. 1687, pp. 2 – 19.

[3] T. Berger, S. She, R. Lotufo, A. Wasowski, and K. Czarnecki, "Variability modeling in the real: a perspective from the operating systems domain," in *ASE '10: Proceedings of the IEEE/ACM international conference on Automated software engineering*. ACM, 2010, pp. 73–82.

[4] C. Kästner, P. Giarrusso, T. Rendel, S. Erdweg, K. Ostermann, and T. Berger, "Variability-aware parsing in the presence of lexical macros and conditional compilation," in *OOPSLA '11: Proceedings of the 2011 Conference on Object-Oriented Programming Systems, Languages, and Applications*, 2011.

[5] R. Lotufo, S. She, T. Berger, K. Czarnecki, and A. Wasowski, "Evolution of the Linux kernel variability model," in *SPLC'10: Proceedings of the 14th International Conference on Software Product Lines: Going Beyond*. Springer-Verlag, 2010, pp. 136–150.

[6] B. Adams, K. De Schutter, H. Tromp, and W. De Meuter, "The evolution of the Linux build system," in *EVOL '07: Proceedings of the 3rd International ERCIM Symposium on Software Evolution*, vol. 8, no. 0, 2007.

[7] S. Nadi and R. Holt, "Make it or break it: Mining anomalies in Linux Kbuild," in *WCRE '11: Proceedings of the 18th Working Conference on Reverse Engineering*, 2011.

[8] ——, "Mining Kbuild to detect variability anomalies in Linux," in *CSMR '12: Proceedings of the 16th European Conference on Software Maintenance and Reengineering*, March 2012, pp. 107 –116.

[9] S. She, R. Lotufo, T. Berger, A. Wasowski, and K. Czarnecki, "Variability model of the linux kernel," in *VaMoS 2010: Proceedings of the 4th International Workshop on Variability Modeling of Software-intensive Systems*, 2010.

[10] S. Nadi and R. Holt, "The linux kernel: A case study of build system variability," *Journal of Software: Evolution and Process*, 2013, *(Accepted to appear)*.

[11] J. Liebig, S. Apel, C. Lengauer, C. Kästner, and M. Schulze, "An analysis of the variability in forty preprocessor-based software product lines," in *ICSE '10: Proceedings of the 32nd International Conference on Software Engineering*, vol. 1, may 2010, pp. 105 –114.

[12] R. Tartler, D. Lohmann, J. Sincero, and W. Schröder-Preikschat, "Feature Consistency in Compile-Time Configurable System Software," in *Proceedings of the EuroSys 2011 Conference (EuroSys '11)*, G. Heiser and C. Kirsch, Eds., New York, NY, USA, 2011, pp. 47–60.

[13] S. McIntosh, B. Adams, and A. Hassan, "The evolution of Java build systems," *Empirical Software Engineering*, pp. 1–31, 2011.

[14] T. Berger, S. She, R. Lotufo, K. Czarnecki, and A. Wasowski, "Feature-to-code mapping in two large product lines," in *SPLC'10: Proceedings of the 14th International Software Product Line Conference. Poster Session*. Springer Berlin/Heidelberg, 2010.

[15] C. Dietrich, R. Tartler, W. Schröder-Preikschat, and D. Lohmann, "Understanding Linux feature distribution," in *AOSD-MISS '12: Proceedings of the 2nd AOSD Workshop on Modularity in Systems Software*, 2012.

[16] C. Dietrich, R. Tartler, W. Schröder-Preikschat, and D. Lohmann, "A robust approach for variability extraction from the Linux build system," in *SPLC '12: Proceedings of the 16th International Software Product Line Conference (to appear)*, 2012.

[17] S. Kim, T. Zimmermann, K. Pan, and E. Whitehead, "Automatic identification of bug-introducing changes," in *ASE '06: Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering*, Sept. 2006, pp. 81 –90.

[18] L. Aversano, L. Cerulo, and C. Del Grosso, "Learning from bug-introducing changes to prevent fault prone code," in *IWPSE '07: Proceedings of the 9th International workshop on Principles of software evolution: in conjunction with the 6th ESEC/FSE joint meeting*. New York, NY, USA: ACM, 2007, pp. 19–26.

[19] J. Śliwerski, T. Zimmermann, and A. Zeller, "When do changes induce fixes?" in *MSR '05: Proceedings of the 2nd International Workshop on Mining Software Repositories*, ser. MSR '05. New York, NY, USA: ACM, 2005, pp. 1–5.