CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

A testing technique for conflict-resolution facilities in software configurators

Bachelor of Science Thesis in Software Engineering and Management

EVGENY GROSHEV

Department of Computer Science and Engineering
UNIVERSITY OF GOTHENBURG
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2020

CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

A testing technique for conflict-resolution facilities in software configurators

Supervisor: Thorsten Berger
Examiner: Richard Berntsson Svensson

University of Gothenburg
Chalmers University of Technology
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Department of Computer Science and Engineering
UNIVERSITY OF GOTHENBURG
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2020

A Testing Technique for Conflict-Resolution Facilities in Software Configurators

Evgeny Groshev
University of Gothenburg
Gothenburg, Sweden

1 INTRODUCTION

Variability modeling and feature models are important concepts in the context of software product lines, where a large number of individual products can be generated from the total set of available configuration options (features). Configuration tools utilise feature models to check validity of product configurations and detect anomalies such as conflicts, dead and false-optional features.

These research concepts have been shown to map well to the Linux kernel project and its feature modeling language Kconfig, which is also used by a number of other open-source projects [7, 8]. The Linux kernel configurators use Kconfig internally but lack support for automated conflict-resolution, therefore making kernel configuration a challenging and error-prone process [12].

The recent years have seen substantial progress in the development of conflict-resolution algorithms based on constraint satisfaction problems (CSP), and in translation of the input variability models into Boolean form required by such algorithms [9, 15, 21].

Research problem. However, little is known about evaluation of conflict-resolution algorithms for their use in the real-world configuration tools. We have the algorithms but we need a systematic way of ensuring that conflict fixes calculated by any such algorithm lead to new *valid* configurations — the opposite would not improve the current state, and possibly even make it worse. The need for additional studies has been identified with connection to the RangeFix algorithm [21] and its use for Kconfig [15]. Among others, identified research directions included evaluation of this algorithm involving real users [21], and its integration with the Linux kernel configuration tool [15].

Purpose of the study. To address this knowledge gap, we conduct a design science study that investigates how conflict-resolution algorithms can be evaluated for their use in software configurators.

We aim at coming up with an approach that will allow to evaluate such algorithms in an experimental setting. By evaluating we mean capturing data pertaining to the implemented functionality in order to measure and understand its correctness, and to improve its quality.

Research questions. The main research question addressed by this study is the following:

> ***RQ1.*** *How can conflict-resolution algorithms for software configurators be evaluated?*

This question is related to conceiving a suitable testing technique for conflict resolution. When such technique is in place, we hope to additionally address the following question:

> ***RQ2.*** *What is the correctness of the selected conflict-resolution implementation?*

To answer these questions we propose a black-box testing technique that allows evaluating conflict-resolution algorithms. We then instantiate this technique for a particular configurator (*xconfig*) and a particular algorithm (RangeFix), and evaluate it it practice.

Significance. As seen from the design science viewpoint, the research problem of this study can be linked to a larger *practical* problem [13] within the practice of software configuration. The practical problem is that of conflict-resolution for software configurators, as confirmed by Linux community survey results in [12]. Not only the practical problem mentioned above is related to this *global* practice [13] — its relevance has also been established for research areas of software product lines, feature modelling and variability analysis.

The anticipated results of this study may benefit researchers in the fields of variability modeling and automated variability analysis, and designers of configuration tools. Secondly, we expect it to be useful to the users and the developers of software configuration tools.

We hope that the results of this study will contribute to the configuration practice by improving conflict-resolution. We also hope that the results will useful for the ongoing research related to feature modeling, configuration tools, and conflict resolution.

Report structure. The remainder of this report provides a background to the research problem (Section 2), explains the used methodology (Section 3), and describes the proposed evaluation technique (Section 4.1). Its implementation in practice is demonstrated in Section 4.2, and Section 5 presents the results of evaluating the technique for testing the RangeFix algorithm. Section 6 discusses the limitations of the proposed technique together with possible directions for future research. Section 7 concludes the report.

2 BACKGROUND

2.1 Variability modeling

The research problem of this study fits into the context of software product line engineering, where *variability models* are used to describe common and variable aspects of the product *variants*, which may target different hardware, market segments, contexts of use etc. Having a way to model variability of a given product line allows to *automatically* validate its variant configurations (i.e. features selected therein), detect conflicts and dead features, and to support users of configuration tools (*configurators*) by implementing automated conflict resolution.

Feature modeling languages is one of the approaches to variability modeling, where available product features are expressed in tree-like structures. In such models, features can be mandatory and

optional, and configuration choices may be limited by group and cross-tree constraints.

Feature modeling is a long-lived topic within the research literature, with well established concepts like automated detection of model anomalies, validity checking of configurations, and conflict-resolution by decision propagation [6, 16].

Although well established in research, automated reasoning for resolution of configuration errors is not fully realised in real-world highly configurable systems [7]. Real-world configurators succeed at detecting configuration conflicts but challenge their users by presenting inconsistent resolution advice, making it hard to determine minimal configuration, and to enable inactive features [12].

2.2 Kconfig

The Linux kernel is an open-source project started by Linus Torvalds in 1991, aimed at creating a monolithic Unix-like operating system kernel that is compliant with POSIX and Single Unix Specification [4]. The project's main development language is C but other languages are used for scripting and build infrastructure (Perl, Python, GNU make).

Today, the Linux kernel has been ported to many different platforms including mainframe, server, smartphones, and embedded devices, and as such, the kernel represents a *highly-configurable software*. More that 500 operating system *distributions* based on the Linux kernel exist [3], which target different user groups, device types, and architectures. Moreover, it is possible to configure and build the kernel for a device of virtually any size and architecture. In that regard, the kernel is a *family of products*, and can be considered a software product line (SPL), despite that its development process, led by a "a loosely-knit team of hackers" [4], does not follow the SPL guidelines [19]. Its variation points include hardware architecture, device drivers, and specific features and implementations divided into subsystems, which can be additionally fine-tuned during the configuration process.

The Linux kernel project uses the *Kconfig language* to describe the available configuration options (*symbols*). The semantics of the language is defined only informally [1] though researchers attempted to formalise it [18] and to analyse its consistency [11]. Correspondence between feature modeling concepts and Kconfig was established in [20], and the analysis was expanded in [7].

The Kconfig syntax includes keywords that allow defining configuration options (`config`, `menuconfig`), structuring them (`menu`, `choice`, `if`), and displaying additional information (`mainmenu`, `comment`). The kernel feature model is modular, with more than 1400 files representing different subsystems and features linked in a top-down manner using the `include` keyword.

The kernel configuration options are organised into a tree structure, which the Kconfig documentation ambiguously calls "configuration database", "configuration files", and "menu"; the configuration options are often called "menu entries" [1]. This overlaps with other concepts and keywords and may cause a significant confusion for a novice.

Depending on their purpose, Kconfig menu entries can have various attributes to specify their type, default and allowed values, help message, input prompt, and dependencies on other symbols. Majority of the symbols have boolean and tristate logical value types, and strings and numerical values are used to a less extent.

A simple Kconfig example is given in Figure 1. In this configuration space, all symbols in group 2 can only be selected if symbol A, which defauls to "no", is enabled. Additionally, symbol D is only available if B is inactive, and C is selected.

```
mainmenu "Example menu"

menu "Group 1"
  config A
    bool "Option A"
    default n
  config B
    bool "Option B"
endmenu

menu "Group 2"
  depends on A
  config C
    tristate "Option C"
  config D
    bool "Option D"
    depends on !B && C

endmenu
```

Figure 1: Example Kconfig file

2.3 Kernel Configuration

The Kernel build system (Kbuild) is a collection of files and tools used for generating concrete kernel variants. It includes Kconfig files (describing the kernel configuration space), configuration tools (used to create a concrete kernel configuration), and makefiles (compiling the code based on given configuration and linking it into a runnable kernel image).

The first step needed in order to build a kernel is *kernel configuration* which results in a configuration file, called `.config`. It is important to distinguish between this file and the kernel configuration space (represented by Kconfig files), especially since both are sometimes misleadingly called "configuration".

Kbuild offers a number of configurators that rely on the same *backend* code for parsing Kconfig and `.config` files and dependency checking [20], but use different *frontends* that differ significantly with respect to the granularity of user choices. The options range from fully interactive graphical user interfaces (*xconfig*, *gconfig*) to more constrained text-based user interfaces (*menuconfig*, *config*) to fully automatic `.config` generation based on default choices (*localyesconfig*, *allnoconfig*) or even random values (*randconfig*) [2]. Technically, these configurators are separate executables and as such may demonstrate differences in the behaviour.

Besides relying on a default or random symbol selection, users may control the configuration process interactively, by manually

creating or reusing existing `.config` files, and by setting environment variables like `KCONFIG_ALLCONFIG` and `KCONFIG_PROBABILITY`, which would affect the automatic configurators.

When a valid `.config` file is present, it is possible to build an executable kernel module that will include the selected configuration options.

2.4 Conflict resolution

The Kconfig dependency checking algorithm is only sparsely documented. The documentation [1] provides rules for calculating symbol values and visibility, but omits implementation details that can be crucial for the user.

For example, the Kconfig backend maintains two values for every symbol: the externally set (for example, by the user or in a precreated `.config` file) and the calculated value (which is saved in `.config` when a configurator terminates). The external choices only propagate to the calculated values only in the absence of conflicts. This is definitely a desired property from the validity point of view but may be confusing from the usability perspective.

Consider an example configuration presented in Figure 2. According to the configuration space (Fig. 1), the symbols B and D cannot be selected at the same time, and the configurator will resolve this conflict by disabling and hiding symbol D, giving the user little choice on the matter.

```
CONFIG_A=y
CONFIG_B=y
CONFIG_C=y
CONFIG=D=y
```

Figure 2: Conflicting configuration

This silent disregard for some of the user choices without presenting options for conflict resolution poses a serious problem for the actual kernel model, with thousands of features and numerous cross-tree constraints.

On the other hand, even when a desired option is visible in the configurator, enabling it might still be a challenge due to complex dependencies. The configurator will prevent selecting a disabled symbol if that would lead to a conflict with the currently active feature selection — again, without providing clear guidance.

Lack of automated conflict-resolution and insufficient guidance were among the main kernel configuration challenges identified in the survey conducted by Hubaux et al. [12].

2.5 RangeFix

In this context, when the Kconfig dependency checker makes literal conflicts unlikely, conflict resolution means finding a solution that would allow enabling one or several symbols that are blocked by the current configuration. Such a solution would typically require changing the choice of several other symbols. One such algorithm, dubbed *RangeFix*, was proposed in [21], and its feasibility for use in kernel configuration was studied in [15].

One of the main advantages of this algorithm is producing a list of configuration options that must be changed in order to resolve a conflict — together with a *range* of values for every such option, allowing for for a more compact representation of conflict-resolution choices. The fixes themselves are generated based on unsatisfiable cores that can be calculated by a constraint satisfaction problem (CSP) solver.

Besides correctness, designers of the algorithms pursued the following properties for it [21]:

- Generating a complete set of fixes.
- Minimal number of options included in conflict fixes.
- Maximal range of option values leading to conflict resolution.

In this study, we focus on evaluating the RangeFix algorithm implementation that was developed by Patrick Franz during his master's thesis project [10].

3 METHODOLOGY

This study joins a larger ongoing design science research [10, 15], and follows the framework described in [13]. It focuses mainly on the evaluation activity of that framework, because the practical problem being addressed is well-understood (see Section 1). The high-level requirements for its solution have been elicited in [7, 12], and the more detailed requirements for RangeFix integration were reported in [15].

RQ1. To answer this question, we focus on evaluation of the selected RangeFix implementation, and to do that, we are developing a secondary artefact. We are instrumenting the *xconfig* Linux kernel configurator with a mechanism to introduce conflicts and to apply fixes produced by the conflict-resolution algorithm. We then use the internal dependency checking functionality of Kconfig to validate the generated fixes. This artefact will be demonstrated in section 4 of this report, and it will be evaluated by using it to answer RQ2.

RQ2. After completion of the artefact development, we plan to study the correctness of the RangeFix implementation by conducting an experiment. Since exhaustive testing of all possible Linux configurations is not feasible, we plan to use one of the available product sampling algorithms to generate a set of conflicting configurations [5, 14]. During this step, we plan to collect quantitative data about the correctness of the RangeFix implementation.

This data will be further analysed by calculating the conflict-resolution accuracy of the implementation. Additionally, depending on the initially discovered patterns, we may also employ some forms of regression or correlation analysis — for example, to identify trends in conflict-resolution depending on the characteristics of the product samples.

4 TESTING TECHNIQUE

We propose a technique for quality assurance of conflict-resolution algorithm implementations that target software configurators for highly configurable systems. After giving a conceptual description of the proposed technique we then instantiate it for a conflict-resolution feature based on the RangeFix algorithm developed by Xiong et al. [21]. Patrick Franz, Sarah Nadi, Thorsten Berger, and

Ibrahim Fayaz adopted, implemented [1], and integrated [2] this algorithm into one of the Linux kernel configurators, *xconfig* [10].

4.1 Conceptual Description

The proposed testing technique comprises the following distinct steps.

(1) *Sample configuration space.* Thorough testing of a conflict-resolution algorithm requires that we collect representative real-life configurations, or generate random configurations, where a configuration is a set of enabled features.
These configuration space *samples* should provide enough coverage of the features contained therein. This implies that all important variation points (such as hardware architecture) in the configuration space must be considered.

(2) *Introduce conflicts.* For every sample, a number of random configuration conflicts must be generated. Configuration conflicts are selections of candidate options together with their wanted values, where a candidate is a option whose value is locked by the current configuration. Changing the value of such candidate may be non-trivial and require *reconfiguration*, which involves other dependent options, — hence the need for *automatic* conflict resolution.
Therefore, when selecting conflict candidates we must check whether their values are locked. A reasonable way to introduce conflicts is to iterate the configuration space and randomly select a reasonable number of such candidates, and to obtain their *desired* values by inverting their *current* values.
Number of conflicts and their size can be adjusted based on the initial performance measurements of the implementation in order to find a balance between test coverage and feasibility.

(3) *Run the algorithm.* For every conflict, run the algorithm. The configuration sample and the conflict will be inputs to the algorithm, which will either produce one or several configuration fixes or report that the conflict cannot be resolved.

(4) *Validate fixes.* A fix is a list of configuration options together with their target values. Besides the conflicting options themselves, generated fixes will necessarily include some additional options whose values must be changed *in order* to set the former to the new, desired state. These additional options, which may be too hard to reconfigure manually, constitute the very nature of a configuration conflict.
For every fix returned by the algorithm, this step involves applying this fix to the sample configuration and verifying whether that it at least satisfies the following properties:
   (a) user needs shall be satisfied i.e. the conflicting options selected in step 2 will have received their target value;
   (b) product configuration after fix application shall be valid;
   (c) no unnecessary changes shall be made to the product configuration.

These steps convey the general requirements for application of the technique.

- A variability model of the configuration space, which will also most likely be a prerequisite for any serious conflict-resolution feature.
- A software configurator with reconfiguration support.
- An implementation of a conflict-resolution algorithm that is integrated in the configurator.

The next section demonstrates a concrete implementation of the proposed technique for a real-life software configurator.

4.2 Implementation for the Linux Kernel

We have used the approach described above to test an implementation of the RangeFix algorithm [10]. Below, we use the terms *algorithm* and *implementation* interchangeably, and they both mean the implementation in question. The steps taken to apply the testing technique in practice were affected by the properties of the Kconfig backend.

4.2.1 Sample configuration space. The Linux kernel configuration space is represented by Kconfig files in its source tree, and a concrete configuration can be created using some of the kernel configurators (see Section 2.3). Such configuration can be saved in a `.config` file and later re-used, which is exactly what we need for testing the algorithm. Obtaining *representative* real-life `.config` files is not feasible due to the large variability of target architectures, hardware platforms and components that must be covered. Instead, we obtain `.config` samples by using the *randconfig* configurator. We consider these samples representative, because *randconfig* is one of the officially supported configuration tools, and as such, we expect it to have some real-life usage together with other kernel configurators.

However, some kernel options depend on the target hardware architecture. By default kernel configurators use the architecture of a host computer, but the Linux kernel supports cross-compilation (on a host computer with architecture X for target architecture Y) by setting the `ARCH` environment variable [3]. In order to achieve proper test coverage, we utilise this feature to generate configuration samples for every supported architecture by the kernel. We set the `ARCH` to each of the 20+ supported architectures [4] before we run `make randconfig`, and we use a shell script to automate this process [5].

Table 1 shows the list of architectures for which we generated configuration samples, together with the number of options available in the corresponding configuration space. It is clear that the kernel has a separate configuration space for every supported architecture.

4.2.2 Introduce conflicts. Our initial idea was to introduce configuration conflicts by generating random `.config` files with conflicting options, and by using the Kconfig backend to parse these files, — in other words, generate both the configuration sample and the conflict therein at the same time. We were then going to apply the RangeFix algorithm to such invalid in-memory configuration. However, the properties of the Kconfig backend mentioned in Section 2.4 make it impossible to add conflicts in this manner without

[1] https://bitbucket.org/easelab/configfix
[2] https://github.com/vaasu/linux/tree/satconfig_gui_rebased_against_patrick_code_13jun2020
[3] https://www.kernel.org/doc/html/latest/kbuild/kbuild.html#arch
[4] https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/arch?h=v5.3
[5] https://bitbucket.org/easelab/configfix/src/cftestconfig/code/scripts/kconfig/gen_config.sh

Table 1: Number of the kernel options depending on the target architecture

Architecture	ARCH value	No. options
Alpha	`alpha`	14425
ARC	`arc`	14450
ARM 32-bit	`arm`	15871
ARM 64-bit	`arm64`	14825
TI TMS320C6x	`c6x`	14375
C-SKY	`csky`	14397
H8/300	`h8300`	14386
Qualcomm Hexagon	`hexagon`	14371
Intel Itanium (IA-64)	`ia64`	14664
Motorola 68000	`m68k`	14491
MicroBlaze	`microblaze`	14398
MIPS	`mips`	15117
NDS32	`nds32`	14423
Nios II	`nios2`	14409
OpenRISC	`openrisc`	14374
PA-RISC	`parisc`	14392
PowerPC	`powerpc`	15050
RISC-V	`riscv`	14416
IBM System/390 and z/Architecture	`s390`	14510
SuperH 32-bit, 64-bit	`sh, sh64`	14741
SPARC 32-bit, 64-bit	`sparc32, sparc64`	14546, 14545
User-Mode Linux	`um`	14446
x86 32-bit, 64-bit	`i386, x86_64`	14960, 14961
Xtensa	`xtensa`	14444

circumventing the dependency checker, which may be risky from the correctness point of view.

Instead, we opted for an alternative idea of introducing conflicts by imitating user actions. To do that, we need a configurator that allows the user to identify a currently unavailable option and invoke the RangeFix algorithm to get a conflict resolution suggestion. To that end, we use Ibrahim Fayaz's integration of the target implementation into *xconfig* [6] (Figure 3), and we additionally extend it with functionality to programmatically select features that conflict with the current configuration.

Conflict candidates considered by our implementation are limited to the menu entries that contain prompt text i.e. those for which *xconfig* will provide some guidance to the user. This is reasonable, since prompt-less options are often automatically activated by the prompt options using forward (`depends on`) and reverse (`select`) dependencies. Among such menu entries, conflict candidates are the options that cannot be currently changed (enabled or disabled). Configuration conflicts modelled in this way clearly correspond to user intent during kernel configuration — adding missing and removing unnecessary options — which might be blocked due to the dependencies between options.

[6] https://github.com/vaasu/linux/tree/satconfig_gui_rebased_against_patrick_code_13jun2020

4.2.3 Run the algorithm. After generating a conflict we simply reuse the available RangeFix integration programmatically. It invokes the algorithm in order to obtain configuration fixes.

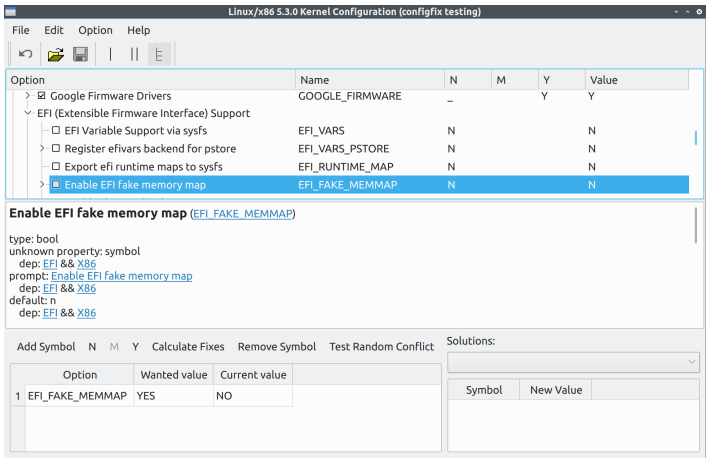

Figure 3: *xconfig* instrumented with interactive conflict-resolution and randomised testing

4.2.4 Validate fixes. Any configuration fix that the algorithm returns must be applied — all options included in it must receive the proposed target value. The resulting configuration must be then checked in order to verify that these fixes indeed resolve the conflict. Again, we simulate user actions by programmatically triggering GUI events that correspond to option selection and value change.

By nature of configuration conflicts any fixes that resolve them will contain additional options, besides the ones included the conflicts. One particular challenge related to such multi-option fixes is application order. Since fixes to Kconfig configurations cannot be applied as atomic transactions, they must be applied one option at a time. The RangeFix implementation under test produces fixes that include options in an arbitrary order, and, because Kconfig backend prevents illegal configuration states, a particular application path may result in new intermediate configuration conflicts [10].

We cope with such potential deadlocks, we do the following:

(1) Generate the permutations of the obtained fixes by enumerating the included options and using the function `std::next_iteration()` from the C++ standard library to permutate the option indices. We then rearrange the options in the fixes according to the index permutations.
(2) Iterate and attempt applying the fix permutations [7], and abandon those that result in new conflicts.
(3) To be able to safely reset the configuration to its initial (conflict-free) state, we implement a way to create its in-memory backup. After every unsuccessful application attempt we reload the configuration sample using the `conf_read()` function in the Kconfig backend. After that we compare the reloaded configuration with the backup to verify the successful reset.

[7] See the function `apply_fix_bool()` in https://bitbucket.org/easelab/configfix/src/cftestconfig/code/scripts/kconfig/kconfig-sat/rangefix.c

A fix is considered *valid* if it can be fully applied in an order provided by some of its permutations. When all permutations have been tested and rejected, the fix is deemed *invalid*.

Figure 3 displays the *xconfig* configurator with integrated RangeFix implementation and user interface elements that allow users to create conflicts by adding and removing options, and setting their target values. In the example, the currently inactive option `EFI_FAKE_MEMMAP` is selected, and it is marked to be enabled. By clicking the *Calculate Fixes* button, the user is able to run the conflict-resolution algorithm and obtain configuration fixes.

We added to this implementation the *Test Random Conflict* button, which streamlines the testing process. After loading a previously generated configuration sample into *xconfig*, it allows to perform steps (2)-(4) of our technique (Section 4.1) with a single click.

The source code of our implementation of the proposed testing technique is avaiable online [8].

5 EVALUATION RESULTS

The purpose of evaluation in design science research is to confirm whether the developed artefact can resolve or mitigate the practical problem at hand [13].

To evaluate our technique we performed limited testing of the RangeFix implementation. Due to time constraints we could only test a limited number of single-option conflicts for a selection of the architectures supported by the Linux kernel.

The tests were performed on a Lubuntu 19.10 virtual machine in a VirtualBox 6.1.12 hypervisor running on a MacBook Pro 11,3 host with 2.3 GHz Intel Core i7 quad-core CPU and 16 Gb RAM. The guest machine was allocated with 2 host CPU cores and 10 Gb RAM.

Table 2 presents a summary of the test results. The full set of test data and the test results are available online [9].

The main obstacle faced during our testing effort was resetting the configuration to its initial status between testing of different fix permutations. Although it worked well during initial testing of our implementation, it did not perform well during volume testing. In most cases, we could not reset the configuration, which made thorough validation of fixes impossible. Only 14 of the 177 solutions (7.9%) could be validated — and only because the first application attempt succeeded.

This stresses the importance of studying the internal state of the configurators not only for those working with evaluation of conflict-resolution features, but also for the developers of such facilities, since configuration rollback is likely to be a critical requirement prior to mass adoption of any such feature.

Even though we could not fully validate the fixes found by the algorithm, we did, however, find 14 conflicts for which the algorithm did not find *any* fixes. We additionally discovered an odd fix that consisted of a single option — the conflict itself! — which is clearly a bug. All this provides valuable input to the developers of the conflict-resolution feature that we tested.

In general, despite the current implementation shortcomings for the chosen system and algorithm, our technique looks promising for anyone who wants to get a quick feedback on the quality of the conflict-resolution implementations, and do not have the time or the resources to invest in more expensive verification efforts (for example, using formal methods). As such, we believe this technique to promise a good return on investment.

Table 2: RangeFix Testing Statistics

Metric	Value
Architectures	OpenRISC, PA-RISC, RISC-V SPARC 32-bit, SPARC 64-bit x86 32-bit, x86 64-bit System/390, Xtensa
No. configuration samples	15
Conflict size	1
No. conflicts	75
Min. resolution time	1.3 sec
Max. resolution time	304.3 sec
Mean resolution time	47.2 sec
No. resolved conflicts	61
Efficiency (% resolved conflicts)	81.3%

6 LIMITATIONS AND FUTURE WORK

We see two main threats to the validity of this study.

Its external validity might be affected by the fact that we concentrate on a single feature modeling language (Kconfig) and its use within a single project in a single domain (Linux kernel, operating systems). However, we do not consider this risk as high — in fact, previous studies have often concentrated on the Linux kernel because its feature model makes it somewhat more representative than the purely academic models [8]. Moreover, since Kconfig is not used exclusively in Linux, other practice and research communities might benefit from our findings.

Due to the lack of knowledge about the Linux kernel, we generated its test samples purely algorithmically, without using additional inputs such as information about the feature distribution among Linux variants, or expert knowledge. This might affect the internal validity of the study.

A recent study reports that scaling of product sampling algorithms to large real-world product lines is a challenge [17]. Since Linux kernel, with its thousands of features, is definitely a large-scale product line, this may affect the outcome of our study. One possible way to address this would be to evaluate the RangeFix implementation using an earlier version of the kernel with fewer features. While this would definitely call for additional evaluation with a more recent version, this still may be enough to answer the research questions posed here.

One possible future study direction is related to integration of the RangeFix algorithm into the Kconfig backend, and using the algorithm during dependency checking. This will allow to evaluate the algorithm in context of all available kernel configurators, and might shed new light upon both the correctness and the usability perspectives.

[8] https://bitbucket.org/easelab/configfix/src/cftestconfig
[9] https://github.com/izimbra/configfix_test

7 CONCLUSION

Our study demonstrates that evaluation of conflict-resolution features in software configuration tools is indeed a complex problem. The proposed testing technique allowed us to gain insights into the workings of a black-box conflict-resolution algorithm, and gather some statistics about its effectiveness, correctness, and performance.

However, more often than not our testing efforts faced impediments related to technique implementation for the chosen modelling language and configurator (Kconfig and *xconfig*). Though specific to the Linux kernel, these obstacles shed some light on the areas that require additional attention by the researchers who study software configuration.

Together with the research directions identified in Section 6, we believe that these results will be of general interest to both the researchers and the practitioners.

STATEMENT OF COLLABORATION

This study would not be possible without the work previously conducted by Patrick Franz, who implemented the RangeFix algorithm in the C programming language, and Ibrahim Fayaz, who integrated this implementation into *xconfig* configurator.

REFERENCES

[1] [n.d.]. *Kconfig language.* Retrieved May 30, 2020 from https://www.kernel.org/doc/html/latest/kbuild/kconfig-language.html

[2] [n.d.]. *Kconfig make config.* Retrieved May 30, 2020 from https://www.kernel.org/doc/html/latest/kbuild/kconfig.html

[3] [n.d.]. *LWN Distribution List.* https://lwn.net/Distributions/

[4] 2017. *About Linux Kernel.* Retrieved May 30, 2020 from https://www.kernel.org/linux.html

[5] Mustafa Al-Hajjaji, Sebastian Krieter, Thomas Thüm, Malte Lochau, and Gunter Saake. 2016. IncLing: efficient product-line testing using incremental pairwise sampling. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences - GPCE 2016.* ACM Press, New York, New York, USA, 144–155. https://doi.org/10.1145/2993236.2993253

[6] David Benavides, Sergio Segura, and Antonio Ruiz-Cortés. 2010. Automated analysis of feature models 20 years later: A literature review. *Information Systems* 35, 6 (9 2010), 615–636. https://doi.org/10.1016/j.is.2010.01.001

[7] Thorsten Berger, Steven She, Rafael Lotufo, Andrzej Wąsowski, and Krzysztof Czarnecki. 2010. Variability modeling in the real. In *Proceedings of the IEEE/ACM international conference on Automated software engineering - ASE '10.* ACM Press, New York, New York, USA, 73. https://doi.org/10.1145/1858996.1859010

[8] 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 (12 2013), 1611–1640. https://doi.org/10.1109/TSE.2013.34

[9] David Fernandez-Amoros, Ruben Heradio, Christoph Mayr-Dorn, and Alexander Egyed. 2019. A kconfig translation to logic with one-way validation system. In *Proceedings of the 23rd International Systems and Software Product Line Conference - volume A - SPLC '19.* ACM Press, New York, New York, USA, 1–6. https://doi.org/10.1145/3336294.3336313

[10] Patrick Franz. 2020. *Realising Configuration Conflict-Resolution for the Linux Kernel Configurator.* Master's thesis. Chalmers University of Technology and University of Gothenburg.

[11] Stefan Hengelein. 2015. *Analyzing the Internal Consistency of the Linux KConfig Model.* Master's thesis. University of Erlangen. https://www4.cs.fau.de/Ausarbeitung/MA-I4-2015-04-Hengelein.pdf

[12] Arnaud Hubaux, Yingfei Xiong, and Krzysztof Czarnecki. 2012. A user survey of configuration challenges in Linux and eCos. In *Proceedings of the Sixth International Workshop on Variability Modeling of Software-Intensive Systems - VaMoS '12.* ACM Press, New York, New York, USA, 149–155. https://doi.org/10.1145/2110147.2110164

[13] Paul Johannesson and Erik Perjons. 2014. *An Introduction to Design Science.* Springer International Publishing, Cham. https://doi.org/10.1007/978-3-319-10632-8

[14] Martin Fagereng Johansen, Øystein Haugen, and Franck Fleurey. 2011. Properties of Realistic Feature Models Make Combinatorial Testing of Product Lines Feasible. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics).* 638–652. https://doi.org/10.1007/978-3-642-24485-8_47

[15] Daniel Jonsson. 2016. *A Case Study of Interactive Conflict-Resolution Support in Software Configuration.* Master's thesis. Chalmers University of Technology and University of Gothenburg. https://hdl.handle.net/20.500.12380/238168

[16] Jens Meinicke, Thomas Thüm, Reimar Schröter, Fabian Benduhn, Thomas Leich, and Gunter Saake. 2017. *Mastering Software Variability with FeatureIDE.* Springer International Publishing, Cham. https://doi.org/10.1007/978-3-319-61443-4

[17] Tobias Pett, Thomas Thüm, Tobias Runge, Sebastian Krieter, Malte Lochau, and Ina Schaefer. 2019. Product sampling for product lines. In *Proceedings of the 23rd International Systems and Software Product Line Conference - volume A - SPLC '19.* ACM Press, New York, New York, USA, 1–6. https://doi.org/10.1145/3336294.3336322

[18] Steven She and Thorsten Berger. 2010. Formal Semantics of the Kconfig Language. Technical Note. http://www.eng.uwaterloo.ca/~shshe/kconfig_semantics.pdf

[19] Julio Sincero, Horst Schirmeier, Wolfgang Schröder-Preikschat, Olaf Spinczyk, and Friedrich-Alexander. 2007. Is The Linux Kernel a Software Product Line?. In *Proc. SPLC Workshop on Open Source Software and Product Lines.*

[20] Julio Sincero and Wolfgang Schröder-Preikschat. 2008. The Linux Kernel Configurator as a Feature Modeling Tool. In *First International Workshop on Analysis of Software Product Lines (ASPL'08).*

[21] Yingfei Xiong, Hansheng Zhang, Arnaud Hubaux, Steven She, Jie Wang, and Krzysztof Czarnecki. 2015. Range Fixes: Interactive Error Resolution for Software Configuration. *IEEE Transactions on Software Engineering* 41, 6 (6 2015), 603–619. https://doi.org/10.1109/TSE.2014.2383381