A Demo of ConfigFix: Semantic Abstraction of Kconfig, SAT-based Configuration, and DIMACS Export

Jude Gyimah
Ruhr-University Bochum
Germany, Bochum

Jan Sollmann
Ruhr-University Bochum
Germany, Bochum

Ole Schuerks
Ruhr-University Bochum
Germany, Bochum

Patrick Franz
University of Gothenburg
Sweden, Gothenburg

Thorsten Berger
Ruhr-University Bochum and
Chalmers | University of Gothenburg
Germany, Bochum

Abstract

The Linux kernel and its huge configuration space (>15,000 features) has been a frequent study object. While the research community has developed intelligent software configuration tools, often motivated by the Linux kernel and its configuration language *Kconfig*, the kernel's own configurator xconfig lacks behind. Configuration conflicts need to be resolved manually, which often causes substantial overhead. Unfortunately, *Kconfig* is a complex and intricate language, and while transformations into propositional logic exist, they typically have shortcomings and are difficult to integrate into xconfig. We contribute research results back to the Linux community and present a demo of ConfigFix. It is a plain-C-based extension of xconfig, providing the currently most accurate abstraction of the *Kconfig* semantics into propositional logic. It provides configuration conflict resolution. Integrated into the xconfig UI, it offers configuration fixes to users trying to enable or disable kernel features restricted by dependencies. In addition, researchers benefit from the DIMACS export. Our demo presents the main capabilities of xconfig as well as its evaluation showing the accuracy of it.

Keywords

configuration, Linux Kernel, configuration conflicts, SAT solving

ACM Reference Format:
Jude Gyimah, Jan Sollmann, Ole Schuerks, Patrick Franz, and Thorsten Berger. 2024. A Demo of ConfigFix: Semantic Abstraction of Kconfig, SAT-based Configuration, and DIMACS Export. In *Proceedings of 19th International Working Conference on Variability Modelling of Software-Intensive Systems (VaMoS'25).* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

The Linux kernel's applicability in many different computing environments—ranging from small Android devices to large scale supercomputer clusters—has contributed to making it one of the world's largest software development projects [8]. Designed as a *highly configurable system* [50] it boasts 28 million lines of code [26] and over 15,000 configuration options (a.k.a. *features* [5, 7, 40]). To this end, it has a configurable build system [6], preprocessor-based variation points [7], a model of its features (*Kconfig model*) and constraints [7, 34], and an interactive configurator tool (*xconfig*) [50].

A plethora of academic studies and techniques on the kernel's variability and its configuration space exist, ranging from variability anomaly detection [2, 31, 35, 52], techniques to identify, extract, and analyze configuration constraints [29, 34, 49, 50, 55], as well as automated support for reviewing and testing patches [27, 28]. In addition to that, many studies have also been conducted on the kernel's evolution [4, 19, 41, 43] and maintenance [1, 18, 20, 56], analyzing its feature model [7, 47], the modeling language (*Kconfig*), the evolution of the model [29], the co-evolution and consistency of its variation points [22, 36, 37, 41, 42, 54], and the representation of feature constraints in its codebase [34]. Many of the results have inspired software configurator tools [13, 15]. Various other tools [51, 58] and techniques [32, 33, 44], including the synthesis of feature models from code or feature constraints [23, 30, 48], have also drawn inspiration from the Linux kernel and its configurator, by borrowing from or directly extending its capabilities.

The kernel's main configurator xconfig, however, lacks behind the state of the art. Kernel users have faced challenges when manually creating their desired configurations, given the kernel's vast configuration space and intricate feature constraints. Furthermore, there is limited support for choice propagation and a lack of intelligent configurator support for resolving configuration conflicts. These challenges are common, because enabling a feature often requires transitively changing many others. Consequently, achieving a desired configuration can in turn be laborious and error-prone. A survey [17] revealed that kernel users commonly struggle with conflict resolution, with 20 % of the respondents needing at least "a few dozen minutes" to do so. Despite this, insofar as we know, no configuration technique stemming from academia, has been specifically developed for the Linux kernel such that it can be officially integrated into the Linux kernel mainline.

In 2015, with the Kconfig-sat initiative [24] kernel developers recognized the need for such a solution. They got in touch with researchers working on kernel configuration studies, to integrate such techniques back into the kernel configurator. Indeed, implementing a sound translation of the kernel's variability model to propositional logic, given the expressiveness and intricate nature of

VaMoS'25, 4-6 February 2025, Rennes, France
© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

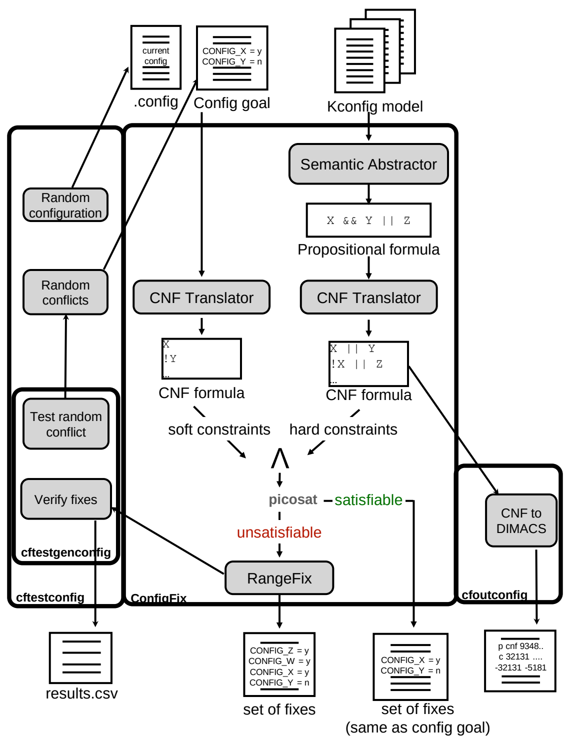

Fig. 1: ConfigFix components

the *Kconfig* language semantics (explained shortly), has long been an open problem in the community. Multiple translation attempts have been proposed to remedy this [10, 21, 46, 53], but none of them have been without limitations [9].

We present a demo of ConfigFix [11, 12], which offers intelligent configuration support integrated into the Linux kernel configurator *xconfig*. It is completely implemented in C, extends *xconfig*'ss graphical user interface, and comes with a testing framework. Since researchers can draw value from the DIMACS export, we also discuss the Kconfig's semantics and the implementation details of our semantic abstraction. In contrast to our previous work [11], this paper is a demo, showcasing enhancements in the user interface, test infrastructure, and code quality, and it presents the DIMACS export. The current version also integrates feedback from the kernel community. Our efforts have led to a stable version of ConfigFix and its test infrastructure [12], that is currently in the process of being integrated in the kernel's source tree (patches submitted and discussed).

2 ConfigFix Design

Figure 1 shows the main components of ConfigFix: translation of the *Kconfig* model into a propositional formula, translation into conjunctive normal form (CNF), conflict resolution algorithm, test infrastructure, and DIMACS export. When invoked, ConfigFix takes the current configuration and the user's configuration goal as input. It then creates a single formula that is a conjunction of all constraints and then queries the SAT solver for satisfiability. In case of a configuration conflict (i.e., the user's goal is not applicable), it triggers our C-based, RangeFix-inspired implementation (discussed shortly) to calculate fixes [11, 59].

Linux Kernel Configuration. The kernel has three different configurators: *make xconfig*, *make menuconfig*, and *make config*. The *xconfig* provides a graphical user interface, while the others are tailored towards shell users. Via them correct configurations of kernel features (which come with many different characteristics defined in the *Kconfig* model) that adhere to constraints (e.g., types and dependencies) can be created and modified. The underlying DSL [3] *Kconfig* provides feature-modeling concepts [7, 47]. It declares features (called symbols there) of different types (i.e., bool, tristate, string, hex, and int) [45, 46]. Tristate features are often used to control the binding modes of features via three states: y (yes, compile feature into kernel), n (no, do not compile feature), or m (compile feature as loadable kernel module), where constraint semantics follow Kleene's rules for three-state logic [25]. A persistent challenge has been obtaining a sound logical representation (mainly due to its complexity) of the main semantics of *Kconfig*, as a prerequisite to develop analysis and configuration techniques [11]. After we were the first to formalize its semantics [45, 46] (reverse-engineered from *xconfig*'s behavior), many others followed up [9, 24, 38, 45, 61] with their own translations. These works demonstrated that the semantics of *Kconfig* can indeed be abstracted into propositional logic, to be used by SAT solvers.

Translation. Given *Kconfig*'s expressiveness beyond propositional logic, an abstraction is required. Specifically, ConfigFix translates each feature (called symbol in *Kconfig*) into a set of variables that represent the possibility of the feature assuming a specific value. For example, a Boolean feature will be translated to a single variable A which is either true or false. Other features may only assume the values "yes," "module," or "no." For such a feature A, the variables A and A_M would be created, where A is true if and only if A is "yes." and A_M is true if and only if A is "module." In case of an integer feature A, with default value of 5 and a constraint $2 \leq A \leq 8$, it would be translated into six variables in the propositional formula: A=0, A=1, A=2, A=5, A=8 and A=n. A=0, A=1 and A=n are created for every feature by default with A=n handling the case that A is hidden and has no value. A=2, A=5 and A=8 are "known values," which could be values that are either default for the feature or part of a range constraint. The propositional formula is then constructed based on these variables. It is noteworthy that this aspect of the ConfigFix translation implementation is the most complicated, given that it is mainly tristate and the sheer scale of everything (features, values, types and especially the CNF conversion), was challenging. The formula is subsequently translated into CNF using Tseitin transformation [57].

Fix Generation. The ConfigFix UI is responsible for receiving configuration conflicts specified as the user-defined configuration goal that is not applicable in the current configurator, and then displaying calculated fixes. The constraint translation component of ConfigFix first translates the constraints in the loaded *Kconfig* model into a CNF formula. To calculate and generate fixes from conflicts, a suitable conflict-resolution is required. Various conflict-resolution algorithms exist [16], but many were not applicable to the scope of ConfigFix. They either produce just a single fix, suggest a long list of fixes or only offer limited support for non-Boolean constraints [39]. However, our conflict-resolution algorithm of choice, a rangefix-inspired conflict resolver, produces fixes that offer a range of values for features, supports non-Boolean features and constraints, adheres to correctness, contains a maximum range in the event of overlapping ranges and requires minimal feature set changes [11]. In our implementation, it takes a CNF model where features are represented by variables and generates minimal sets of features (refered to as diagnoses) that must be changed. Next, it calculates new values for each variable in a diagnosis. All unchanged variables are then replaced by their current values and any violated constraints are minimized via heuristic rules defined and split into minimal clauses to generate the fixes. In summary, with those *RangeFix* capabilities, when there are infinitely many possible solutions available, the user is only presented with the minimal set of these [60].

3 ConfigFix Demo

We now demonstrate ConfigFix's conflict resolution workflows and DIMACS export functionality. We show the *xconfig* UI, the specification of the configuration goal (the assignment of values for a set of features to be configured), the calculation of adequate fixes, and their application.

Workflows. Figure 2 presents the complete workflow of ConfigFix. ① Upon launch, *Xconfig* provides a view that presents a dependency-based hierarchy of nested menus containing features and allows feature value changes when the requirements for the target values are met. The *Kconfig* language allows specifying conditions for when a feature should be visible in the view. To change the values of features, even when they are invisible, the option "Show Prompt Options" can be used. Then all features that can be visible with the default visibility option under any configuration are always visible. Users can add any tristate feature to the set of features that should be changed by selecting it in this view and clicking the "Add symbol" button (as shown in ②). Conflicts with numeric values and string values are not supported by ConfigFix. The set of features that need be changed is also called a conflict. The features that are currently added to the conflict are displayed in the menu on the bottom left. The desired value of a feature in the conflict can be set by selecting it in the menu on the bottom left and using one of the buttons "Y", "M", or "N" to set it to "Yes", "Module", or "No," respectively. Alternatively, users can cycle through the three values by clicking on the corresponding cell in the column "Wanted Value." The difference between "Yes" and "Module" is that with the former, a feature will be statically linked into the kernel, whereas with the latter it will be built as a dynamically loadable kernel module. If a feature cannot be built as a kernel module, the button "M" is grayed out. A feature (symbol) can be removed from the conflict by selecting it in the menu on the bottom left and clicking the button "Remove Symbol."

Once the desired feature values have been set, fixes can be calculated by clicking "Calculate Fixes." The fixes are then displayed in the table on the bottom right (as shown in ③). Users can switch between different fixes with a dropdown menu. The fixes associate each feature in the conflict with a new value, and ideally, there is an order of the changes the fix presents in which they can be applied such that after applying all changes, the desired changes would have been achieved. The changes can be made manually, or by clicking on "Apply Selected Solution", the values can be applied automatically, in which case the user is informed via a dialog when the fix has been applied successfully ④. ConfigFix supports the

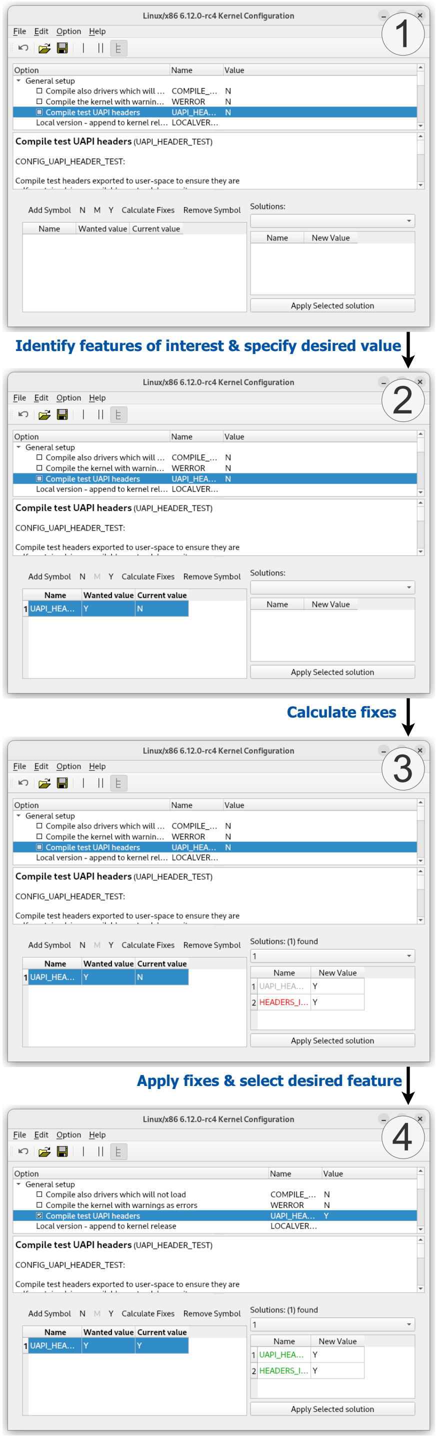
1
Linux/x86 6.12.0-rc4 Kernel Configuration
Identify features of interest & specify desired value
2
Calculate fixes
3
Apply fixes & select desired feature
4
Add Symbol
Calculate Fixes
Remove Symbol
Solutions:
Apply Selected solution

Fig. 2: User workflow using ConfigFix inside xconfig

user in applying the changes manually by coloring the names of the features in the table showing the fixes based on when the value can and needs to be set manually. A feature is red if its value can be set to the value provided by the fix and that value differs in the current configuration (as shown in ③). It is green if its value in the current configuration already matches with the value the fix proposes (as shown in ④). It is gray if the fix's value differs from the configuration's value for the feature, but further dependencies need to be fulfilled in order to set the feature to the value of the fix. Figure 2 shows the sequence of the workflow described from the identification of features of interest to be configured, right down to the application of calculated fixes.

DIMACS Export. ConfigFix allows *Kconfig* model exports in DIMACS notation, using "make cfoutconfig." The DIMACS output file offers users the *Kconfig* model in CNF form (see Listing 1). This export feature is included to support further research on the kernel's feature model and variability mechanisms. For instance, such DIMACS exports could be potentially useful in logical inference scenarios where system based conclusions can be drawn from known configuration constraints, using algorithms that verify the satisfiability (SAT) of the underlying propositional logic statements.

Listing 1: DIMACS export snippet for UAPI_HEADER_TEST

```
p cnf 934872 2764895
c 32131 UAPI_HEADER_TEST
-32131 -51181 0
```

4 Test Infrastructure

We evaluated ConfigFix based on the number of conflicts it resolves and the quality of fixes. We created a test infrastructure that generates random configuration samples, introduces conflicts, executes ConfigFix, and then validates the fixes. This implementation extends the *qconf.cc* configurator and is invoked by the command "make cftestconfig". The *cftestconfig* module is split into two submodules, the first is *ConflictFrameworkSetup* responsible for initializing system parameters and the other (*ConflictFramework*), which handles the generation of configurations and conflicts.

Cftestconfig can be used to generate configurations and conflicts in three different modes. ***mode="single"*** gives the user the option to generate several conflicts from an existing configuration. When these conflicts are resolved, the data is saved in a results file. In this mode, "make cftestgenconfig" can be called as many times as required. ***mode="multi"*** provides the option to generate multiple random configurations while simultaneously generating for every configuration, a random set of conflicts for a given architecture. The conflicts are solved and the data is subsequently saved in the results file. In its order of execution, "make randconfig" is called first, followed by "make cftestgenconfig". ***mode="multi_arch"*** gives users the option to generate multiple random configurations and then generate for each, a random set of conflicts for multiple architectures.

Specifically, the test infrastructure works as follows. First, *it samples the configuration space.* Samples should provide enough coverage of the features contained therein [14]. This requires all important variation points (such as hardware architecture) in the configuration space to be considered [14]. Our test infrastructure uses `randconfig` (an existing tool in the build system) to generate random configurations. Second, *it introduces conflicts.* For each of the generated configurations, it generates random conflicts containing a specific number of changes requested (i.e., the conflict size, which is configurable).

Table 1: Evaluation results

metric	value
number of sampled configurations	27[1]
conflict sizes	1–10
total generated conflicts for evaluation	1,350[2] (100.0 %)
conflicts with >= 1 fix produced by ConfigFix	1,150 (85.2 %)
number of resolved conflicts	1,150 (85.2 %)
total number of fixes produced by ConfigFix	2,645 (100.0 %)
fixes that resolve the conflict	**2,643 (99.9 %)**
fully applicable and resolves conflict	1,116 (42.2 %)
not fully applicable, but resolves conflict	1,527 (57.7 %)
does not resolve conflict	2 (0.1 %)

[1] One for each architecture and probability.
[2] For each configuration sample, five conflicts of each size.

We then generate a base configuration using `randconfig` with a probability of 100% of an option being set with all conflicts chosen as subsets of the base configuration. Third, *it executes the fix generation* for each conflict. The generated configuration sample and conflict become inputs to the test algorithm, which will either produce a single or several configuration fixes, or provide feedback that the conflict cannot be resolved. Fourth, *it validates the fixes.* A fix may include some additional features subject to change, in order to set them to their new, desired state. Such features typically bear configuration conflict properties, which may be too hard to reconfigure manually. As such, for every fix returned by the algorithm, it is applied to the sample configuration and verified to ensure that user needs are satisfied, the configuration after the fix application is valid, as well as no unnecessary changes have been made to the product configuration.

All generated configurations, conflicts and fixes are stored in a directory supplied by the user. Conflicts are generated by repeatedly choosing a random boolean or tristate option with a prompt that has at least one value to which it cannot resolve due to unmet dependencies. The conflict then requires this option to be set to one of its blocked values (chosen at random).

For the actual evaluation, we generated configurations (i.e., in the multi_arch mode) for three different architectures (`x86_64`, `arm64`, and `openrisc`), and for nine different probabilities for a feature to be selected, ranging from 10 % to 90 %. In total, the infrastructure evaluated 27 different configurations. We set it to generate conflict sizes of 1–10 features. Table 1 shows the results. It generated 1350 conflicts. For each of the 27 configurations, and each of the 10 conflict sizes, 5 conflicts were generated. ConfigFix successfully resolved 1,150 conflicts (85.2%) and produced 2,645 fixes in total. Almost all fixes (99.9%) resolved the conflicts, with only 2 fixes (0.1%) failing to do so. Of the fixes, 1,116 (42.2%) were fully applicable and resolved the conflict, while 1,527 (57.7%) resolved the conflict despite not being fully applicable. The evaluation details are provided in the ConfigFix repository [12].

Note, a fix could still not resolve a conflict since our abstraction compresses the higher level *Kconfig* model into a propositional formula, which may not be able to capture all the constraints of the original model. Thus a fix can be fully inapplicable but still resolve the conflict, since the fix may change the value of a variable that is not directly involved in the conflict, but is a dependency of a variable that is involved in the conflict.

5 Conclusion

We presented a demo of ConfigFix. It helps to configure the Linux kernel by offering automated conflict resolution support. While details are in our preceding conference paper [11], we focus on demonstrating the capabilities, as well as showing improvements since then. We hope the C-based integration developed by discussion with Linux developers contributes research results back to the Linux kernel community, supporting users to configure their desired kernel.

References

[1] Iago Abal, Claus Brabrand, and Andrzej Wasowski. 2014. 42 Variability Bugs in the Linux Kernel: A Qualitative Analysis. In ASE.

[2] Mathieu Acher, Hugo Martin, Juliana Alves Pereira, Arnaud Blouin, Djamel Eddine Khelladi, and Jean-Marc Jézéquel. 2019. Learning From Thousands of Build Failures of Linux Kernel Configurations. Technical Report. Inria ; IRISA. https://inria.hal.science/hal-02147012

[3] Wąsowski Andrzej and Berger Thorsten. 2023. Domain-Specific Languages: Effective Modeling, Automation, and Reuse. Springer International Publishing.

[4] Giuliano Antoniol, Umberto Villano, Ettore Merlo, and Massimiliano Di Penta. 2002. Analyzing cloning evolution in the linux kernel. Information and Software Technology 44, 13 (2002), 755–765.

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

[6] Thorsten Berger, Steven She, Rafael Lotufo, Krzysztof Czarnecki, and Andrzej Wasowski. 2010. Feature-to-Code Mapping in Two Large Product Lines. In SPLC.

[7] 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 Trans. Softw. Eng. 39, 12 (Dec. 2013), 1611–1640.

[8] Swapnil Bhartiya. 2016. Linux is the largest software development project on the planet: Greg Kroah-Hartman. urlhttps://www.cio.com/article/3069529/linux-is-the-largest-software-development-project-on-the-planet-greg-kroah-hartman.html. Accessed: 2020-01-06.

[9] Sascha El-Sharkawy, Adam Krafczyk, and Klaus Schmid. 2015. Analysing the Kconfig semantics and its analysis tools. In GPCE.

[10] David Fernandez-Amoros, Ruben Heradio, Christoph Mayr-Dorn, and Alexander Egyed. 2019. A Kconfig Translation to Logic with One-Way Validation System. In SPLC.

[11] Patrick Franz, Thorsten Berger, Ibrahim Fayaz, Sarah Nadi, and Evgeny Groshev. 2021. ConfigFix: Interactive Configuration Conflict Resolution for the Linux Kernel. In IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP).

[12] Patrick Franz, Ibrahim Fayaz, Thorsten Berger, Sarah Nadi, Evgeny Groshev, Lukas Günther, Dorina Sfirnaciuc, Jude Gyimah, Jan Sollman, and Ole Schürks. 2020. ConfigFix. https://bitbucket.org/easelab/configfix.

[13] José A Galindo, David Benavides, Pablo Trinidad, Antonio-Manuel Gutiérrez-Fernández, and Antonio Ruiz-Cortés. 2019. Automated analysis of feature models: Quo vadis? Computing 101 (2019), 387–433.

[14] Evgeny Groshev. 2020. A testing technique for conflict-resolution facilities in software configurators. Master's thesis. University Of Gothenburg and Chalmers University Of Technology.

[15] Arnaud Hubaux, Dietmar Jannach, Conrad Drescher, Leonardo Murta, Tomi Männistö, Krzysztof Czarnecki, Patrick Heymans, Tien Nguyen, and Markus Zanker. 2012. Unifying software and product configuration: a research roadmap. In CONFWS.

[16] A. Hubaux, D. Jannach, C. Drescher, L. Murta, T. Männistö, K. Czarnecki, P. Heymans, T. Nguyen, and M. Zanker. 2012. Unifying Software and Product Configuration: A Research Roadmap. In ConfWS.

[17] Arnaud Hubaux, Yingfei Xiong, and Krzysztof Czarnecki. 2012. A User Survey of Configuration Challenges in Linux and eCos. In VaMoS.

[18] Ayelet Israeli and Dror G Feitelson. 2009. Characterizing software maintenance categories using the Linux kernel. Technical Report 2009–10. School of Computer Science and Engineering, The Hebrew University of Jerusalem.

[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.

[20] Yujuan Jiang, Bram Adams, and Daniel M German. 2013. Will my patch make it? and how fast? case study on the linux kernel. In MSR.

[21] Christian Kästner. 2014. KConfig Reader. https://github.com/ckaestne/kconfigreader.

[22] Christian Kästner, Paolo G. Giarrusso, Tillmann Rendel, Sebastian Erdweg, Klaus Ostermann, and Thorsten Berger. 2011. Variability-Aware Parsing in the Presence of Lexical Macros and Conditional Compilation. In OOPSLA.

[23] 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

[24] Kernelnewbies. 2017. Linux kconfig SAT integration. https://kernelnewbies.org/KernelProjects/kconfig-sat. Accessed: 2019-12-05.

[25] S. C. Kleene. 1938. On notation for ordinal numbers. Journal of Symbolic Logic 3, 4 (1938), 150–155. https://doi.org/10.2307/2267778

[26] Michael Larabel. 2020. The Linux Kernel Enters 2020 At 27.8 Million Lines In Git But With Less Developers For 2019. https://www.phoronix.com/news/Linux-Git-Stats-EOY2019. Accessed: 2020-01-06.

[27] Julia Lawall and Gilles Muller. 2017. JMake: Dependable Compilation for Kernel Janitors. In 2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN). 357–366. https://doi.org/10.1109/DSN.2017.62

[28] Julia Lawall and Gilles Muller. 2018. Coccinelle: 10 Years of Automated Evolution in the Linux Kernel. In 2018 USENIX Annual Technical Conference (USENIX ATC 18). https://www.usenix.org/conference/atc18/presentation/lawall

[29] Rafael Lotufo, Steven She, Thorsten Berger, Krzysztof Czarnecki, and Andrzej Wasowski. 2010. Evolution of the Linux Kernel Variability Model. In International Conference on Software Product Lines (SPLC). 136–150.

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

[31] Jean Melo, Elvis Flesborg, Claus Brabrand, and Andrzej Wąsowski. 2016. A quantitative analysis of variability warnings in linux. In Proceedings of the 10th International Workshop on Variability Modelling of Software-Intensive Systems. 3–8.

[32] Johann Mortara and Philippe Collet. 2021. Capturing the diversity of analyses on the Linux kernel variability. In Proceedings of the 25th ACM International Systems and Software Product Line Conference - Volume A (Leicester, United Kingdom) (SPLC '21). Association for Computing Machinery, New York, NY, USA, 160–171. https://doi.org/10.1145/3461001.3471151

[33] Daniel-Jesus Munoz, Jeho Oh, Mónica Pinto, Lidia Fuentes, and Don Batory. 2019. Uniform random sampling product configurations of feature models that have numerical features. In SPLC.

[34] Sarah Nadi, Thorsten Berger, Christian Kästner, and Krzysztof Czarnecki. 2015. Where do Configuration Constraints Stem From? An Extraction Approach and an Empirical Study. IEEE Trans. Softw. Eng. 41, 8 (2015), 820–841.

[35] Sarah Nadi, Christian Dietrich, Reinhard Tartler, Richard C. Holt, and Daniel Lohmann. 2013. Linux variability anomalies: What causes them and how do they get fixed?. In MSR.

[36] Sarah Nadi and Richard C. Holt. 2011. Make it or break it: Mining anomalies from Linux Kbuild. In WCRE. 315–324.

[37] Sarah Nadi and Richard C. Holt. 2012. Mining Kbuild to detect variability anomalies in Linux. In CSMR.

[38] Vegard Nossum. 2019. satconfig. https://github.com/vegard/linux-2.6/tree/v4.7+kconfig-sat. Accessed: 2019-12-05.

[39] Leonardo Passos, Marko Novakovic, Yingfei Xiong, Thorsten Berger, Krzysztof Czarnecki, and Andrzej Wąsowski. 2011. A Study of Non-Boolean Constraints in a Variability Model of an Embedded Operating System. In Feature-Oriented Software Development (FOSD).

[40] Leonardo Passos, Jesús Padilla, Thorsten Berger, Sven Apel, Krzysztof Czarnecki, and Marco Tulio Valente. 2015. Feature Scattering in the Large: A Longitudinal Study of Linux Kernel Device Drivers. In International Conference on Modularity (MODULARITY).

[41] Leonardo Passos, Rodrigo Queiroz, Mukelabai Mukelabai, Thorsten Berger, Sven Apel, Krzysztof Czarnecki, and Jesus Padilla. 2021. A Study of Feature Scattering in the Linux Kernel. IEEE Trans. Softw. Eng. 47 (2021), 146–164. Issue 1.

[42] Leonardo Passos, Leopoldo Teixeira, Nicolas Dintzner, Sven Apel, Andrzej Wasowski, Krzysztof Czarnecki, Paulo Borba, and Jianmei Guo. 2016. Coevolution of Variability Models and Related Software Artifacts. Empirical Softw. Engg. 21, 4 (Aug. 2016), 1744–1793.

[43] Tu Qiang and Godfrey Michael W. 2000. Evolution in open source software: A case study. In ICSM.

[44] Muhammad Ejaz Sandhu. 2021. Comparison of Fault Simulation Over Custom Kernel Module Using Various Techniques. Lahore Garrison University Research Journal of Computer Science and Information Technology 5, 3 (2021), 73–83. https://doi.org/10.54692/lgurjcsit.2021.0503220

[45] Steven She. 2013. LVAT. https://github.com/shshe/linux-variability-analysis-tools.

[46] Steven She and Thorsten Berger. 2010. Formal Semantics of the Kconfig Language. Technical Note. https://arxiv.org/abs/2209.04916.

[47] Steven She, Rafael Lotufo, Thorsten Berger, Andrzej Wasowski, and Krzysztof Czarnecki. 2010. The Variability Model of the Linux Kernel. In Fourth International Workshop on Variability Modelling of Software-intensive Systems (VAMOS 2010).

[48] Steven She, Rafael Lotufo, Thorsten Berger, Andrzej Wasowski, and Krzysztof Czarnecki. 2011. Reverse Engineering Feature Models. In ICSE.

[49] Julio Sincero, Horst Schirmeier, Wolfgang Schröder-Preikschat, and Olaf Spinczyk. 2007. Is The Linux Kernel a Software Product Line?. In SPLC-OSSPL.
[50] Julio Sincero and Wolfgang Schröder-Preikschat. 2008. The Linux Kernel Configurator as a Feature Modeling Tool. In ASPL.
[51] Eduard Staniloiu, Razvan Nitu, Cristian Becerescu, and Razvan Rughinis. 2021. Automatic Integration of D Code With the Linux Kernel. In 2021 20th RoEduNet Conference: Networking in Education and Research (RoEduNet). 1–6. https://doi.org/10.1109/RoEduNet54112.2021.9638307
[52] Stefan Strueder, Mukelabai Mukelabai, Daniel Strueber, and Thorsten Berger. 2020. Feature-Oriented Defect Prediction. In 24th ACM International Systems and Software Product Line Conference (SPLC).
[53] Reinhard Tartler, Christian Dietrich, Julio Sincero, Wolfgang Schröder-Preikschat, and Daniel Lohmann. 2014. Static Analysis of Variability in System Software: The 90,000 #ifdefs Issue. In USENIX ATC.
[54] Reinhard Tartler, Daniel Lohmann, Julio Sincero, and Wolfgang Schröder-Preikschat. 2011. Feature Consistency in Compile-Time-Configurable System Software: Facing the Linux 10,000 Feature Problem. In EuroSys.
[55] Reinhard Tartler, Julio Sincero, Wolfgang Schröder-Preikschat, and Daniel Lohmann. 2009. Dead or Alive: Finding Zombie Features in the Linux Kernel. In FOSD.
[56] Y. Tian, J. Lawall, and D. Lo. 2012. Identifying Linux bug fixing patches. In ICSE.
[57] G. S. Tseitin. 1983. On the Complexity of Derivation in Propositional Calculus. Springer Berlin Heidelberg, Berlin, Heidelberg, 466–483. https://doi.org/10.1007/978-3-642-81955-1_28
[58] Ying-Jie Wang, Liang-Ze Yin, and Wei Dong. 2021. AMCheX: Accurate Analysis of Missing-Check Bugs for Linux Kernel. Journal of Computer Science and Technology 36, 6 (Dec. 2021), 1325–1341.
[59] Yingfei Xiong, Arnaud Hubaux, Steven She, and Krzysztof Czarnecki. 2012. Generating range fixes for software configuration. In 34th International Conference on Software Engineering (ICSE).
[60] Y. Xiong, H. Zhang, A. Hubaux, S. She, J. Wang, and K. Czarnecki. 2015. Range Fixes: Interactive Error Resolution for Software Configuration. IEEE Trans. Softw. Eng. 41, 6 (June 2015), 603–619.
[61] Christoph Zengler and Wolfgang Küchlin. 2010. Encoding the Linux kernel configuration in propositional logic. In Proceedings of the 19th European Conference on Artificial Intelligence (ECAI 2010) Workshop on Configuration.