Towards Automated Configuration Documentation

Jobayer Ahmmed
Iowa State University
Ames, Iowa, USA
jobayer@iastate.edu

Myra B. Cohen
Iowa State University
Ames, Iowa, USA
mcohen@iastate.edu

Paul Gazzillo
University of Central Florida
Orlando, Florida, USA
paul.gazzillo@ucf.edu

ABSTRACT

Configurability is a common property of software allowing programs to be customized for the user. While configurability is pervasive, it can also lead to faults (or misconfigurations) and make program evolution challenging. Dependencies can be missed, essential code can be left in place when a configuration option is removed, or code can be deleted or changed when still in use by other configuration options. A key issue is a lack of sufficient documentation and traceability between configuration options and code during software evolution. Existing approaches to solve these problems include automated documentation, analysis of version control history, or the use of special program configuration management languages. However, none of these provide a sufficient solution managing configuration changes over time. In this paper we propose our vision for an automated approach called ConfiGen, which provides user-facing documentation along with a back-end analysis showing definitions and uses of configuration options along with traceability to lines of program code for evolution. We performed a case study demonstrating its potential usefulness.

CCS CONCEPTS

• **Software and its engineering** → **Software evolution**; **Documentation**; **Software product lines**.

KEYWORDS

configuration, software evolution, documentation

ACM Reference Format:
Jobayer Ahmmed, Myra B. Cohen, and Paul Gazzillo. 2024. Towards Automated Configuration Documentation. In *39th IEEE/ACM International Conference on Automated Software Engineering (ASE '24), October 27-November 1, 2024, Sacramento, CA, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3691620.3695311

1 INTRODUCTION

Software maintenance is an expensive part of the development lifecycle. Documentation plays a key role, supporting tasks such as testing and debugging, refactoring, and the design and architecture for new functionality. Yet, documentation is often lacking or hard to find. Even when developers comment individual methods and modules, without automated documentation support, valuable information lies distributed in the code. Recent approaches to documentation automation have helped to alleviate this concern by aggregating code level comments into developer usable documents with traceability back to the code. As maintainers change code they can leverage this information and ensure consistency over time. However, most of the commonly used documentation tools, such as Javadocs [20], Docstrings [13], or Doxygen [14] are monolingual, meaning they document a single language at a time. This works well for documentation of variables, methods and classes, but it leaves a hole for an important dimension of software which may be critical during maintenance, that of configurability.

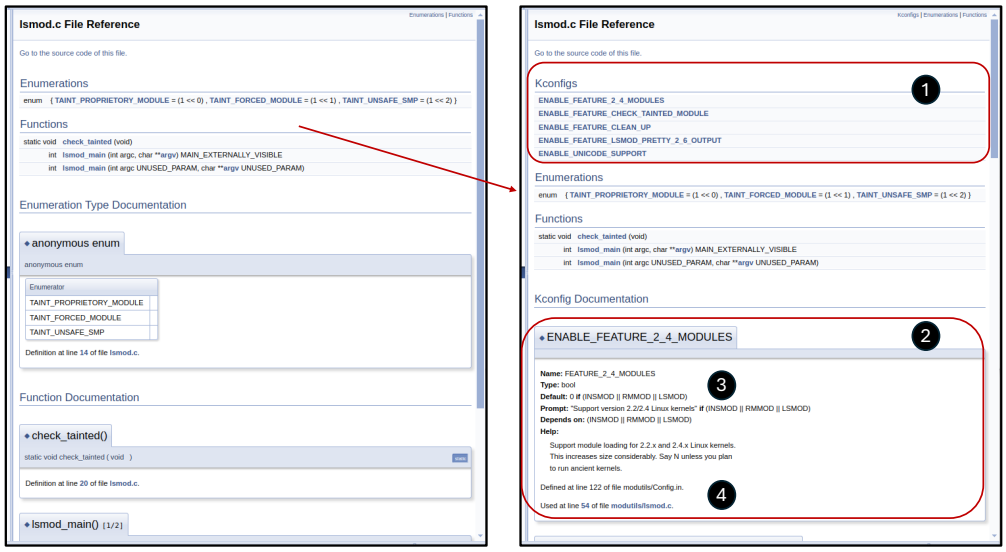
Much of our software today is configurable. Users or administrators can add and remove features to customize the software's behavior. Many software systems have hundreds or even thousands of configuration options (features which can be changed during customization). This has led to a large body of literature on misconfigurations [4, 43, 45, 47], runtime faults due to interacting features [16], and arguments for less complex configuration interfaces [42]. While reducing the complexity of configurations may help with some of these issues, most real configuration spaces are continually increasing. For instance, the Linux kernel, often used as an exemplar of configurable software, has grown from over 4,000 [19] in 2010, to more than 15,000 [30] in 2021. The Kconfig language [22] has been built to support definition and selection of configurations at build time, however, this information is isolated within the Kconfig documentation.

In recent work [18] we argued for configurations to be first class elements in a programming language, and at an abstraction level that matches the programming languages used by developers and maintainers. There has been much research on variability representation languages [1, 3, 33] and at providing configuration visualization inside of the programming Interactive Development Environment or IDE [21]), however this ignores the larger systematic problem of ensuring configurations are fundamental constructs.

In this paper we propose to make configurability a first class element in documentation. We call our approach ConfiGen. To illustrate its benefits, we build a prototype on top of a popular automated documentation tool; one that extracts user documentation from code and generates output in user readable forms. We augment this with static analysis that identifies usage of the features in the code and identifies deletions, additions and modifications of code between versions. We evaluate the potential for ConfiGen on three popular open source applications. We can generate configuration information for less than a 1.5% overhead over the base documentation automatically while documenting between 11-25% of the files in each version. These results provide evidence that (1) documentation about configurability is indeed fragmented and (2) that merging the documentation provides a cohesive view and traceability of both configuration definition and usage.

ASE '24, October 27-November 1, 2024, Sacramento, CA, USA

ACM ISBN 979-8-4007-1248-7/24/10
https://doi.org/10.1145/3691620.3695311



(a) Normal Documentation (b) ConfiGen Documentation

Figure 1: Traditional Documentation (a) with configuration information added (b)

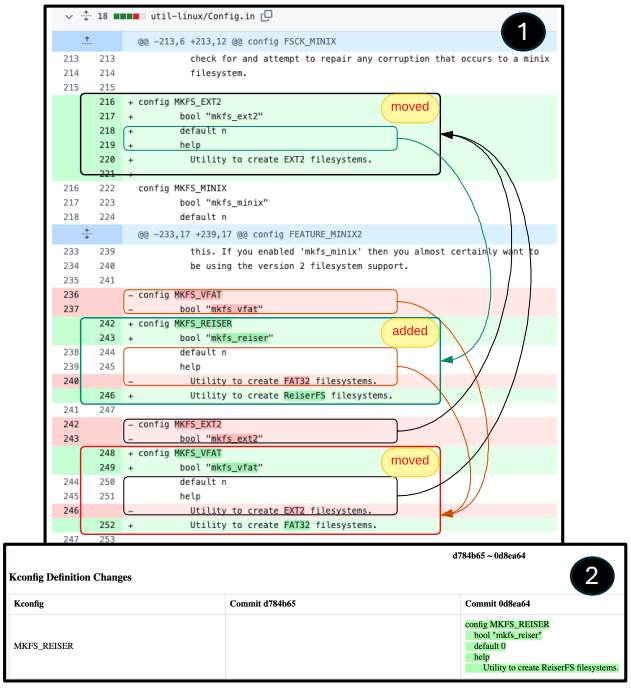


Figure 2: (1) Movement of features shown in git diff (2) ConfiGen Analysis shows only the addition of a feature in green

code and provides documentation and traceability for enumerations, classes and functions, etc. Users can configure the output to include macro information, however, it does not provide any information about configurability. While it does support expanding `#ifdef` constructs which are often used to represent blocks of code associated with features, it does not process this in any special way and information related to features is lost. On the right (b) we see documentation from our prototype tool that includes configuration information. #1 shows the list of features defined in Kconfig files. Each of the features is detailed after enumerations. In this case we see the first feature `ENABLE_FEATURE_2_4_MODULES` (#2). #3 shows documentation extracted from the Kconfig documentation and #4 lists the uses within the file by line number.

Figure 2 on the other hand, demonstrates a challenge seen during system evolution. The top portion (#1) shows the diff between two versions of a system taken from GitHub. Lines 236-7, 248, 242-243, and 246 are removed, while there are also a number of additions (216-217, 242-243, 246, 248-249 and 252). Looking more closely we see some of the code has been moved (shown by annotation arrows) and some is brand new (e.g. 242-243). It is not easy to determine what has happened in this diff. All of this churn is simply the result of a single addition of a feature, `MKFS_REISER`. An addition on line 242 can be seen, but the complete behavior is difficult to ascertain. Yet it is important for a tester or debugger to understand what has happened. The bottom portion (#2) shows a diff from ConfiGen's analysis showing that a feature is added.

2 MOTIVATION AND RELATED WORK

We begin with a motivating example to demonstrate the need for ConfiGen. Figure 1(a) shows automated documentation from a tool called Doxygen [14]. It extracts method level comments from the

2.1 Related Work

There has been a large body of research on misconfigurations [9, 11, 17, 44–46], performance issues related to configurability[2, 29, 41], modeling of configurations [5, 24, 36] and software product lines

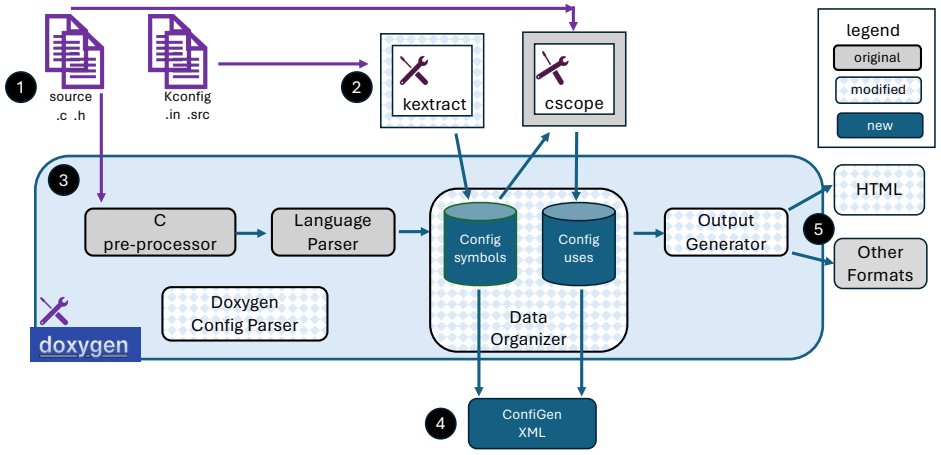


Figure 3: Overview of ConfiGen. The blue portion is the Doxygen workflow. Gray boxes are unchanged, light blue dashed boxes are modified by us from their original source, and the solid blue-green boxes indicate new modules we added.

[31, 35]. In addition, there is research on reverse engineering configuration variables and constraints [28, 32, 34]. Others have performed empirical analyses of configuration spaces [26, 27] and their evolution [6, 25]. The reverse engineering research is closest to our work however it does not provide documentation. Some researchers have defined variational languages (e.g. choice calculus [10, 15]), however it expects the programmer to encode their configuration options using a low level programming language construct. Tools such as Javadoc, Doxygen and DocStrings [13, 14, 20] provide an automated process to create a systematic set of documentation with traceability. We leverage Doxygen in this work, however it does not have the idea of configurability built in.

3 CONFIGEN

Figure 3 shows an overview of ConfiGen. The blue portion in the middle of the figure is the main part of the workflow of the Doxygen documentation tool [14]. In this figure, the gray boxes are unchanged parts, the light blue dashed boxes are modified by us from their original source, and the solid blue-green boxes indicate new modules we added. ConfiGen starts by taking in the source code of a C application (e.g. the .h and .c) files (#1). This is passed to the Doxygen C pre-processor and language parser (#3), existing (and unchanged modules) that create the standard documentation.

We use `kextract` (part of the `kmax` tool suite [23]) to parse the Kconfig files (.in and .src). It reads and parses configuration files that use Kconfig as their configuration management tool. Kconfig symbols are defined in the Kconfig files. Each symbol contains attributes such as its type definition, input prompt, default value, dependencies, reverse dependencies, weak reverse dependencies, numerical ranges, and help text. Kextract supports different argument options for parsing. One limitation is that if there is a dependency for the *range* attribute, it does not output that dependency. It also does not provide information about where a specific Kconfig symbol is defined, i.e., in which Kconfig file and at what line. We extended `kextract` and created a new option to include all the attribute values of a Kconfig symbol. We added the precise location information of a Kconfig symbol - (which Kconfig file and line the symbol is defined on). Last, we changed it to output the dependencies for the *range* attribute.

ConfiGen keeps the output of *kextract* in a Config symbols data dictionary. This builds a new dataset inside of the Doxygen data organizer (`Config symbols`). We then read this and use `Cscope` [12] to find uses of these configurations in the source (.c and .h) files and store these in a second data set (`Config uses`).

Doxygen also has a module it calls the `Config Parser` used for internal configurations of the Doxygen tool (not for features of the applications being documented). We modified this. From the two new data sets we create an XML that contains all of the configuration and uses in our system (#4). In Doxygen, the `Data Organizer` module builds dictionaries of the extracted classes, files, namespaces, variables, functions, packages, pages, and groups from the source code. We added the `Config symbols` and `Config uses` data dictionaries. We use this data to create XML files, used for further analysis of the configuration space, i.e. during system evolution. The output generator module uses the information generated by the data organizer module. We extended the HTML generator to present the configuration information in the HTML documentation files, including Kconfig symbols' attributes, the defined locations, and traceability of the symbols used in the source code.

4 CASE STUDY

We performed a case study to understand the potential of ConfiGen. We evaluate its ability to provide relevant information and compare ConfiGen against the state of the art tools.[1]

4.1 Subjects Studied

We choose three subjects that use Kconfig as their configuration language and have a long development history. Berger et al. [6] presented 11 projects that use the Kconfig language. We selected three open source projects, that are variants of Unix utilities: BusyBox, uClibc-ng, and Toybox. **BusyBox** [7, 8] is a C library that combines

[1] Supplementary data website: https://github.com/myracohen/ConfiGen-NIER24

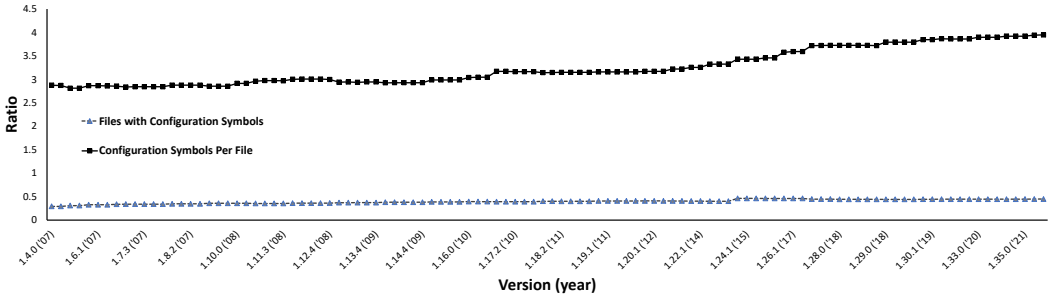


Figure 4: Ratio of Kconfig symbols in ConfiGen-generated documentation of BusyBox. The top line shows the ratio of feature changes per file and the bottom line shows the number of files that have feature changes.

many essential Unix utilities into a small executable, used often in embedded systems. **uClibc-ng** [39, 40] is a C library for embedded Linux systems. **Toybox** [37, 38] is a command-line tool that combines many Linux command-line utilities into a small executable.

We implemented a prototype of ConfiGen on top of Doxygen in C++. We ran both Doxygen and ConfiGen on the full history for the three subjects. For BusyBox, this includes v1.4.0 (2007) - v1.36.1 (2023), for uClibc-ng: v1.0.0 (2015) - v1.0.45 (2023), and for Toybox: v0.0.2 (2007) - v0.8.10 (2023). We failed to analyze v0.8.6 of Toybox because of a recursive dependency.

4.2 Results

> Our study suggests the potential for ConfiGen. (1) features are a significant part of the code; (2) they are distributed between files; and (3) the overhead for ConfiGen is small.

Table 1 shows data from our analysis for the number of versions analyzed, the average (and standard deviation) of the number of files in each version, the average (and standard deviation) of those files which have configuration options in the version, and the average number of configuration symbols (or features) in those files. We can see that all of these repositories have significant churn over the years. The percent of files with configuration-related code, ranges from 12.9% for uClibc-ng, to 39.9% for BusyBox. The number of times individual features are used within files, is between 1.4 and 3.3 times per file, suggesting the changes to configurations are distributed and can impact maintenance negatively.

Table 1: Number of versions for each subject, average number of files in version, average number of files with configuration symbols, and average number of configuration symbols.

	No.	Files		with Configs		Config Symbs	
Subject	Vers.	avg	std	avg	std	avg	std
BusyBox	108	715.73	60.21	285.06	47.02	938.33	241.79
uClibc-ng	46	3762.07	328.91	483.70	29.41	812.02	37.80
Toybox	47	192.83	87.61	48.17	27.24	66.64	35.31

Figure 4 shows this graphically over time, normalized as ratios. The top line is the number of features per file (for those files with configuration symbols) and the bottom line is the ratio of the number of files with configuration symbols (or features).

Table 2: Timing data for analysis of a single version across all versions (seconds)

	Doxygen		ConfiGen		
Subj.	Avg	Std	Avg	Std	OvHead
BusyBox	13.57	1.73	18.37	2.88	1.35
uClibc-ng	24.96	3.00	29.00	6.74	1.16
Toybox	2.85	1.30	4.02	1.77	1.42

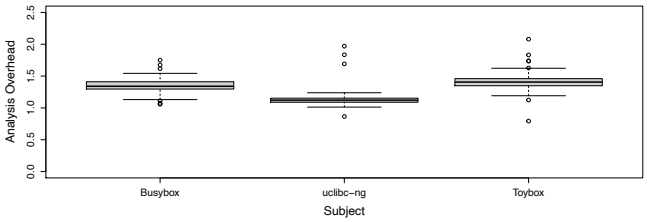


Figure 5: Overhead of ConfiGen over Doxygen

We now turn to the performance/overhead of using ConfiGen. Table 2 shows the average and standard deviation on seconds to generate documentation both using Doxygen and ConfiGen. As we can see the overhead (last column) ranges from 1.16 to 1.42. Last, Figure 5 shows a boxplot of this data. Given that the average time to create documentation for ConfiGen is less than half a minute this seems to be a practically useful tool.

5 CONCLUSIONS AND FUTURE WORK

We have presented ConfiGen, an approach to build configuration documentation into an automated documentation system. We believe this has potential to make configuration documentation first class and can aide with common maintenance tasks such as testing, debugging and refactoring. As future work we will build a more robust version of ConfiGen and evaluate this to understand its ability to help with these use cases. We will also add in runtime configurability (not yet supported) which does not rely on Kconfig.

ACKNOWLEDGMENTS

This work was funded in part by the National Science Foundation, CNS-2234908, CNS-2234909, CCF-1941816, and CCF-1909688.

REFERENCES

[1] Mathieu Acher, Philippe Collet, Philippe Lahire, and Robert B. France. 2013. FAMILIAR: A domain-specific language for large scale management of feature models. *Sci. Comput. Program.* 78, 6 (jun 2013), 657–681. https://doi.org/10.1016/j.scico.2012.12.004

[2] Juliana Alves Pereira, Mathieu Acher, Hugo Martin, and Jean-Marc Jézéquel. 2020. Sampling Effect on Performance Prediction of Configurable Systems: A Case Study. In *Proceedings of the ACM/SPEC International Conference on Performance Engineering (ICPE '20)*. 277–288. https://doi.org/10.1145/3358960.3379137

[3] Maurice H. ter Beek, Klaus Schmid, and Holger Eichelberger. 2019. Textual Variability Modeling Languages: An Overview and Considerations. In *Proceedings of the 23rd International Systems and Software Product Line Conference - Volume B* (Paris, France) *(SPLC '19)*. 151–157. https://doi.org/10.1145/3307630.3342398

[4] Farnaz Behrang, Myra B. Cohen, and Alessandro Orso. 2015. Users beware: preference inconsistencies ahead. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering* (Bergamo, Italy) *(ESEC/FSE 2015)*. 295–306. https://doi.org/10.1145/2786805.2786869

[5] David Benavides, Sergio Segura, and Antonio Ruiz-Cortés. 2010. Automated analysis of feature models 20 years later: a literature review. *Information Systems* 35, 6 (2010), 615–636.

[6] Thorsten Berger, Steven She, Rafael Lotufo, Andrzej Wasowski, and Krzysztof Czarnecki. 2013. A Study of Variability Models and Languages in the Systems Software Domain. *IEEE Transactions on Software Engineering* 39, 12 (2013), 1611–1640. https://doi.org/10.1109/TSE.2013.34

[7] BusyBox Git Repository [n. d.]. *busybox - BusyBox: The Swiss Army Knife of Embedded Linux*. Retrieved 2024 from https://git.busybox.net/busybox

[8] BusyBox Library [n. d.]. *BusyBox*. Retrieved Jun 10, 2024 from https://busybox.net/about.html

[9] Qingrong Chen, Teng Wang, Owolabi Legunsen, Shanshan Li, and Tianyin Xu. 2020. Understanding and Discovering Software Configuration Dependencies in Cloud and Datacenter Systems. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Virtual Event, USA) *(ESEC/FSE 2020)*. 362–374. https://doi.org/10.1145/3368089.3409727

[10] Sheng Chen, Martin Erwig, and Eric Walkingshaw. 2012. An error-tolerant type system for variational lambda calculus. *SIGPLAN Not.* 47, 9 (sep 2012), 29–40. https://doi.org/10.1145/2398856.2364535

[11] Runxiang Cheng, Lingming Zhang, Darko Marinov, and Tianyin Xu. 2021. Test-Case Prioritization for Configuration Testing. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual, Denmark) *(ISSTA 2021)*. 452–465. https://doi.org/10.1145/3460319.3464810

[12] Cscope [n. d.]. *Cscope Home Page*. Retrieved Jun 10, 2024 from https://cscope.sourceforge.net

[13] DocString [n. d.]. *PEP 257 – Docstring Conventions | peps.python.org*. Retrieved Jun 10, 2024 from https://peps.python.org/pep-0257/#what-is-a-docstring

[14] Doxygen [n. d.]. *Doxygen homepage*. Retrieved Jun 10, 2024 from https://www.doxygen.nl

[15] Martin Erwig and Eric Walkingshaw. 2011. The Choice Calculus: A Representation for Software Variation. *ACM Trans. Softw. Eng. Methodol.* 21, 1, Article 6 (dec 2011), 27 pages. https://doi.org/10.1145/2063239.2063245

[16] Brady J. Garvin and Myra B. Cohen. 2011. Feature Interaction Faults Revisited: An Exploratory Study. In *2011 IEEE 22nd International Symposium on Software Reliability Engineering*. 90–99. https://doi.org/10.1109/ISSRE.2011.25

[17] Paul Gazzillo. 2020. Inferring and Securing Software Configurations Using Automated Reasoning. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Virtual Event, USA) *(ESEC/FSE 2020)*. 1517–1520. https://doi.org/10.1145/3368089.3417041

[18] Paul Gazzillo and Myra B. Cohen. 2022. Bringing Together Configuration Research: Towards a Common Ground. In *Proceedings of the 2022 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! 2022)*. 259–269. https://doi.org/10.1145/3563835.3568737

[19] Ayelet Israeli and Dror G. Feitelson. 2010. The Linux kernel as a case study in software evolution. *Journal of Systems and Software* 83, 3 (2010), 485–501. https://doi.org/10.1016/j.jss.2009.09.042

[20] Javadoc [n. d.]. *javadoc*. Retrieved Jun 10, 2024 from https://docs.oracle.com/javase/8/docs/technotes/tools/windows/javadoc.html

[21] Christian Kastner, Thomas Thum, Gunter Saake, Janet Feigenspan, Thomas Leich, Fabian Wielgorz, and Sven Apel. 2009. FeatureIDE: A tool framework for feature-oriented software development. In *2009 IEEE 31st International Conference on Software Engineering*. 611–614. https://doi.org/10.1109/ICSE.2009.5070568

[22] Kconfig Language [n. d.]. *Kconfig Language — The Linux Kernel documentation*. Retrieved Jun 10, 2024 from https://www.kernel.org/doc/html/next/kbuild/kconfig-language.html

[23] Kmax Toolset [n. d.]. *paulgazz/kmax: A collection of analysis tools for Kconfig and Kbuild constraints*. Retrieved Jun 10, 2024 from https://github.com/paulgazz/kmax

[24] Elias Kuiter, Sebastian Krieter, Jacob Krüger, Thomas Leich, and Gunter Saake. 2019. Foundations of Collaborative, Real-Time Feature Modeling. In *Proceedings of the 23rd International Systems and Software Product Line Conference - Volume A (SPLC)*. ACM, 257–264.

[25] Rafael Lotufo, Steven She, Thorsten Berger, Krzysztof Czarnecki, and Andrzej Wąsowski. 2010. Evolution of the Linux kernel variability model. In *Software Product Lines: Going Beyond: 14th International Conference, SPLC 2010, Jeju Island, South Korea, September 13-17, 2010. Proceedings 14*. Springer, 136–150.

[26] Flávio Medeiros, Christian Kästner, Márcio Ribeiro, Rohit Gheyi, and Sven Apel. 2016. A Comparison of 10 Sampling Algorithms for Configurable Systems. In *Proceedings of the 38th International Conference on Software Engineering (ICSE)*. ACM Press, New York, NY, 643–654. https://doi.org/10.1145/2884781.2884793

[27] Jens Meinicke, Chu-Pan Wong, Christian Kästner, Thomas Thüm, and Gunter Saake. 2016. On Essential Configuration Complexity: Measuring Interactions In Highly-Configurable Systems. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*. ACM Press, New York, NY, 483–494. https://doi.org/10.1145/2970276.2970322

[28] Sarah Nadi, Thorsten Berger, Christian Kästner, and Krzysztof Czarnecki. 2014. Mining configuration constraints: static analyses and empirical results. In *Proceedings of the 36th International Conference on Software Engineering (ICSE 2014)*. 140–151. https://doi.org/10.1145/2568225.2568283

[29] Vivek Nair, Tim Menzies, Norbert Siegmund, and Sven Apel. 2017. Using Bad Learners to Find Good Configurations. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2017)*. 257–267. https://doi.org/10.1145/3106237.3106238

[30] Jeho Oh, Necip Fazıl Yıldıran, Julian Braha, and Paul Gazzillo. 2021. Finding Broken Linux Configuration Specifications by Statically Analyzing the Kconfig Language. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Athens, Greece) *(ESEC/FSE 2021)*. 893–905. https://doi.org/10.1145/3468264.3468578

[31] Philipp Dominik Schubert, Paul Gazzillo, Zach Patterson, Julian Braha, Fabian Schiebel, Ben Hermann, Shiyi Wei, and Eric Bodden. 2022. Static Data-Flow Analysis for Software Product Lines in C. *Automated Software Engineering* (2022), 35 pages. https://doi.org/10.1007/s10515-022-00333-1 To appear..

[32] Steven She, Rafael Lotufo, Thorsten Berger, Andrzej Wąsowski, and Krzysztof Czarnecki. 2011. Reverse Engineering Feature Models. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE)*. ACM, 461–470.

[33] Chico Sundermann, Kevin Feichtinger, Dominik Engelhardt, Rick Rabiser, and Thomas Thüm. 2021. Yet another textual variability language? a community effort towards a unified language. In *Proceedings of the 25th ACM International Systems and Software Product Line Conference - Volume A* (Leicester, United Kingdom) *(SPLC '21)*. 136–147. https://doi.org/10.1145/3461001.3471145

[34] Thammasak Thianniwet and Myra. B. Cohen. 2015. SPLRevO: Optimizing complex feature models in search based reverse engineering of software product lines. In *Proceedings of the 1st North American Search Based Software Engineering Symposium (NasBASE)*. 1–16.

[35] Thomas Thüm, Sven Apel, Christian Kästner, Ina Schaefer, and Gunter Saake. 2014. A classification and survey of analysis strategies for software product lines. *ACM Computing Surveys (CSUR)* 47, 1, Article 6 (2014), 45 pages.

[36] Thomas Thüm, Christian Kästner, Fabian Benduhn, Jens Meinicke, Gunter Saake, and Thomas Leich. 2014. FeatureIDE: An extensible framework for feature-oriented software development. *Science of Computer Programming* 79 (2014), 70 – 85. Experimental Software and Toolkits (EST 4): A special issue of the Workshop on Academic Software Development Tools and Techniques (WASDeTT-3 2010).

[37] Toybox Git Repository [n. d.]. *landley/toybox: toybox*. Retrieved Jun 10, 2024 from https://github.com/landley/toybox

[38] Toybox Library [n. d.]. *What is toybox?* Retrieved Jun 10, 2024 from https://landley.net/toybox

[39] uClibc-ng Git Repository [n. d.]. *wbx-github/uclibc-ng: Embedded C Library (mirror)*. Retrieved Jun 10, 2024 from https://github.com/wbx-github/uclibc-ng

[40] uClibc-ng Library [n. d.]. *Welcome to uClibc-ng! - Embedded C library*. Retrieved Jun 10, 2024 from https://uclibc-ng.org

[41] Miguel Velez, Pooyan Jamshidi, Norbert Siegmund, Sven Apel, and Christian Kästner. 2021. White-Box Analysis over Machine Learning: Modeling Performance of Configurable Systems. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. 1072–1084. https://doi.org/10.1109/ICSE43902.2021.00100

[42] Tianyin Xu, Long Jin, Xuepeng Fan, Yuanyuan Zhou, Shankar Pasupathy, and Rukma Talwadker. 2015. Hey, you have given me too many knobs!: understanding and dealing with over-designed configuration in system software. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering* (Bergamo, Italy) *(ESEC/FSE 2015)*. 307–319. https://doi.org/10.1145/2786805.2786852

[43] Tianyin Xu, Jiaqi Zhang, Peng Huang, Jing Zheng, Tianwei Sheng, Ding Yuan, Yuanyuan Zhou, and Shankar Pasupathy. 2013. Do not blame users for misconfigurations. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (Farminton, Pennsylvania) *(SOSP '13)*. 244–259. https://doi.org/10.1145/2517349.2522727

[44] Tianyin Xu and Yuanyuan Zhou. 2015. Systems Approaches to Tackling Configuration Errors: A Survey. *ACM Comput. Surv.* 47, 4, Article 70 (July 2015), 41 pages. https://doi.org/10.1145/2791577

[45] Zuoning Yin, Xiao Ma, Jing Zheng, Yuanyuan Zhou, Lakshmi N. Bairavasundaram, and Shankar Pasupathy. 2011. An empirical study on configuration errors in commercial and open source systems. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (Cascais, Portugal) *(SOSP '11)*. 159–172. https://doi.org/10.1145/2043556.2043572

[46] Jialu Zhang, Ruzica Piskac, Ennan Zhai, and Tianyin Xu. 2021. Static Detection of Silent Misconfigurations with Deep Interaction Analysis. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 140, 30 pages. https://doi.org/10.1145/3485517

[47] Sai Zhang and Michael D. Ernst. 2014. Which configuration option should I change?. In *Proceedings of the 36th International Conference on Software Engineering* (Hyderabad, India) *(ICSE 2014)*. 152–163. https://doi.org/10.1145/2568225.2568251