Published at the
43rd International Conference on Software Engineering (ICSE 2021), Software Engineering in Practice (SEIP) track

ConfigFix: Interactive Configuration Conflict Resolution for the Linux Kernel

Patrick Franz*, Thorsten Berger†*, Ibrahim Fayaz‡, Sarah Nadi§, Evgeny Groshev*
*Chalmers | University of Gothenburg †Ruhr University Bochum ‡VecScan AB (Vector Sweden) §University of Alberta

***Abstract*—Highly configurable systems are highly complex systems. The Linux kernel is arguably one of the most well-known examples. Given its vast configuration space, researchers have used it to conduct many empirical studies as well as to build dedicated methods and tools for analyzing, configuring, testing, optimizing, and maintaining the kernel. However, despite a large body of work, mainly bug fixes that were the result of such research made it back into the kernel's source tree. Unfortunately, Linux users still struggle with kernel configuration and resolving configuration conflicts, since the kernel largely lacks automated support. Additionally, there are technical and community requirements for supporting automated conflict resolution in the kernel, for example, using a pure C-based solution that uses only compatible third-party libraries (if any).**

With the aim of contributing back to the Linux community, we present CONFIGFIX, a tooling that we integrated with the Linux kernel configurator, that is purely implemented in C, and that is finally a working solution able to produce fixes for configuration conflicts. We describe our experiences of building upon the large body of research done on the kernel configuration mechanisms as well as how we designed and realized CONFIGFIX while adhering to the Linux kernel's community requirements and standards. CONFIGFIX not only helps Linux kernel users obtain their desired configuration, but our implemented semantic abstraction provides the basis for many of the above techniques supporting kernel configuration.

***Index Terms*—software configuration, semantic abstraction, conflict resolution, Linux kernel**

I. INTRODUCTION

The Linux kernel is the world's largest software development project [20] by the number of its contributors. Being highly versatile, the kernel operates in a diversity of environments, ranging from Android phones to large supercomputer clusters. As such, it is not only a successful operating-system kernel, but also a *highly configurable system* [67]—nowadays boasting 28 million lines of code [47] and over 15,000 configuration options (a.k.a., *features* [**?**], [11]). To this end, the kernel relies on mechanisms known from the fields of software product lines [8], [18], model-driven engineering [25], and software configuration [13]. Specifically, the Linux kernel includes a configurable build system [16], preprocessor-enabled variation points, a model-based representation of configuration options and their constraints (a.k.a., *variability model*) [17], [51], and an interactive configurator tool [67]. Being completely open source, with a vast evolution history available, researchers have studied many different aspects of it, including software evolution [35], [76], [7], [**?**] and software maintenance [73], [34], [5], [36] aspects, as well as its configuration mechanisms—the focus of this paper.

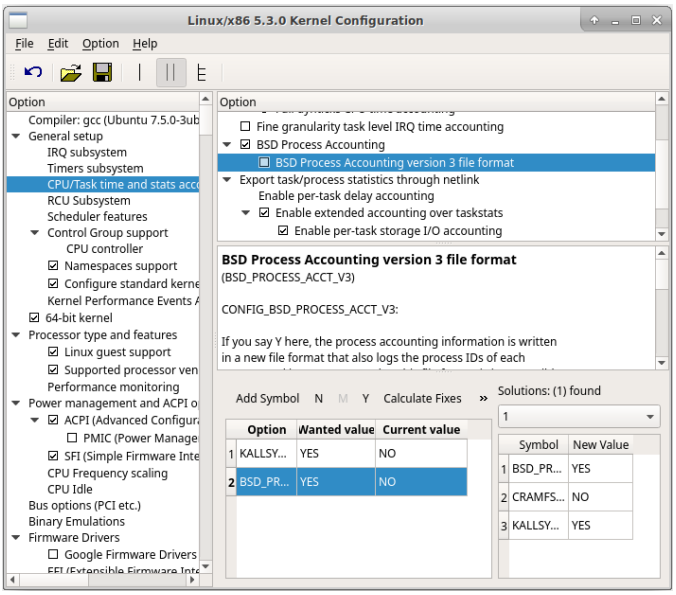

Fig. 1. The kernel configurator extended with CONFIGFIX

Studies of the Linux kernel's configuration mechanisms started back in 2007 [66], [67], [71], followed in 2010 by our and other researchers' studies of its variability modeling language Kconfig and its variability model [64], [17]. Examples include the evolution of this model [49], the co-evolution and consistency of variation points [57], [**?**], [52], [53], [40], [70], as well as the synthesis of variability models from code [65], [51]. Despite all the above research efforts related to configuration, users of the kernel did not benefit directly yet. They still struggle with creating their desired configuration [33], given the huge configuration space and intricate constraints among features. Beyond a simple and very limited support for choice propagation, the configurator does not offer any intelligent support for resolving configuration conflicts. Often, enabling a feature requires transitively changing many others. As such, achieving the desired configuration can be laborious and error-prone, which is unfortunate given all the work in the research community that never made it back into the kernel.

Only few contributions originating from research made it back into the kernel, and mainly in the form of bug fixes [56], [70], [52]. While there are tools such as Coccinelle [48] and Undertaker [69] that become known and often individually adopted among kernel developers, none were formally integrated into the kernel codebase or are listed as a necessary kernel tool. In 2015, the situation was about to change with the Kconfig-sat

initiative [41], where kernel developers recognized the need and got in touch with researchers working on kernel configuration studies, including us. Given our experience with studying the Linux kernel's configuration information and developing tools to analyze it, we decided it was time to give back to the Linux community and try to practically integrate these techniques into the kernel configurator. In fact, implementing a sound translation of the variability model to propositional logics (which is a semantic abstraction) given the expressiveness and intricate semantics of the Kconfig language (explained shortly), has long been an open problem. Multiple translations were proposed [63], [3], [39], but each has its own shortcomings [29].

In this paper, we describe experiences over a decade of different efforts on reverse-engineering the formal semantics from the Linux kernel configurator and implementing sound semantic abstractions. These are the prerequisite for many techniques and use cases supporting software configuration. After providing an overview on the Linux kernel's configuration tooling, research efforts on it, and their historical perspective, we introduce our tool CONFIGFIX. It offers intelligent configuration support for the Linux kernel configurator and realizes a pure-C implementation of a semantic abstraction and a technique to resolve configuration conflicts. Upon a current configuration (an assignment of features to values) and a set of features whose values the user wants to change, CONFIGFIX calculates *fixes* to reach the desired configuration. We built on our and others' prior work in the field to realize a translation of the kernel's variability model into propositional logics, to implement a configuration-conflict resolution algorithm relying on solving satisfiability problems (SAT), and to integrate both into the graphical configurator tool *xconfig*. Our work led to finally obtaining a viable solution integratable in the kernel's source tree. CONFIGFIX is freely available [4], together with details about its evaluation.

II. SOFTWARE CONFIGURATION AND THE LINUX KERNEL

We now briefly introduce the field of software configuration and the Linux kernel's configuration facilities.

A. Software Configuration

Software configuration is concerned with methods to configure software, originally stemming from the field of product configuration, a subfield of AI [32], [59]. The challenge is to obtain a configuration that meets end-user requirements, considering all constraints among the configuration options (i.e., *features*). The configuration process is typically supported with an interactive configurator tool, offering support for propagating choices and resolving configuration conflicts. Software configurators [9], [10] have been studied in many domains [18], [77], [14], including configurable systems software (e.g., Linux kernel, eCos operating system or 3D printer firmware [66], [17], [45], [15], [63]), automotive [31], avionics [61], and telecommunication systems [68], embedded and safety-critical software [74], [43], [12], [60], as well as web-based configuration [6].

B. The Linux Kernel and its Configurator

The Linux kernel's configurability aims at customizing the kernel beyond its core functionality of CPU & memory management towards many different hardware architectures (ranging from embedded devices to supercomputer architectures) and including optional functionality, such as device or filesystem drivers. Currently, over 15,000 configuration options (henceforth called *features* [**?**], [11]), control variation points in C source files using conditional compilation directives (e.g., `#if`) of the C preprocessor. These features also control the inclusion of individual files in the build process. In addition to this static mechanism, many features control *loadable kernel modules* (e.g., network or USB drivers) that can be loaded dynamically at runtime.

Users configure the kernel interactively via its configurator, which exists in three variants. Figure 1 shows the graphical configurator *xconfig*. The other two variants target shell users. All features come with default values, and users can then assign values to the individual features according to their types and constraints, establishing a configuration. The features, their organization in a hierarchy, and their constraints are declared in files using the Kconfig language, which are input to the configurator.

Xconfig supports basic validity checking of configuration choices as well as simple imperative choice propagation. The latter, given the absence of an intelligent reasoner, needs to be encoded with a dedicated imperative mechanism in Kconfig (explained shortly), which is error-prone and, given its imperative nature, cannot be used to resolve configuration conflicts. In contrast, various open-source [50], [28], [15] and commercial [19], [44] configurators come with a reasoner.

C. The Kconfig Language

At the core of the Linux kernel configurator is Kconfig—a domain-specific language for variability modeling. Originally created for Linux, it has since been adopted by at least ten other open-source projects (e.g., BusyBox) [17]. A core challenge in the research community was obtaining a sound logical representation of the main semantics of Kconfig as a prerequisite to develop analyses and configuration techniques. However, Kconfig is surprisingly expressive with exceptionally intricate semantics.

Language Concepts. Kconfig comes with a textual syntax and concepts known from feature modeling (a popular kind of variability modeling language) [38], [27], [54]: a *hierarchy of features*, different *feature types*, *feature groups* (e.g., OR, XOR or MUTEX groups), and *cross-tree constraints* [17]. The main semantics of a feature model describe the set of all possible configurations. Since feature modeling languages are typically limited to Boolean features and constraints, they can easily be translated into propositional logics.

Kconfig's syntax and semantics go well beyond feature modeling. For scaling the variability model and configuration process, Kconfig incorporates concepts such as visibility conditions (to conditionally show whole subtrees), modularization concepts, derived defaults and features, hierarchy manipulation), and an expressive constraint language including comparison, arithmetic, and String operators. Interestingly, features can also inherit constraints of their parents in non-transparent ways. Furthermore, Kconfig has a domain-specific vocabulary (main keywords) that fosters comprehension among Linux developers.

Features can be of different types: `bool`, `tristate`, `string`, `hex`, and `int`. Tristate features control the binding mode of features and can have three values: **y** (yes, compile into kernel), **n** (no, do not compile) or **m** (mod, compile as loadable kernel module). Their evaluation follows Kleene's rules for three-state logics [42]. Intuitively, the value of a tristate feature is encoded as 0, 1 or 2. The logical operators are then defined over numbers: && returns the minimum and || the maximum of the two operands, and ! returns 2 minus the operand.

All these concepts substantially complicate Kconfig's syntax and semantics. In fact, many intricate semantic interactions between different language elements exist—most notably between seven language constructs to express constraints (`prompt`, `default`, `depends on`, `select`, `imply`, `visible if`, and `range`). For instance, a default value becomes a constraint when the feature is not visible, as determined by other constraints. Further details are in Kconfig's official documentation [58] and our prior work [63], [17], including our reverse-engineered denotational semantics of Kconfig. Finally, Kconfig is continuously extended with language constructs, such as recently with the statement `imply` (gitlab.freedesktop.org/panfrost/linux/commit/237e3ad0) as a special case of the `select` statement used for imperative choice propagation. The latter is significantly driving the complexity of the Kconfig semantics. The `select` statement interacts with other Kconfig elements in intricate ways. This in fact complicated our and others' efforts realizing a sound semantic abstraction. We believe it would have been better to keep the language cleaner and separately implement choice propagation via a reasoner, as CONFIGFIX does.

In summary, Kconfig's complexity results from the design of its configurator tooling. Instead of performing language engineering [26], [46] and adopting a configurator that comes with more intelligent reasoning capabilities [9], we learned that the community prefers transparent and easily scriptable solutions as opposed to heavy machinery, such as off-the-shelf reasoners that may be difficult to understand. We learned this preference from the discussion on the kernel mailing list preceding the introduction of Kconfig and its tooling. An alternative candidate was a configurator tool and language with built-in conflict-resolution support, which the community explicitly decided against.

Kconfig Language and Configurator Design

Kconfig is a popular language, but surprisingly expressive, coming with intricate syntax and semantics. The kernel community preferred the script-style xconfig and Kconfig over more systematically engineered tooling, mainly to be able to fully control and evolve the tooling.

Language Semantics and Abstractions. Motivated by prospective empirical insights and being able to evaluate research prototype tools, researchers started looking into Kconfig and its tooling back in 2007 [66], [67], [71]. This was followed up with holistic studies of Kconfig, its tooling, and its models in 2010 [64], [17]. For instance, we were the first to formally describe the semantics of Kconfig, which allowed its translation into propositional logic both by ourselves and other researchers [63], which resulted in the first translation tool called LVAT [62]. Around the same time, Zengler and Küchlin also provided a translation into propositional logics [80], [29]. Furthermore, the tool Undertaker [3] identifies dead `#ifdef` code (whose constraints conflict with the variability model) and also came with a translation into propositional logics. As part of the TypeChef infrastructure [40], Kästner implemented KConfigReader [39]. Recently, Fernandez-Amoros et al. [30] proposed another translation which, however, ignores Tristate features (i.e., a large part of the semantics).

A translation into SMT was created by Xiong et al. [79], who presented RangeFix—a technique to generate configuration conflict fixes for non-propositional configuration spaces. For non-Boolean features, it provides ranges to which the value needs to be changed to resolve a conflict. Our fix generation is conceptually based on RangeFix, but has simplifications since we do not need all computation steps.

A translation not originating from researchers exists as well. As part of Google's Summer of Code, Vegard Nossum contributed Satconfig [55], which comes with a pure-C translation and allows reasoning via the SAT solver PicoSAT [24], for instance, completing a configuration based on an initial, partial configuration. We previously investigated Satconfig [37], but found shortcomings in the handling of tristate features, leading to incorrect fixes; there was also limited documentation of the implementation. Still, Satconfig was fast, indicating that a C-based solution can be scaled to the size of the kernel's varibility model. It also showed the advantages of running a SAT solver directly in the configurator tool, as well as the feasibility of implementing the translation in pure-C. Its design inspired our data structures.

With the exception of one tool [30], all produce propositional formulas in conjunctive normal form (required by SAT solvers) and typically apply a Tseitin transformation [75], which introduces auxiliary variables to avoid formula explosion.

A systematic comparison [29] of LVAT [62], Undertaker [3], and KConfigReader [39] showed that all had shortcomings in their abstraction of Kconfig's semantics. Notably, KConfigReader could correctly handle most of Kconfig's semantics, which steered our decision to re-implement KConfigReader's translation in C with some deviations.

Kconfig Semantic Abstraction

Over the last decade, multiple researchers and one practitioner implemented propositional abstractions for Kconfig—to provide the basis for SAT-based reasoning, system analysis techniques, and guiding users configuring the kernel. None of these abstractions was fully sound and complete, given the complexity of the Kconfig language.

The Kconfig-SAT Initiative. In October 2015, the Kernel developer Luis R. Rodriguez contacted researchers including us who have worked on SAT-based configuration support for the Linux kernel and Linux package management. After discussing configuration issues related to Kconfig, and after being made

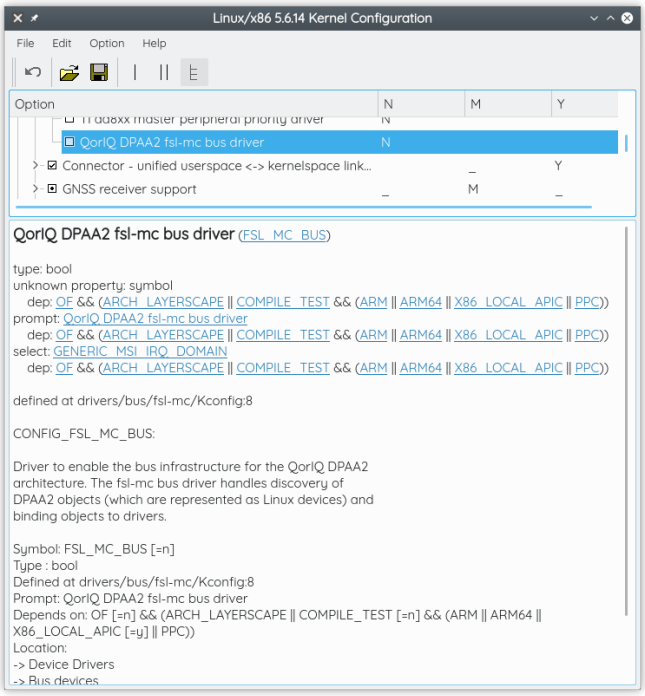

Fig. 2. A configuration conflict in *xconfig*

aware of our research, he launched the Kconfig-SAT initiative (with a wiki page [41] and a mailing list [2]). Kconfig-SAT is now also described in the Kconfig documentation [58].

Among the kernel community, the awareness for needing intelligent configuration support rose, despite some skepticism about SAT solvers by Linus Torvalds: *"The SAT solver will only hurt, because it will bring in all those irrelevant people who are interested in SAT solving, not in making things easy for users"* (lists.linuxfoundation.org/pipermail/ksummit-discuss/2017-June/004499). Overall, our interaction via the mailing list was insightful, as it helped understanding the community and obtaining requirements. For instance, in addition to providing sound fixes, a solution should be fast for user acceptance [37].

III. ConfigFix

We introduce the notion of configuration conflict, give a practical introduction into ConfigFix, and discuss its design.

A. Configuration Conflicts

A conflict arises when changing the value of one or multiple features violates a constraint. To resolve conflicts, users need to (transitively) follow the dependencies, which is laborious and error-prone. A survey [33] found that users are commonly challenged with conflict resolution in the kernel configurator, with 20 % of the survey respondents needing roughly "a few dozen minutes" to resolve a conflict. The feature descriptions only provide incomplete and sometimes hard to understand (or even incorrect) advice, leading to users blindly choosing default or recommended values without grasping the consequences. Furthermore, default values sometimes contradict with the advice, for instance, when the description recommends enabling a feature, but the default value is "no" (disabled). As such, resolving conflicts can be particularly challenging and frustrating for inexperienced users, who then simply resort to guessing.

Let us illustrate configuration conflicts with a feature that has particularly complex constraints. Figure 2 shows the feature "QorIQ DPAA2 fsl-mc bus driver" (a bus infrastructure driver) in xconfig. Currently, it cannot be enabled (checkbox not selectable, indicated by missing underscores in columns "M" and "Y") due to the following unmet constraint (the current values of the involved features are shown at the bottom of Fig. 2):

```
depends on OF && (ARCH_LAYERSCAPE ||
          (COMPILE_TEST && (ARM || ARM64 ||
          X86_LOCAL_APIC || PPC)))
```

Xconfig does not even show which parts of the constraint are unmet. The user needs to manually resolve the conflict by transitively looking at the features' dependencies, taking the full richness of its constraint language into account, which might mean needing to enable and disable a set of many features at the same time. Doing so, however, might have ripple effects by triggering the imperative choice propagation (select statement) in *xconfig*, which might in turn invalidate the user's resolution. The user might not even realize what other features the imperative choice propagation is enabling or disabling. Depending on constraints, the respective feature might not even be visible to the users in the configurator, further challenging the configuration process. The goal of ConfigFix is to facilitate this conflict resolution process by automatically finding the feature values needed to reach a desired configuration.

B. ConfigFix Overview

Workflow. Figure 3 shows the overall workflow of ConfigFix from the end user's perspective. (1) shows the view the user sees once they launch *xconfig* and have chosen "Show All Options" from the Options menu. This option shows features that would have normally been hidden, because they have unmet dependencies. The user can then identify the features they wish to enable, and click on "Add Symbol" which will add the feature to the bottom left pane, as shown in (2). At that point, the user can change the value of the feature to "Y", "M" or "N". The option "M" is greyed out for this feature since the feature cannot be compiled as a loadable kernel module ("M"=module). Instead, it will be compiled directly into the kernel binary ("Y"=yes) or not at all ("N"=no). Once the user has added all the features they would like to change the values for, they can click on "Calculate Fixes," which will call ConfigFix's internal conflict resolution algorithm with the list of features specified in the bottom left pane. The returned solutions will be displayed in the bottom right pane as shown in (3). Each solution is a set of feature values that need to be applied in order to set the wanted value(s) for the feature(s) indicated in (2). As there can be multiple ways to satisfy the user constraints, each specific solution can be viewed by selecting it from the Solutions combobox. The user can then apply any of the solutions, either by letting ConfigFix apply all changes automatically or by changing every feature manually.

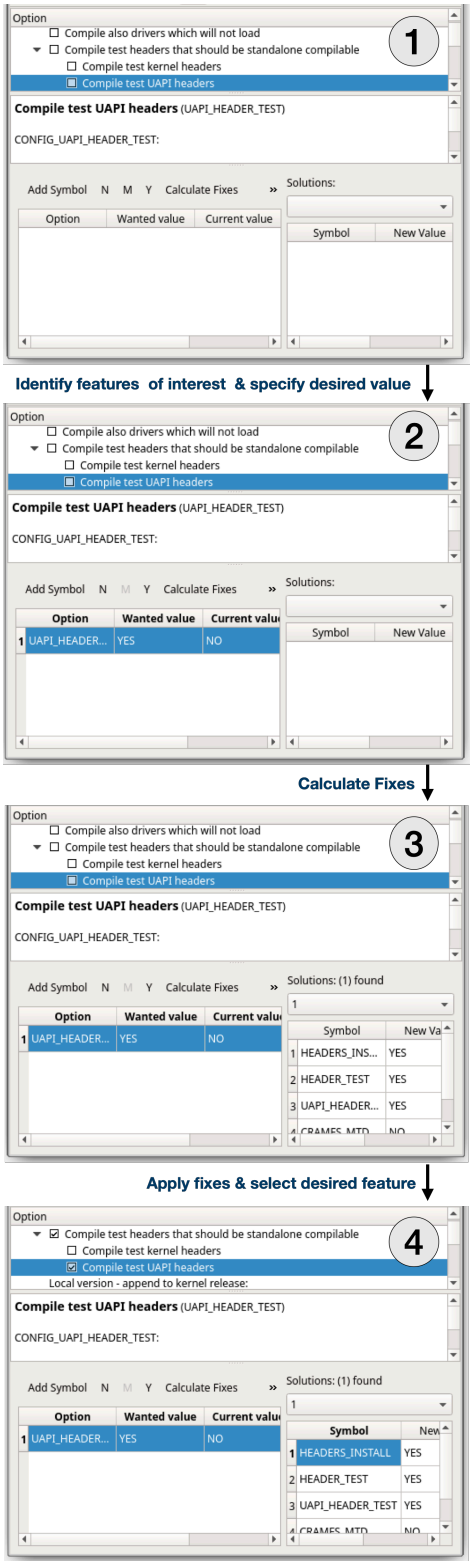

Fig. 3. User's workflow using CONFIGFIX inside *xconfig*

Technical Details. Currently, CONFIGFIX depends on the library GLib [72] which needs to be installed before *xconfig* can be started. There is, however, no need to either install or set up a SAT solver for this purpose as this is done automatically.

Given the demand, CONFIGFIX also offers an export into DIMACS [1]—a file format accepted by many SAT solvers. The export can be launched via make cfoutconfig.

C. Solution Overview

Figure 4 illustrates CONFIGFIX's fix calculation. First, CONFIGFIX translates the Kconfig model into a propositional formula according to the Kconfig semantics we reverse-engineered and re-implemented (and discuss shortly in Sec. IV-A). It then translates this formula into conjunctive normal form (CNF) to be able to later process it with a SAT solver. The formula encoding the Kconfig variability model represents the *hard constraints* that must always be satisfied. In other words, CONFIGFIX cannot violate any of these constraints during its fix generation. As input, CONFIGFIX also takes in the current configuration (a list of features and their corresponding values) and the user's configuration goal (a list of features to change and their desired values). CONFIGFIX considers the current configuration as *soft constraints*, since some of the current feature values in the configuration will need to change. As the next step, CONFIGFIX creates a single formula that is a conjunction of all hard and soft constraints and then queries the SAT solver for satisfiability. If it is satisfiable, then there is nothing to find fixes for, and the desired values can be applied. Otherwise, CONFIGFIX triggers our C-based RangeFix implementation to calculate fixes.

D. Fix Generation

Given a conflict, we want to find a fix that requires a minimal number of changes to the configuration. In the literature, various conflict-resolution algorithms exist [32], but many are not applicable here. They either produce only one fix, a long list of fixes (challenging users to identify/apply the most suitable one), or they only offer limited support for non-Boolean constraints. We selected *RangeFix* [79], which was designed to mitigate these shortcomings. Its fixes adhere to three main properties: *Correctness* (any configuration resulting from a fix must be correct, i.e. satisfy the violated constraints), *Maximality of ranges* (when ranges overlap, the fix shall contain a maximum range), and *Minimality of changes* (the number of features that need to be changed should be reasonably minimal—realized using heuristics we defined—to avoid unnecessarily breaking feature values set by the user before). RangeFix generates fixes in three stages. Its input is an abstraction of a variability model (e.g., CNF formula), where features are represented by variables. In stage 1, all minimal sets of variables that have to be changed, are generated. These sets are called diagnoses. In stages 2 and 3, the new values for each variable in a diagnosis are calculated.

We now explain the algorithm using an example. Let us define the tuple (V, e, c) as a *constraint violation*, where V is a set of variables, e a configuration with a value defined for each variable and c a set of constraints which is violated. The set of constraints c represents hard constraints and the set of variables V and their configuration e represent the soft constraints. The goal is to find a new configuration e', such that c is satisfied. We define our set of variables V as:

$$\{m : Boolean, a : Integer, b : Integer\}$$

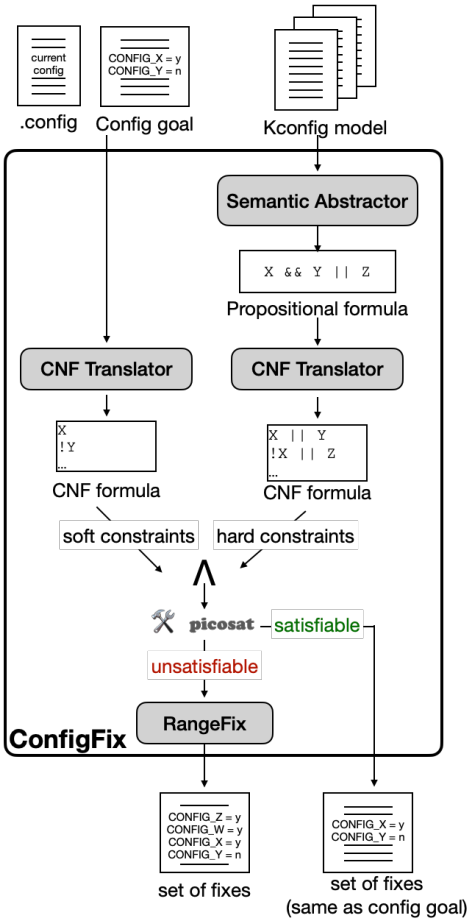

Fig. 4. Overview of ConfigFix's components

We define a set of constraints as:

$$(m \rightarrow a > 10) \ \wedge (\neg m \rightarrow b > 10) \ \wedge (a < b)$$

Finally, a configuration e with values for the three variables is required: $\{m := True, a := 6, b := 5\}$. This configuration violates the first and third constraints.

RangeFix generates the diagnoses during the first stage by using a constraint solver's ability to find unsatisfiable cores. During each iteration in the first step, a constraint from an unsatisfiable core is removed and the diagnoses are extended until no more unsatisfiable core is found. Applying this to our example yields the following two diagnoses: $\{m, b\}$ and $\{a, b\}$. At this point, we know that we need to change either $\{m, b\}$ or $\{a, b\}$ to resolve the conflict.

The stages 2 and 3 are subsequently performed together for each diagnosis. First, all unchanged variables are replaced by their current values during stage 2. Finally, in stage 3, the violated constraints are minimized via heuristic rules we defined and split into minimal clauses to generate the fixes. This leads to the following two fixes for the conflict:

- $[m := False, b : b > 10]$
- $[(a, b) : a > 10 \wedge a < b]$

If any of these two fixes is applied, all previously violated constraints will be satisfied again. If the first fix is chosen, then m needs to be set to $False$ and b simply to any value larger than 10. If the second fix is chosen, then a needs to be set to a value larger than 10, and b must simply be larger than a. This illustrates the advantages of RangeFix compared to other conflict-resolution algorithms. While there are infinitely many possible solutions, the user is presented with only the minimal set. Others might have presented only one of the two solutions or a long list of possible solutions, since there are infinitely many possible combinations that satisfy a being smaller than b in the second fix. As such, CONFIGFIX returns a maximum of three fixes for each conflict even if more fixes exist. In most cases, presenting three fixes for a user is sufficient to choose a suitable fix and generating more fixes can take significantly more time.

E. Example Fix

Given a default configuration, assume a user wants to enable the feature `MEDIA_TUNER_SIMPLE`. The feature has dependencies, and its parent feature depends on other features via visibility conditions. In the default configuration, the parent is hidden and the dependencies are not met, so it is not configurable by the user.

For this conflict, eight possible fixes exist and each fix changes the values of five features, although some values change implicitly through Kconfig's imperative choice propagation (`select` statements). Generally, the feature can be enabled through two different means: The parent feature can be made visible and then the user can explicitly set a value for `MEDIA_TUNER_SIMPLE` after its dependencies have been met. The other possibility is to enable the feature through a `select` statement while keeping it invisible.

In the first case, there are four possible fixes that make the feature visible such that the user can then set an explicit value. These four fixes only differ in one symbol and one such fix is the following:

- `MEDIA_SUPPORT => yes`
- `MEDIA_DIGITAL_TV_SUPPORT => yes`
- `MEDIA_SUBDRV_AUTOSELECT => no`
- `MEDIA_TUNER_SIMPLE => yes`

In the latter case, there are again four possible fixes only differing in a single symbol. One such fix ensuring that the feature is selected and, therefore, enabled is:

- `MEDIA_SUPPORT => yes`
- `MEDIA_ANALOG_TV_SUPPORT => yes`

The feature is invisible, though, and it might not be obvious to the user why the feature has been enabled. In the end, all eight fixes calculated are able to enable `MEDIA_TUNER_SIMPLE` in one way or another.

IV. EXPERIENCES AND CHALLENGES

We now discuss experiences and challenges faced when realizing CONFIGFIX. Overall, our first attempt [37] was much closer to the RangeFix implementation. We experimented with the C-based translation by Vegard Nossum (cf. Sec. II-C) and an existing Scala-based RangeFix implementation prototypically integrated into *xconfig*, but which only covered the first of the three stages of RangeFix. We surveyed kernel developers and Kconfig-SAT members about the user interface we implemented, and provided screencasts illustrating the

solution [37]. The results contributed to the present attempt, where we focused on a purely C-based translation and fix generation, the C-based SAT solver Picosat [23], and an improved integration into *xconfig*.

A. Semantic Abstractor

The largest challenge was obtaining a sound and stable logical abstraction of Kconfig. To obtain requirements, we interacted with the community, specifically via the kconfig-sat mailing list. We recognized a strong preference for SAT solvers, as opposed to more expressive solvers, such as SMT [21]. Even though, the latter could support a larger part of the semantics, it was pointed out that integrating a SAT solver could also help at other places in the kernel, especially CPU scheduling support. Furthermore, SMT solvers are typically slower, and not many come with a GPL-compatible license, as required for integration into the kernel's codebase. In order to make the integration into the various kernel configurators as easy as possible, we also decided that it should be possible to compile the SAT solver with the tools needed to compile every kernel configurator, such as *gcc*. This requirement excluded some modern and very fast solvers, including CaDiCaL [22].

In general, our options for realizing a propositional semantic abstraction for a SAT solver were to (i) develop a new translation from scratch, to (ii) build on *Satconfig* (cf. Sec. II-C) or to (iii) investigate other existing alternatives. Strategy (i) has the disadvantage of lacking a reference, when the translation is checked for correctness. Since correctness was essential, we disregarded this strategy. As explained above, Satconfig showed deficiencies in handling `tristate` features and in code documentation. Based on others' systematic comparison of translations [29] (also cf. Sec. II-C), we chose to re-implement and extend the Scala-based *KConfigReader* [39].

Our overall strategy was to inspect the translation in KConfigReader, to re-implement it freely in C, being inspired by data structures from Satconfig, and to test the translation incrementally with smaller, hand-crafted models, thereby also debugging and fixing our implementation as well as remaining deficiencies of KConfigReader. Finally, we tested with example conflicts in the full kernel model (we described one of these conflicts in Sec. III-E). KConfigReader works accurately for `bool` and `tristate` features, as well as for many non-Boolean properties, but with some remaining smaller limitations.

Adapting and re-implementing KConfigReader in C allowed the direct integration into the kernel configurator in the kernel source tree. It also allowed us to use the configurator's parser to parse the variability model, which provides robustness as the parser is regularly maintained. We traversed the internal, AST-based representation and stored intermediate results in our own C data structures. A challenge was to implement scalable representations of the propositional formula in C, in a way that it can be easily traversed and transformed into conjunctive normal form by applying logical laws and a Tseitin transformation. As a consequence, some parts of the translation had to be implemented in a completely different way than in KConfigReader.

To account for limitations, we needed to deviate from the semantics realized in KConfigReader. One example was the translation of the imperative choice propagation—over 10,000 `select` statements exist in the whole variability model. So, how to model this type of constraint has substantial impact on the performance. KConfigReader models the `select`-constraints in a single constraint for the feature that is selected. An advantage of this behavior is the number of constraints, since there will only be one constraint for each selected feature. The disadvantage is that the constraints can potentially become very large formulas, when a feature is selected by many other features. As these formulas need to be converted into CNF, large formulas can easily result in hundreds or thousands of CNF clauses.

B. Choosing a SAT Solver

Choosing PicoSAT [23] was inspired by Satconfig (cf. Sec. II-C), which showed that PicoSAT can be easily integrated into the kernel. More importantly, it: (i) is written in C and can be compiled with `gcc`; (ii) has a C-API, so can be called directly within *xconfig* without needing external calls; (iii) has a Linux-compatible license (MIT); (iv) can identify and return unsatisfiable cores; and (v) is reasonably fast [24]. This made PicoSAT our best candidate. However, a downside was that the solver is outdated and not actively maintained anymore, despite still being used in industry. Still, it has a well-structured implementation and is fast enough for interactive fix generation.

C. GUI Integration

We extended the graphical configurator *xconfig* to provide an intuitive interface for entering the desired feature values, for observing the proposed fixes, and for applying the desired ones. We decided on having the conflict resolution integrated within the same window to follow conflict resolution interfaces found in other configurators, specifically that of ECOS [78], [33], [17]. We added a new pane at the bottom of *xconfig* to collect the features desired by the user. The view is divided into two parts: (i) the collected feature list and (ii) the solutions CONFIGFIX produces.

D. Scalability and Performance Improvements

The main challenge was translating the huge formulas of the full kernel variability model with over 15,000 features into CNF. Notably, the encoding of certain Kconfig aspects had a substantial impact on the resulting CNF and SAT solver performance.

To improve the performance of the SAT solver, we changed the encoding of the `select` statement by splitting up the various statements Instead of a single constraint for a selected feature, each `select` statement now creates constraints on its own. But, since this also had an affect on how we encoded the dependencies of a feature, introducing a new variable was needed, which indicates whether a feature has been selected. This significantly simplified the constraint encoding.

While we have increased the number of constraints in total by several thousand constraints, we were able to reduce the number of CNF clauses and auxiliary variables significantly. As

a consequence, a single run of PicoSAT became more than 65 % faster than it was before. A disadvantage of this decision is, that we lost the ability to syntactically check our constraints for equivalence against the constraints produced by KConfigReader. Still, the gain in performance justified this decision, and we conceived an alternative evaluation, explained shortly.

We achieved a final translation time of the entire Linux kernel variability model into a CNF formula of around 1.5 seconds on an Intel i7 laptop. The first run of PicoSAT to check for satisfiability takes about 2.5 to 3 seconds. Finally, finding fixes for a conflict can be achieved in as little as 1 second in some cases, although the number depends heavily on the conflict and the number of enabled features. The most impact on improving the performance came from incorporating our domain/expert knowledge into the translation, including effective formula splitting and using a Tseitin transformation.

V. Evaluation

We now discuss our evaluation of ConfigFix.

A. Conflict and Fix Generation

With over 15,000 features, the configuration space for the Linux kernel is huge and, therefore, crafting a small number of examples to be evaluated is not sufficient. Instead, a more systematic approach is needed. We make use of the kernel's ability to generate random configurations, and then we create random conflicts for each of those.

To obtain sufficiently diverse test data, we use the kernel's `randconfig` tool to generate configuration samples for three of the more popular architectures supported by the kernel: `x86_64`, `arm64`, and `openrisc`. This tool also allows to skew the probabilities for the features of type `bool` and `tristate` to be enabled; we create configuration samples for probabilities ranging from 10 % to 90 % for a feature to be set to `no`. These steps allow us to evaluate the performance of ConfigFix for varying proportions of enabled features in configurations on different architectures.

For each random configuration, we then introduce conflicts by randomly choosing target features that: (i) have a prompt (i.e., are not completely invisible, like derived features [17]), (ii) are of type `bool` or `tristate`, (iii) are not a choice group, and (iv) have at least one other possible value that is different from the current one and cannot be selected at the moment. In the case of `tristate` options with two values that cannot be selected at the moment, the target value is randomly chosen from these two values.

While finding fixes, ConfigFix may not return results for two reasons. First, a fix may not exist. Some features depend on other architecture-specific options; therefore, they can only be configured for certain architectures. Second, ConfigFix may also not return fixes due to bugs in our implementation or design. In order to objectively evaluate ConfigFix, we, therefore, want to ensure that our chosen target features can be configured (i.e., the conflict can indeed be resolved) for the used architecture to rule out the first possibility. To achieve this, we generate a base configuration for each architecture using `randconfig` with the probability of 100% for a feature to be enabled. Such base configuration will have as many enabled features as possible. For each configuration sample, we restrict the selection of conflicting features to those that can receive their target values on the chosen architecture, as witnessed by the base configuration. We create such resolvable conflicts containing between 1–10 features and let ConfigFix generate fixes for each conflict.

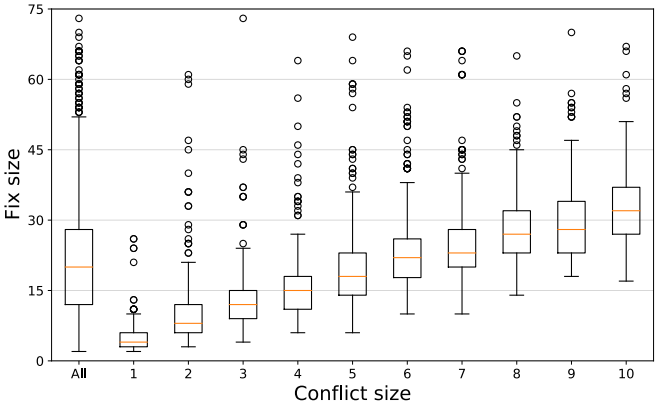

Fig. 5. Distribution of fix sizes (number of features that ConfigFix indicates need to be changed) related to conflict size (number of unchangeable features that a user wants to change). The first box shows the distribution of fix size across all conflict sizes. Diagram cut-off at 75 (more outliers exist).

B. Results

We generated 27 random configuration samples with a varying propotion of enabled features for the three target architectures (`x86_64`, `arm64`, and `openrisc`). For each configuration, we created 50 conflicts containing between 1–10 features to change. This resulted in 1,350 conflicts for ConfigFix to solve. We ran the tests on a Lubuntu 19.10 virtual machine allocated with 2 CPU cores and 10 GB RAM, on a macOS host with an Intel i7-4850HQ CPU @ 2.3 GHz and 16 GB RAM. We imposed a time limit of 300 seconds for fix generation. ConfigFix took between 1.8–355.1 seconds per fix (mean of 70.6 and median of 41.7 seconds).

Table I summarizes the results. Out of the 1,350 conflicts, ConfigFix returned at least one fix for 1,055 (78.2 %) conflicts and resolved 723 (53.6%) conflicts. It did not produce any fixes for the remaining 295 conflicts mostly due to timeouts. For the 1,055 conflicts that received at least one fix, ConfigFix produced a total of 2,482 fixes (recall our intentional limitation to a maximum of three fixes per conflict, as discussed in Sec. III-D). The obtained fixes comprise 2–175 features that needed to be changed.

Fig. 5 shows the distribution of fix size for each conflict size (cut-off at a fix size of 75). The distributions are left-skewed, with an average fix size of 23 features, and the majority of the fixes lying around the median of 20 features. The figure shows that in general, fix sizes demonstrate a seemingly linear dependence on the conflict size. This indicates that ConfigFix can be used for resolving conflicts of varying sizes without the risk of fix size explosion.

TABLE I
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	1,055 (78.2 %)
number of resolved conflicts	723 (53.6 %)
total number of fixes produced by CONFIGFIX	2,482 (100.0 %)
fixes that resolve the conflict	**1,609 (65.0 %)**
fully applicable and resolves conflict	1,317 (53.0 %)
not fully applicable, but resolves conflict	292 (12.0 %)
does not resolve conflict	868 (35.0 %)

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

We also analyzed the outcome of each fix, summarized in Table I. The optimal outcome—*fully applicable and resolves conflict*—was achieved for 1,317 of these fixes (53 %). A non-optimal, but acceptable outcome—*not fully applicable, but resolves conflict*—was achieved for 292 fixes (12 %). This means some of the values specified in the fix cannot be applied, but it resolves the conflict nonetheless. It contains invalid or redundant feature values, but the target features, which are part of the fix, can be changed. Finally, 868 (35 %) fixes were invalid—*does not resolve conflict*. So, in summary, 65 % of the fixes returned by CONFIGFIX resolved the conflict.

While the ideal outcome would have been to resolve all conflicts, we believe that this percentage is still acceptable given that the semantic abstraction is sound, but cannot be complete. While perhaps an encoding into SMT could yield a slightly better result, recall that there was a strong preference amongst the Linux community for SAT solving using a C-based solver.

VI. THREATS TO VALIDITY

Internal Validity. As KConfigReader cannot parse recent kernels anymore, we relied on testing our semantic abstraction with small hand-crafted examples and then compare the output of both tools. We created those examples to cover as much as possible of the Kconfig semantics. However, the kernel boasts over 15,000 configuration options and we cannot be sure that our examples cover all the semantics used in the kernel.

Construct Validity. We used artificial randomly generated conflicts to evaluate CONFIGFIX. We do not know if these generated conflicts are representative of conflicts that developers face in practice. However, our evaluation involved the same configurators that are used by the Linux kernel users and our conflicts, despite being randomly generated, originate from the same variability model that these users would use. Additionally, we perform the evaluation across several kernel architectures to make sure our results are not skewed towards a single architecture. In the future, we plan to conduct an evaluation that involves real Linux kernel users to determine if CONFIGFIX can generate useful fixes for them.

External Validity. Our work is focused on the configuration of the Linux kernel. We did not try our approach on other projects that use Kconfig (E.g., BusyBox). While expanding to other systems is feasible, we intentionally focus on supporting the Linux kernel, given its wide-spread usage and importance in modern computer systems. We believe our results are relevant not only to the Linux practitioners but also to the research community—since the kernel is a more representative example of a highly configurable system than purely academic models [17].

VII. CONCLUSION

We reported experience of leveraging results from 13 years of academic research. We created the tool CONFIGFIX providing interactive configuration conflict resolution support for the Linux kernel, by adhering to all requirements coming from the Linux community within the Kconfig-SAT initiative [41]. CONFIGFIX realizes a transformation of the configurator's underlying language Kconfig into a propositional abstraction, as well as it provides a configuration-conflict resolution technique that can guide users achieving their desired configuration. We believe that our tool [4] can help the Linux kernel community not only supporting the configuration process, but also conduct further analyses of the kernel's configuration facilities. Likewise, we invite researchers to evaluate their novel techniques upon the transformation, as well as to improve the fix generation (e.g., optimize for certain quality attributes).

REFERENCES

[1] "DIMACS," www.satcompetition.org/2009/format-benchmarks2009.html.
[2] "kconfig-sat," https://groups.google.com/forum/#!forum/kconfig-sat.
[3] "Undertaker," https://vamos.informatik.uni-erlangen.de/trac/undertaker.
[4] "ConfigFix," https://bitbucket.org/easelab/configfix, 2020.
[5] I. Abal, C. Brabrand, and A. Wasowski, "42 variability bugs in the linux kernel: A qualitative analysis," in *ASE*, 2014.
[6] E. Abbasi, A. Hubaux, M. Acher, Q. Boucher, P. Heymans, A. Heymans, F. FSR, and W. Region, "What's in a web configurator? empirical results from 111 cases," University of Namur, Tech. Rep., 2012.
[7] G. Antoniol, U. Villano, E. Merlo, and M. Di Penta, "Analyzing cloning evolution in the linux kernel," *Information and Software Technology*, vol. 44, no. 13, pp. 755–765, 2002.
[8] S. Apel, D. Batory, C. Kästner, and G. Saake, *Feature-Oriented Software Product Lines*. Springer, 2013.
[9] R. Bashroush, M. Garba, R. Rabiser, I. Groher, and G. Botterweck, "Case tool support for variability management in software product lines," *ACM Comput. Surv.*, vol. 50, no. 1, Mar. 2017.
[10] D. Benavides, S. Segura, and A. Ruiz-Cortés, "Automated analysis of feature models 20 years later: A literature review," *Information Systems*, vol. 35, no. 6, pp. 615–636, 2010.
[11] T. Berger, D. Lettner, J. Rubin, P. Grünbacher, A. Silva, M. Becker, M. Chechik, and K. Czarnecki, "What is a feature? a qualitative study of features in industrial software product lines," in *SPLC*, 2015.
[12] T. Berger, D. Nair, R. Rublack, J. M. Atlee, K. Czarnecki, and A. Wasowski, "Three cases of feature-based variability modeling in industry," in *MODELS*, 2014.
[13] T. Berger, R.-H. Pfeiffer, R. Tartler, S. Dienst, K. Czarnecki, A. Wasowski, and S. She, "Variability mechanisms in software ecosystems," *Information and Software Technology*, vol. 56, no. 11, pp. 1520–1535, 2014.
[14] T. Berger, R. Rublack, D. Nair, J. M. Atlee, M. Becker, K. Czarnecki, and A. Wasowski, "A survey of variability modeling in industrial practice," in *VaMoS*, 2013.
[15] T. Berger and S. She, "Formal semantics of the CDL language," 2010, technical Note. Available at http://thorsten-berger.net/cdl_semantics.pdf.
[16] T. Berger, S. She, R. Lotufo, K. Czarnecki, and A. Wasowski, "Feature-to-code mapping in two large product lines," in *SPLC*, 2010.
[17] T. Berger, S. She, R. Lotufo, A. Wasowski, and K. Czarnecki, "A Study of Variability Models and Languages in the Systems Software Domain," *IEEE Trans. Softw. Eng.*, vol. 39, no. 12, pp. 1611–1640, Dec. 2013.

[18] T. Berger, J.-P. Steghöfer, T. Ziadi, J. Robin, and J. Martinez, "The state of adoption and the challenges of systematic variability management in industry," *Empirical Software Engineering*, vol. 25, pp. 1755–1797, 2020.

[19] D. Beuche, "Variants and variability management with pure::variants," in *SPLC*, 2004.

[20] S. Bhartiya, "Linux is the largest software development project on the planet: Greg Kroah-Hartman," http://cio.com/article/3069529.

[21] A. Biere, M. Heule, H. van Maaren, and T. Walsh, *Handbook of Satisfiability: Volume 185 Frontiers in Artificial Intelligence and Applications*. Amsterdam, The Netherlands, The Netherlands: IOS Press, 2009.

[22] A. Biere, "CaDiCaL Simplified Satisfiability Solver," http://fmv.jku.at/cadical.

[23] ——, "PicoSAT," http://fmv.jku.at/picosat.

[24] ——, "Adaptive Restart Strategies for Conflict Driven SAT Solvers," in *SAT*, 2008.

[25] M. Brambilla, J. Cabot, and M. Wimmer, *Model-driven software engineering in practice*. Morgan & Claypool, 2017.

[26] B. Combemale, R. France, J.-M. Jézéquel, B. Rumpe, J. Steel, and D. Vojtisek, *Engineering modeling languages: Turning domain knowledge into tools*. CRC Press, 2016.

[27] K. Czarnecki and U. W. Eisenecker, *Generative Programming: Methods, Tools, and Applications*, 2000.

[28] D. Dhungana, P. Grünbacher, and R. Rabiser, "The DOPLER Meta-Tool for Decision-Oriented Variability Modeling: a Multiple Case Study," *J. ASE*, vol. 18, no. 1, pp. 77–114, Mar. 2011.

[29] S. El-Sharkawy, A. Krafczyk, and K. Schmid, "Analysing the kconfig semantics and its analysis tools," in *GPCE*, 2015.

[30] D. Fernandez-Amoros, R. Heradio, C. Mayr-Dorn, and A. Egyed, "A kconfig translation to logic with one-way validation system," in *SPLC*, 2019.

[31] R. Flores, C. Krueger, and P. Clements, "Mega-Scale Product Line Engineering at General Motors," in *Proc. SPLC*, 2012.

[32] A. Hubaux, D. Jannach, C. Drescher, L. Murta, T. Männistö, K. Czarnecki, P. Heymans, T. Nguyen, and M. Zanker, "Unifying software and product configuration: A research roadmap," in *ConfWS*, 2012.

[33] A. Hubaux, Y. Xiong, and K. Czarnecki, "A User Survey of Configuration Challenges in Linux and eCos," in *VaMoS*, 2012.

[34] A. Israeli and D. G. Feitelson, "Characterizing software maintenance categories using the linux kernel," The Hebrew University of Jerusalem, Tech. Rep. 2009–10, 2009.

[35] ——, "The linux kernel as a case study in software evolution," *Journal of Systems and Software*, vol. 83, no. 3, pp. 485–501, 2010.

[36] Y. Jiang, B. Adams, and D. M. German, "Will my patch make it? and how fast? case study on the linux kernel," in *MSR*, 2013.

[37] D. Jonsson, "A case study of interactive conflict-resolution support in software configuration," Master's thesis, Chalmers University of Technology, 2016.

[38] K. Kang, S. Cohen, J. Hess, W. Nowak, and S. Peterson, "Feature-oriented domain analysis (FODA) feasibility study," Carnegie-Mellon University, Pittsburgh, PA, USA, Tech. Rep., 1990.

[39] C. Kästner, "KConfig Reader," https://github.com/ckaestne/kconfigreader.

[40] C. Kästner, P. G. Giarrusso, T. Rendel, S. Erdweg, K. Ostermann, and T. Berger, "Variability-aware parsing in the presence of lexical macros and conditional compilation," in *OOPSLA*, 2011.

[41] Kernelnewbies, "Linux kconfig SAT integration," https://kernelnewbies.org/KernelProjects/kconfig-sat, accessed: 2019-12-05.

[42] S. C. Kleene, "On notation for ordinal numbers," *The Journal of Symbolic Logic*, vol. 3, no. 4, pp. 150–155, 1938.

[43] C. W. Krueger, D. Churchett, and R. Buhrdorf, "Homeaway's transition to software product line practice: Engineering and business results in 60 days," in *SPLC*, 2008.

[44] C. W. Krueger and P. C. Clements, "Systems and software product line engineering with biglever software gears," in *SPLC*, 2013.

[45] J. Krueger, W. Gu, H. Shen, M. Mukelabai, R. Hebig, and T. Berger, "Towards a better understanding of software features and their characteristics: A case study of marlin," in *VaMoS*, 2018.

[46] R. Lämmel, *Software languages: Syntax, semantics, and metaprogramming*. Springer, 2018.

[47] M. Larabel, "The Linux Kernel Enters 2020 At 27.8 Million Lines In Git But With Less Developers For 2019," https://www.phoronix.com/scan.php?page=news_item&px=Linux-Git-Stats-EOY2019.

[48] J. Lawall and G. Muller, "Coccinelle: 10 years of automated evolution in the linux kernel," in *USENIX ATC*, 2018.

[49] R. Lotufo, S. She, T. Berger, K. Czarnecki, and A. Wasowski, "Evolution of the Linux Kernel Variability Model," in *SPLC*, 2010.

[50] J. Meinicke, T. Thüm, R. Schröter, F. Benduhn, T. Leich, and G. Saake, *Mastering Software Variability with FeatureIDE*. Springer, 2017.

[51] S. Nadi, T. Berger, C. Kästner, and K. Czarnecki, "Where do configuration constraints stem from? an extraction approach and an empirical study," *IEEE Trans. Softw. Eng.*, vol. 41, no. 8, pp. 820–841, 2015.

[52] S. Nadi and R. C. Holt, "Make it or break it: Mining anomalies from linux kbuild," in *WCRE*, 2011, pp. 315–324.

[53] ——, "Mining kbuild to detect variability anomalies in linux," in *CSMR*, 2012.

[54] D. Nešić, J. Krüger, S. Stănciulescu, and T. Berger, "Principles of Feature Modeling," in *FSE*, 2019.

[55] V. Nossum, "satconfig," https://github.com/vegard/linux-2.6/tree/v4.7+kconfig-sat, accessed: 2019-12-05.

[56] Y. Padioleau, J. Lawall, R. R. Hansen, and G. Muller, "Documenting and automating collateral evolutions in linux device drivers," *SIGOPS Oper. Syst. Rev.*, vol. 42, no. 4, p. 247–260, Apr. 2008.

[57] L. Passos, R. Queiroz, M. Mukelabai, T. Berger, S. Apel, K. Czarnecki, and J. Padilla, "A study of feature scattering in the linux kernel," *IEEE Trans. Softw. Eng.*, vol. 47, pp. 146–164, 2021.

[58] L. Passos, L. Teixeira, N. Dintzner, S. Apel, A. Wasowski, K. Czarnecki, P. Borba, and J. Guo, "Coevolution of variability models and related software artifacts," *Empirical Softw. Engg.*, vol. 21, no. 4, pp. 1744–1793, Aug. 2016.

[59] R. Zippel et al., "kconfig-language.txt," in the kernel tree at kernel.org.

[60] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*. Prentice Hall, 2010.

[61] K. Schmid, I. John, R. Kolb, and G. Meier, "Introducing the pulse approach to an embedded system population at testo ag," in *ICSE*, 2005.

[62] D. C. Sharp, "Reducing avionics software cost through component based product line development," in *DASC*, 1998.

[63] S. She, "LVAT," https://github.com/shshe/linux-variability-analysis-tools.

[64] S. She and T. Berger, "Formal semantics of the kconfig language," 2010, technical Note. www.eng.uwaterloo.ca/ shshe/kconfig$_s emantics.pdf$.

[65] S. She, R. Lotufo, T. Berger, A. Wasowski, and K. Czarnecki, "The variability model of the linux kernel," in *VaMoS*, 2010.

[66] ——, "Reverse engineering feature models," in *ICSE*, 2011.

[67] J. Sincero, H. Schirmeier, W. Schröder-Preikschat, and O. Spinczyk, "Is The Linux Kernel a Software Product Line?" in *SPLC-OSSPL*, 2007.

[68] J. Sincero and W. Schröder-Preikschat, "The linux kernel configurator as a feature modeling tool," in *ASPL*, 2008.

[69] M. Svahnberg and J. Bosch, "Evolution in software product lines: Two cases," *Journal of Software Maintenance*, vol. 11, no. 6, pp. 391–422, Nov. 1999.

[70] R. Tartler, C. Dietrich, J. Sincero, W. Schröder-Preikschat, and D. Lohmann, "Static analysis of variability in system software: The 90,000 #ifdefs issue," in *USENIX ATC*, 2014.

[71] R. Tartler, D. Lohmann, J. Sincero, and W. Schröder-Preikschat, "Feature consistency in compile-time-configurable system software: Facing the linux 10,000 feature problem," in *EuroSys*, 2011.

[72] R. Tartler, J. Sincero, W. Schröder-Preikschat, and D. Lohmann, "Dead or alive: Finding zombie features in the linux kernel," in *FOSD*, 2009.

[73] The GNOME Project, "GLib Reference Manual," https://developer.gnome.org/glib/unstable/, accessed: 2021-01-27.

[74] Y. Tian, J. Lawall, and D. Lo, "Identifying linux bug fixing patches," in *ICSE*, 2012.

[75] P. Toft, D. Coleman, and J. Ohta, "A cooperative model for cross-divisional product development for a software product line," in *SPLC*, 2000.

[76] G. Tseitin, "On the complexity of derivation in propositional calculus," *Zapiski Nauchnykh Seminarov LOMI*, vol. 8, 01 1983.

[77] Q. Tu and M. W. Godfrey, "Evolution in open source software: A case study," in *ICSM*, 2000.

[78] F. J. van der Linden, K. Schmid, and E. Rommes, *Software Product Lines in Action*, 2007.

[79] B. Veer and J. Dallaway, "The eCos component writer's guide," the eCos component writer's guide, Available from http://ecos.sourceware.org/docs-2.0/cdl-guide/cdl-guide.html.

[80] Y. Xiong, H. Zhang, A. Hubaux, S. She, J. Wang, and K. Czarnecki, "Range fixes: Interactive error resolution for software configuration," *IEEE Trans. Softw. Eng.*, vol. 41, no. 6, pp. 603–619, June 2015.

[81] C. Zengler and W. Küchlin, "Encoding the linux kernel configuration in propositional logic," in *ECAI*, 2010.