Analysing the Kconfig Semantics and Its Analysis Tools

Sascha El-Sharkawy[1], Adam Krafczyk[2], Klaus Schmid[1]

[1]Software Systems Engineering, Institute of Computer Science,
University of Hildesheim, Germany
{elscha, schmid}@sse.uni-hildesheim.de

[2]Software Systems Engineering, Institute of Computer Science,
University of Hildesheim, Germany
{krafczyk}@uni-hildesheim.de

Please cite this publication as follows:
Sascha El-Sharkawy, Adam Krafczyk, and Klaus Schmid. "Analysing the Kconfig Semantics and Its Analysis Tools". In: *Proceedings of the 2015 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE'15)*. ACM, 2015, pp. 45–54. DOI: 10.1145/2814204.2814222.

The corresponding BibTeX-entry is:

```
@INPROCEEDINGS{El-SharkawyKrafczykSchmid15,
  author = {Sascha El-Sharkawy and Adam Krafczyk and Klaus Schmid},
  title  = {Analysing the Kconfig Semantics and Its Analysis Tools},
  booktitle={Proceedings of the 2015 ACM SIGPLAN International Conference on Generative Programming: Concepts
  publisher = {ACM},
  year  = {2015},
  pages = {45--54},
  doi   = {10.1145/2814204.2814222}
}
```

Analysing the Kconfig Semantics and Its Analysis Tools

Sascha El-Sharkawy
University of Hildesheim, Institute of Computer Science, Universitätsplatz 1, 31141 Hildesheim, Germany
elscha@sse.uni-hildesheim.de

Adam Krafczyk
University of Hildesheim, Institute of Computer Science, Universitätsplatz 1, 31141 Hildesheim, Germany
krafczy@uni-hildesheim.de

Klaus Schmid
University of Hildesheim, Institute of Computer Science, Universitätsplatz 1, 31141 Hildesheim, Germany
schmid@sse.uni-hildesheim.de

Abstract

The Linux kernel is often used as a real world case study to demonstrate novel Software Product Line Engineering research methods. An important point in this is often the analysis of the Kconfig semantics. However, we detected that the semantics of Kconfig is rather unclear and has many special cases, which are not documented in its short specification. We performed a systematic analysis to uncover the correct behaviour of Kconfig and present the results, which are necessary for applying semantically correct analyses. Further, we analyse existing analysis tools of the research community whether they are aware of the correct semantics of Kconfig. These analyses can be used for improving existing analysis tools as well as decision support for selecting an appropriate tool for a specific analysis. Our main goal is to contribute to a better understanding of Kconfig in the research community to improve the validity of evaluations based on Linux.

Categories and Subject Descriptors D.2.9 [*Software Engineering*]: Management—Software configuration management; D.2.13 [*Software Engineering*]: Reusable Software—Reuse models

General Terms Verification

Keywords Software Product Lines, Linux Kernel Analysis, Kconfig, Undertaker, Kconfig Reader, Linux Variability Analysis Tools, LVAT

1. Introduction

The Linux kernel is often used as a real world case study to demonstrate novel Software Product Line Engineering research methods, as it is a high configurable open source software. The publicly available repository facilitates the analysis of the variability model, the instantiation process, the instantiable artefacts, and the evolution of all of them. An important point in this is often the analysis of the Kconfig semantics, the variability management system of Linux. However, we detected that the semantics of Kconfig is rather unclear and has many special cases, which are not documented in its short specification. This leads to an incomplete understanding of the Kconfig semantics in the scientific research community, which may result in incorrect analysis of Kconfig variability models and hence be a threat to validity for existing research.

In this paper, we uncover hidden aspects of the Kconfig semantics to improve the understanding of the modelled variability of the Linux kernel. Further, we analyse how far these corner cases are considered by existing research tools to support their improvement. We included the translators of Undertaker [3], Kconfig Reader [11], and the Linux Variability Analysis Tools (LVAT) [13] in our analysis, as these are the existing tools for Kconfig translation into formal representations. With our research we predominantly aim at improving the understanding of the Kconfig semantics in the research community. In addition, our results can be used for improving existing analysis tools and identifying potential problems for analysis that have been done in the past. The results serve also as a basis for selecting an incomplete analysis tool, which may be sufficient for analysing a specific aspect. In summary, we contribute to a better understanding of Kconfig and related research tools for Kconfig translation.

In Section 2, we show related work dealing with the analysis of Kconfig, where we focus on tools used to analyse Kconfig for scientific research. Section 3 introduces Kconfig, the textual variability language of Linux. We present the concepts of Kconfig together with an example how they can be modelled. In Section 3, we present our systematic analysis to find undocumented corner cases of the Kconfig semantics. Afterwards, we show how well these corner cases are supported by analysis tools used for scientific work. In Section 6, we conclude and show future work.

2. Related Work

Kconfig was developed as a textual variability language to manage the variability of Linux [9]. While other textual variability languages have already been extensively compared [6], we currently only know of a comparison of eCos and Kconfig [2].

Our analysis is not the first to address the question of Kconfig semantics. In [16] She and Berger presented a formal semantics of Kconfig, while in [19] Zengler and Küchlin developed algorithms to translate the variability information into propositional logic. We go beyond this prior research by identifying details of the semantics that were not captured in their work, e.g. semantics of constraints used inside choices.

In addition to these descriptions of semantics also a number of tools have been developed that aim at translating the Kconfig model into some form of logic. However, most tools are missing a precise description of semantics they imply for Kconfig. Exceptions are the tools developed by She and Berger, and Zengler and Küchlin. We included the tool from She and Berger (LVAT), while the described tool by Zengler and Küchlin was not publicly available. As a result, we analyse the following tools:

GPCE'15, October 26 – 27, 2015, Pittsburgh, PA, USA.
ACM 978-1-4503-3687-1/15/10. . . $15.00.
http://dx.doi.org/10.1145/2814204.2814222

Tool		Capabilities	Output	String / Numbers	Status	Repositories
Undertaker	dumpconf	Translates Kconfig into RSF	RSF	Partially supported	Maintained	`https://vamos.informatik.uni-erlangen.de/trac/undertaker`
	rsf2Model	Translates RSF into Model	Model (boolean formulas)	Not supported		
	satyr	Translates Kconfig into DIMACS	DIMACS	Partially supported		
Kconfig Reader	Current version with reimplemented dumpconf	Translation of Kconfig	XML, Model (boolean formulas), DIMACS, Header files	Supported in all output formats	Maintained	`https://github.com/ckaestne/kconfigreader`
	Old version with modified dumpconf		RSF, Model (boolean formulas), DIMACS, Header files		dumpconf discontinued	Kconfig Reader repository + `https://github.com/ckaestne/undertaker`
LVAT		Translation of Kconfig, statistical analysis	Bool (boolean formulas), DIMACS	Not supported	Discontinued	`https://code.google.com/p/linux-variability-analysis-tools/`, `https://bitbucket.org/tberger/vm2bool`

Table 1. Existing translators and their capabilities, translated output files, and whether they are supporting String and Numerical elements, which is discussed in Section 5.1.

Undertaker [3, 4, 17] is a tool set developed to check the structure of preprocessor directives of the Linux kernel against its configuration model to find code blocks, which are permanently (de-) selected in all configurations. This analysis is done in two steps. First, all Kconfig-files are translated into logical models. In the second step, the preprocessor directives are also translated into logical models and passed directly together with the logical translations of Kconfig to a (SAT) solver to detect problems. Here, we are only interested in the first part. Undertaker embeds two alternative tool chains for translating Kconfig into boolean formulas:

dumpconf + rsf2model

dumpconf is a modified version of menuconfig, which part of Linux. This tool translates the Kconfig-files first into an intermediate format, named RSF-files. The RSF-files mainly contain the same logical information as the Kconfig-files, but dumpconf already adds some extra information for downstream analysis tools, e.g. additional constraints between nesting and nested elements.

rsf2model is used to create boolean formulas based on the RSF-files. These files are saved as Model-files (*.model).

satyr

satyr is an alternative to dumpconf and rsf2model, which was introduced in Undertaker 1.5 [3]. This tool translates the Kconfig-files directly into DIMACS format.

FMDiff [5] was designed for analysing the evolution of Kconfig-models. It uses RSF-files extracted by dumpconf to detect changes that occurred in the variability model of the Linux kernel. We did not include this tool in our analysis, because FMDiff reuses the dumpconf translator.

Kconfig Reader [11] was initially developed for the TypeChef tool [12, 18] to convert Kconfig-files into Boolean formulas for reasoning. Until 12.09.2014, Kconfig Reader used a modified version of Undertaker's dumpconf component for creating RSF-files. These files are used to create boolean formulas, which are stored in Model-files (*.model). However, Kconfig Reader's Model-files are different to Undertaker's Model-files. Kconfig Reader optionally offers the possibility to save the boolean formulas in DIMACS format and offers also the possibility to save value assignments as Header-files (*.h). Unlike Undertaker, Kconfig Reader supports constraints based on Strings and Numbers in all output formats. Recent versions of Kconfig Reader use a reimplemented version of dumpconf that produces XML-files instead of RSF-files as intermediate format.

Linux Variability Analysis Tools (LVAT) [2, 14] is a tool set for performing miscellaneous analyses on top of the variability model of Linux. It offers mechanisms for printing statistical information about a given Kconfig-model like, hierarchies, permanent visible variables, visibility conditions, or different kinds of dependency analyses (OR, XOR, Mutex groups). LVAT also includes a viewer considering constraint hierarchies and is able to translate a Kconfig-model into boolean formulas and also into DIMACS-format.

3. The Kconfig Language

Here, we briefly describe Kconfig [7, 9]. Kconfig is a textual language, which was developed to manage the variability of the Linux kernel. Although it never became a standalone project, it was also used in several other projects [1]. We describe only the most important variability management concepts of Kconfig based on its language specification [9]. We do not include concepts which do not affect Kconfig's variability management logic, like `help`, `comments`, or `mainmenu`. Listing 1 serves as an example to show how the concepts can be modelled (not all possibilities are shown in the example). While the semantics of each element in isolation is already specified [9], we analyse in Section 4 how these elements do interact if they are combined.

3.1 Config Options

Config options (and also menuconfigs) are the most used elements of the language and can be seen as variables. These config options can be of type `tristate`, `bool`, `string`, `hex`, or `int`. All but the first are known from typical programming languages and have the same semantics. Tristate config options encode the following alternatives: `n` (the related feature will not be part of the resulting system), `y` (the related feature will be a permanent part of the resulting system), and `m` (the related feature will be compiled as a module, which means that it can be flexibly loaded or unloaded at runtime). Lines 12–13 of Listing 1 specify a tristate config option.

3.2 Attributes

Any config option can be augmented with (conditional) attributes:

- A prompt displays a name to the user. A config option without a prompt is invisible and cannot be altered directly (only via constraints). Prompts can be defined together with the type definition of the config option (e.g. line 3) or by using the `prompt` keyword (cf. line 9).

```
1  menu "A Menu"
2      config MODULES
3          bool "Enable third state"
4          default y
5          option modules
6
7      config STR_VAR
8          string
9          prompt "A string variable"
10         depends on MODULES
11
12     config TRI_VAR
13         tristate "A tristate variable"
14 endmenu
15
16 choice
17     bool "Like an enumeration"
18
19     config VAL1
20         bool "Value 1"
21
22     config VAL2
23         bool "Value 2"
24         select TRI_VAR
25 endchoice
26
27 if STR_VAR="Hello World"
28     # Include Kconfig model for drivers
29     source drivers/Kconfig
30 endif
```

Listing 1. Example of a Kconfig file (indentation not necessary, only used for better readability).

- The default attribute specifies default values of a config option. These defaults can be changed by the user if the config option is visible. Furthermore, a config option can contain any number of (conditional) defaults (cf. line 4).
- A range specifies an upper and a lower bound for config options of type int or hex.
- The current version of Kconfig supports that at most one arbitrary config option can be attributed with `option modules` for controlling the Tristate semantics (cf. line 5). If the config option is set to `n` (disabled), all Tristate config options are treated as Boolean config options and do not accept `m` as a value.
  Earlier versions of Kconfig did not specify this attribute. Instead, a config option must be named as `MODULES` to control the Tristate semantics (cf. line 2).
- Kconfig specifies two different kinds of constraints to restrict the possible values of a config option:
  - `depends on` is used to describe whether a config option can be configured or is disabled. Contrary to the other attributes, this attribute can not be made conditional (cf. line 10).
    This kind of constraints can also be used to restrict Tristate config options to be selected only as `n` or `m`. This can be done with a `depends on m` constraint.
  - `select` is used to specify a lower bound. The current value of the surrounding config option is used as lower bound for the selected config option. If a config option is selected multiple times, it is set to the largest selection. It is also possible to select a config option without fulfilling its `depends on` constraints. Furthermore, `select` constraints may be conditional. In such a case, the selected config option will only be changed if the selecting variable was changed and the condition of the `select` constraint was fulfilled (cf. line 24 of Listing 1 shows a `select` constraint without a condition).

3.3 Choices

A choice groups several config options and can only be of type `bool` or `tristate`. In case of a Boolean choice, exactly one config option must be selected. A Tristate choice allows the selection of multiple elements as modules. A choice can be attributed with most of the attributes from above. A choice can be `optional`, which allows to set the choice to `n` and none of the contained config options may be selected (cf. lines 16–25).

3.4 Hierarchies

Kconfig offers two possibilities to structure config options in hierarchies:

- Any number of config options (and other elements like choices) can be surrounded by the keywords `menu` and `endmenu` to place them in a sub menu. A sub menu can also contain other menus (cf. lines 1–14).
  The visibility of these menus can be restricted with the attribute `visible if`. This attribute is only applicable for menus and restrict the whole menu including all nested elements. However, these elements still exist and can be selected or modified by constraints or default values, but not by the user.
- Constraints and `if`-statements offer an alternative to model hierarchies. If the visibility of a config option *B* depends on another config option *A* and *B* is directly written below *A* in a Kconfig file, than *B* will also be displayed in a hierarchical structure below *A* inside configuration front-ends (e.g. based on Listing 1, `STR_VAR` would be intended below `MODULES`, as long as there is no other config option introduced between these two config options, which is not also dependent on `MODULES`).

3.5 File Inclusions

Kconfig allows to split a huge variability model into separate files. The `source` statement includes the contents of another Kconfig file into the current file (cf. line 29).

3.6 if-Statements

It is possible to surround the elements from above with `if` and `endif` to make their existence depending on a condition. The kernel developers use this for defining the same config option twice with different attributes or for including different Kconfig files depending on the target architecture (cf. lines 27–30).

3.7 Constraint Logic

Kconfig uses a tristate logic instead of a boolean logic for creating and evaluating expressions and supports the usage of 5 operators: and (`&&`), or (`||`), equality (`=`), unequals (`!=`), and the not operator (`!`). Parenthesis can be used to override the precedence of sub-expressions.

3.8 Constraint Precedence

Kconfig is able to handle contradictory dependencies by means of its own constraint precedence. As already mentioned above, `select` statements are able to select config options without fulfilling their `depends on` constraints or surrounding `if`-statements. Config options, which are not visible to the user but have a default value, can be treated as constant as long as they were not modified by a constraint. Thus, a transformation into a logical model must consider the following constraint precedence:

1. `select`
2. `if`, `depends on`
3. No `prompt` + `default`

contains ⟶ ↑		Config options				Choices		Hierarchies		Constraints		Attributes				If
		bool	tristate	string	numerical	bool	tristate	menu	via Constraints	depends on	select	prompt	default	range	visible if	
Config options	**bool**	×				×		×	✓	✓	⚠	✓	⚠	—	×	×
	tristate								✓		⚠		⚠	—		
	string								+		—		✓	—		
	numerical								+		—		ϟ	ϟ		
Choices	**bool**	✓	⚠	ϟ		+		×	⚠	⚠	Ⓢ	⚠	✓	Ⓢ	×	⚠
	tristate	⚠	✓													
Hierarchies	**menu**	✓				✓		✓	×	✓	×	✓	×	×	✓	✓
	via Constraints								✓		✓		✓	✓	×	+
Constraints	**depends on**	✓		—		—		—	✓	×		×				×
	select							×	+							✓
Attributes	**prompt**	×				×		×		×		⚠				✓
	default															
	range															
	visible if															
If		✓				✓		✓	+	×		×				✓

Table 2. Systematic analysis of undocumented functionalities.

4. Analysis of the Kconfig Semantics

In this section, we will provide our in–depth analysis of the Kconfig semantics. This analysis aims at elaborating missing details in the Kconfig specification [9] by identifying the configuration behaviour from the `menuconfig`–tool which operationalizes Kconfig in practice.

The first part describes how we analysed the Kconfig implementation especially for potentially problematic combinations. In Section 4.2, we present unexpected behaviour of Kconfig, which we discovered through our analysis. In Section 5, we contrast our analysis results to the output of some existing Kconfig translators.

4.1 Analysis Process

The main focus of our research was to perform a systematic analysis of Kconfig (specification and implementation) to uncover possible corner cases [7]. From the specification, we identified the variability management concepts of Kconfig discussed in Section 3. These concepts provide the basis for our analysis of the semantics[1]. The main approach we used was to analyse the pairwise interactions of the language concepts. This served as a basis for Table 2 (rows and columns). Starting from this table, we tested all possible interactions, independently of whether the combinations are specified by the language specification. Based on already discovered corner cases, we created a list of lead questions for a systematic detection of further corner cases instead of applying an ad–hoc search. For each combination, we tried to answer the following questions:

$\mathcal{Q}_1$ How can the elements be combined?

$\mathcal{Q}_2$ In case of a specified combination, is it possible to omit a nested element, e.g. because of a conditional element that is not fulfilled, for instance `select <VAR> if n`. And how does this impact the semantics?

$\mathcal{Q}_3$ Can multiple contradictory elements become active? For instance, the conditions of different default values of the same config option may become active.

$\mathcal{Q}_4$ Is it possible to combine elements of different types, like a default value of type String inside a Numerical config option?

$\mathcal{Q}_5$ Is it possible to use a specified combination in an illegal manner? For instance, specifying a range with a lower bound greater than the upper bound.

We created a minimal Kconfig model for each question. An example of such a model is shown in Listing 4. In order to understand the generally accepted semantics, we executed `menuconfig` from Linux v3.19 for each of these models and monitored its behaviour. Linux is also shipped with other configuration front–ends, but they are all built on top of the same logic implementation.

Inside the `menuconfig`–tool, we tried to configure these models and monitored the tool support, e.g. whether values are propagated. We also saved the configurations and analysed the output, whether user changes were accepted by Kconfig and how the results were saved, e.g. String values for a Numerical config option. We summarized our findings in Table 2 as follows:

— If the nested elements do not influence the model, i.e. omitting the nested element does not change the model. These combinations were also not described in the language specification.

× We were not able to model the desired combination, but the language specification is also not specifying this combination. Thus, we do *not* regard this as a semantics problem.

✓ The observed behaviour of Kconfig is clearly described in the language specification.

⚠ These combinations do not lead to inconsistent configurations, but their behaviour is not described clearly in the language specification. We also found examples for some of these combinations inside Kconfig models of the Linux kernel and expect that the observed behaviour was desired by the Linux developers.
We explain the discovered behaviour in Section 4.2, before we analyse whether existing analysis tools can handle these interactions in Section 5.

[1] We omitted "File Inclusions", as it is clearly specified that this should not have any effect on the semantics.

We found further corner cases, were we expect that these combinations are not desired by the Linux developers [7]. Due to space limitations, we are not able to describe them here. These combinations are marked as follows:

↯ These combinations can be used to create inconsistent models, e.g. a Numerical config option with a String default value. These combinations lead to conflicts, which are very obvious. We expect that such obvious modelling failures would be found fast and resolved quickly. For this reason, we do not expect that a translation tool should be able to handle these combinations, but it could detect them. We excluded this combinations from our analysis in Section 5.

+ These combinations may be represented in a special fashion inside the configuration front–ends. However, the semantics are still clear. These combinations may also be modelled with the basic concepts already described in Section 3.

Ⓢ This relates to cases where the language specification is obviously faulty, but `menuconfig` implements a reasonable, straightforward semantics. For example, the specification defines that "all kinds of attributes" can be used inside a choice, but some combinations are not possible. Ⓢ stands for an incorrect specification.

4.2 Observed Corner Cases

In this section, we describe unexpected behaviour of Kconfig, which we observed while answering $\mathcal{Q}_1$ to $\mathcal{Q}_5$, where we summarized similar cases in one bullet. Here, we list combinations, where answering the questions lead to at least one ⚠. This behaviour is not directly evident from the specification. In case, we found existing examples inside the Linux kernel, we added a notice behind the description in which kernel version we found it. A more detailed description is given in our report [7]. However, it stays open whether such constructs are used inside other Kconfig based projects or in other kernel versions.

***Config Options* × *`select`* (Found in v3.19):** A `select` constraint should take the value from the holding `bool`/`tristate` config option and should set this value as minimal value for the selected config option independent of existing `depends on` constraints.

$\mathcal{Q}_1$ This combination is well specified. ✓

$\mathcal{Q}_2$ The select statement is only optional, thus, it can be omitted, but this is not critical. ✓

$\mathcal{Q}_3$ Multiple `select` statements pointing to the same config options are a well specified concept of Kconfig. If a config option is selected multiple times, its lower bound is set to the maximum selection (`y` > `m` > `n`). Also one config option may select multiple other config options. ✓

$\mathcal{Q}_4$ Selecting a String or Numerical config option has no effect. Also String or Numerical config options selecting Boolean or Tristate config options has no effect. This is probably related to the fact, that String and Numerical values are treated as `n` and `select` statements are used to modify the lower bound of a config option. —

$\mathcal{Q}_5$ Config options nested inside a choice cannot be selected. This behaviour avoids that multiple config options of the same choice are selected. ⚠

***Config Options* × *`default`* (Found in v3.19):**

$\mathcal{Q}_1$ This combination is well specified. ✓

$\mathcal{Q}_2$ A default attribute is only optional. ✓

$\mathcal{Q}_3$ Same as `default` × `default` (see below). ⚠

$\mathcal{Q}_4$ Numerical config options containing a default value of type String lead to inconsistent models [7]. ↯

$\mathcal{Q}_5$ It is possible to specify `m` as default value for Boolean as well as for Tristate config options. For a Boolean config option `m` as default value is equivalent to `y`.
If the Tristate logic was not activated by the config option attributed with `option modules`, also Tristate config options with a default value `m` will be set to `y`. After the Tristate logic has been enabled, Tristate config options will be set to `m`. Boolean config options will still be set to `y`. ⚠

Choices* × *Config Options: Choices can be either of type `bool` or `tristate` and should contain only Boolean/Tristate config options.

$\mathcal{Q}_1$ This combination is well specified. ✓

$\mathcal{Q}_2$ Same as "Choices × Constraints". ⚠

$\mathcal{Q}_3$ Config options cannot be contradictory per se. This may only the case if constraints are used and this is covered by the other cases. ×

$\mathcal{Q}_4$ A Tristate choice may contain Boolean config options or a Boolean choice may contain Tristate config options. Nested Boolean config options will only be selectable if the choice is set to `y`. In this case, the selected config option will also be set to `y`, independently of whether the config option is of type Boolean or Tristate.
If the choice is set to `m`, only the Tristate config options will be selectable as modules and all Boolean config options become permanently invisible. ⚠

$\mathcal{Q}_5$ A choice may contain String or Numerical config options, but this leads to inconsistent models [7]. ↯

Choices* × *Hierarchies (Found in v3.19): As already mentioned above, constraints can be used to model a hierarchy. This is also possible inside a choice, i.e. a nested config option may be dependent on another nested config option. For instance, in Listing 1 a new config option may be introduced between lines 24 and 25, which is also dependent on `VAL2`. This combination is not specified in the Kconfig language specification.

$\mathcal{Q}_1$ Even if the choice is of type Boolean or set to `y`, Kconfig allows the selection of all dependent config options contrary to the specification of Boolean choices. This is used for a more detailed configuration of selected config options of a choice. ⚠

$\mathcal{Q}_2$ A choice without a constraint hierarchy is a well defined concept of Kconfig. ✓

$\mathcal{Q}_3$ Same as "Choices × `if`". ⚠

$\mathcal{Q}_4$ Constraint hierarchies are not typed, thus, we were not able to model a type conflict. ×

$\mathcal{Q}_5$ This led to several critical observations. We named this "Choices × (broken) Hierarchies" to make it referenceable and describe it in detail below. ⚠

Choices* × *(broken) Hierarchies: It is vitally important that all dependent config options are written directly below the containing config option to model a hierarchy via constraints. For instance, if in Listing 1 a new config option is introduced between `MODULES` and `STR_VAR`, which is not also dependent on `MODULES`, the semantics of the constraints slightly changes.

Outside a choice, this does only affect the representation of the dependent config options inside the configuration front–ends, as the dependent config options will not be indented below the containing config option. The constraints must still be fulfilled.

Inside a choice, a nested hierarchy will be removed, if the ordering is shuffled and a dependent config option is not directly written below the related config option. In this case, `menuconfig` will display a warning that a "recursive dependency" exists inside the choice and removes all dependent config options.

***Choices* × *`if`*:**

$\mathcal{Q}_1$ This combination is well specified, as `if`–statements can surround all elements except attributes. ✓

$\mathcal{Q}_2$ The usage of `if`–statements is optional. ✓

$\mathcal{Q}_3$ A hierarchy can be broken if `if`–statements and `depends on` constraints are combined to model dependencies between config

options. Even if the dependency is modelled correctly with `depends on` constraints and the dependent config option is written directly below the containing config option, the hierarchy becomes broken if the containing variable is not used inside the `if`–statement, e.g. only inside the `depends on` constraint. In case of a choice, `menuconfig` removes the whole hierarchy and displays a warning that a "recursive dependency" was detected. ⚠

$\mathcal{Q}_4$ `if`–statements do not have a type. ×

$\mathcal{Q}_5$ An `if`–statement can surround all nested config options. This would have the same effect as described in "Choices × Constraints". ⚠

Choices × Constraints (Found in v2.6.33.3): Choices as well as their included config options can be made dependent on other config options, e.g. via `depends on` constraints.

$\mathcal{Q}_1$ This combination is well specified. ✓

$\mathcal{Q}_2$ `depends on` constraints are only optional. ✓

$\mathcal{Q}_{3,5}$ The choice as well as the included config options need not necessarily depend on the same expression. This allows the creation of empty Boolean choices, where none of the `depends on` constraints of the nested config options is fulfilled. Although the Kconfig language specification specifies the selection of exactly one variable, `menuconfig` is able to handle such situations and will not force the selection of a non–existing config option. ⚠

$\mathcal{Q}_4$ It is not possible to specify a type for a `depends on` constraint. ×

Choices × `prompt`:

$\mathcal{Q}_1$ This combination is well specified. ✓

$\mathcal{Q}_2$ In a choice the `prompt` attribute is optional. If it is present, it can also be made conditional using an if clause. A choice which is not visible to the user, will also not be part of the configuration even if the choice should be present (not `optional` and all `depends on` constraints are fulfilled) and a `default` value was specified. ⚠

$\mathcal{Q}_{3,5}$ A prompt can only be used to make an element visible to the user. Thus, we were not able to use this in a contradictory manner. ×

$\mathcal{Q}_4$ It is not possible to specify a type for a `prompt`. ×

`default` × `default`:

$\mathcal{Q}_{1,3,5}$ It is not possible to nest a default inside another default, because it is not possible to specify a scope for this attribute. However, a related question is what happens if two conditional default values of the same config option became active? In cases were multiple interacting attributes of the same config option or choice become active, only the first one is used. ⚠

$\mathcal{Q}_2$ A default attribute is only optional. ✓

$\mathcal{Q}_4$ Same as $\mathcal{Q}_4$ of "Config Options × `default`". ↯

5. Analysis of Kconfig Analysis Tools

In the previous section, we presented the modelling concepts of Kconfig and discussed undocumented behaviour, which we identified. Based on this, we analysed existing research prototypes, which were developed to translate Kconfig into other logical languages. These translations have been used for analysing the variability of Linux or other Kconfig–based implementations. Thus, our analysis serves as a meta–analysis of existing research tools for Kconfig translation.

We started with an analysis of concepts, which differ fundamentally to propositional logic as this is the final target of all analysed tools. We tested whether the tools support:

- The `option modules` attribute.
- The constraint precedence, i.e. whether config options can be selected even if their dependencies are not fulfilled.

```
1  menu "A Menu" {
2   config MODULES boolean {
3    prompt "Enable third state" if []
4    default [y] if []
5    config STR_VAR string {
6     prompt "A string variable" if [MODULES]
7     depends on [MODULES]
8     inherited [MODULES]
9    }
10  }
11  config TRI_VAR tristate {
12   prompt "A tristate variable" if []
13  }
14 }
```

Listing 2. Intermediate format of LVAT (translated only the menu from Listing 1).

- Constraints including undefined config options, e.g. because the related config option is declared only inside a sub–model for another architecture.

Afterwards, we tested how the tools translate the interactions from Section 4.2. For doing so, we created a Kconfig model for each case and compared the configuration spaces of the translated models with the configuration space given by menuconfig, which we regard as an operationalisation of the Kconfig semantics [7]. We included the translation capabilities of the following tools in our analysis:

- Undertaker: Translation into RSF, Model, and DIMACS files. We used Undertaker in version 1.6.1, which was the latest version, available at time of the analysis.
- Kconfig Reader: Translation into RSF, Model, and DIMACS files. We tested two versions of Kconfig Reader. The commit from 12.09.2014 was the latest version using the modified version of dumpconf, which also produces RSF files as intermediate format. The second version was from 03.06.2015, which was the latest available version. This version uses a reimplementation of dumpconf and produces XML files as intermediate format. We did not include the generated XML files into our analysis, as they are only produced by this tool as an intermediate result and we do not know any other tools which directly use them as input format.
- LVAT: Translation into DIMACS files. LVAT is not able to translate Kconfig models directly into DIMACS. First the models must be translated into an intermediate format, which should simplify parsing. The developers offer two tools for generating the intermediate format [10]:

  exconf translates Kconfig into a text format. This extractor is marked as compatible with Kconfig models prior to Linux v3.0. An example is given in Listing 2. We translated only the menu (lines 1 – 14) of the Kconfig model from Listing 1 to provide a small and meaningful example.

  protoconf translates Kconfig into a binary format and is compatible with more recent versions of Linux. We do not show an example here, as the binary format is not readable.

  In our analysis, we used protoconf for translating Kconfig into the needed intermediate format. Further, we used vm2bool for generating the DIMACS–files. This is an extension of LVAT developed by the same developers to simplify the generation of DIMACS files.

In Section 5.1, we introduce the logical formats generated by the translators. Due to space limitations, we are not able to present all Kconfig models of the analysis from Section 4 together with the generated output of these translators. This is included in the related technical report [7]. Instead, we present a small and meaningful

```
1  Item          MODULES     boolean
2  Prompt        MODULES     "y"
3  HasPrompts    MODULES     1
4  Default       MODULES     "'y'"       "y"
5  Item          STR_VAR     string
6  Depends       STR_VAR     "MODULES"
7  Prompt        STR_VAR     "MODULES"
8  HasPrompts    STR_VAR     1
9  Item          TRI_VAR     tristate
10 Prompt        TRI_VAR     "y"
11 HasPrompts    TRI_VAR     1
```

Listing 3. Example of a RSF-file (generated with Kconfig Reader).

example in Section 5.2. Finally, we show the complete results of the analysis in Section 5.3.

5.1 Logical Models

In this section, we introduce the logical models, generated by the analysed tools. This also provides first insights into their capabilities. We use `<NAME>` to explain how a Kconfig option named with `NAME` is translated into the described output formats.

5.1.1 RSF

RSF-files serve as an intermediate format to simplify further processing. RSF-files (*.rsf) are Tab Separated Value files (TSV) with a fixed set of elements to describe the information of the Kconfig files in a more structured way. The RSF-files should contain the same logical information as the Kconfig files, but dumpconf already adds some extra information for downstream analysis tools, e.g. `depends on` constraints between nested config options and the holding choices. File inclusions are already resolved, the resulting RSF-files contain information about declared config options, attributes (datatypes, how many conditional prompts, default values, select and depends on constraints), choices, and nested config options. Even expressions based on Strings and Numbers are translated. But String values are not surrounded by quotes, thus, it is not possible to clearly determine start and ending of String values inside complex constraints.

dumpconf of Undertaker reorders the items, which makes it impossible to differentiate between "Choices × (broken) Hierarchies" and "Constraint Hierarchies". Kconfig Reader keeps the ordering and adds further information, e.g. ranges of numerical config options and conditions of prompts. Both tools do not consider the `option modules` attribute. Kconfig Reader surrounds String values in single quotes.

An example is given in Listing 3, which was produced by Kconfig Reader. For brevity, we translated only the menu (lines 1–14) of the Kconfig model from Listing 1.

5.1.2 Models Generated by Rsf2Model

Model-files list all translated config options and choices as Boolean variables, each written in a single line. These variables may be followed by an expression, which must be fulfilled if the variable is set to true. Constraints based on Strings and Numbers are not translated. One line may contain `UNDERTAKER_SET ALWAYS_ON` for the permanent selection (`y`) of variables.

Tristate elements are translated into two elements (`CONFIG_<NAME>` and `CONFIG_<NAME>_MODULE`) with the following semantics (realised via constraints):

- `CONFIG_<NAME> = false` and `CONFIG_<NAME>_MODULE = false` denote that the related element inside the Kconfig model was permanently deselected (`n`).
- `CONFIG_<NAME> = true` and `CONFIG_<NAME>_MODULE = false` denote that the related element inside the Kconfig model was permanently selected (`y`).

```
1  config BOOL_VAR
2      bool "A Boolean Variable"
3      select BOOL_VAL2
4
5  choice
6      bool "A Choice"
7
8      config BOOL_VAL1
9          bool "1st Boolean Value"
10
11     config BOOL_VAL2
12         bool "2nd Boolean Value"
13 endchoice
```

Listing 4. Kconfig model for testing "Config Options × `select`".

- `CONFIG_<NAME> = false` and `CONFIG_<NAME>_MODULE = true` denote that the related element inside the Kconfig model was configured as a module (`m`).
- `CONFIG_<NAME> = true` and `CONFIG_<NAME>_MODULE = true` is not allowed.

5.1.3 Models Generated by Kconfig Reader

Even if Kconfig Reader uses the same file extension for saving translated boolean formulas, these Model-files have nothing in common with the Model-files generated by Undertaker. These Model-files contain only boolean formulas in propositional logic. `def(<NAME>)` encodes a selection, `!def(<NAME>)` encodes the deselection.

Tristate elements are translated into two elements (`<NAME>` and `<NAME>_MODULE`). These Model-files allow the same states with the same semantics as the Model-files generated by rsf2model.

String and Numerical elements are translated in several variables in form of `def(<NAME>=<value>)`. Kconfig Reader is creating such a construct for each value comparison found in Kconfig constraints and creates constraints to avoid multiple value selections for the same config option. Further variables are created as follows:

- `def(<NAME>=)` — Config option `<NAME>` was assigned to an emtpy String.
- `def(<NAME>=n)` — Config option `<NAME>` is not part of the configuration, i.e. a dependency was not fulfilled.
- `def(<NAME>=nonempty)` — A value was assigned, which was not covered by one of the `def(<NAME>=<value>)` variables, i.e. a value which was not used inside a constraint.

Kconfig Reader resolves dependencies for config options which are indirectly dependent on other config options. As a consequence, a dependency graph would be flattened, but the constraints become rather complex as more variables are involved in constraints.

5.1.4 DIMACS

DIMACS [15] is a file format to encode boolean formulas, written in conjunctive normal form (CNF), in a very compact manner, which can be handled by most SAT solvers. DIMACS files only consist of comment lines, a preamble and disjunction terms. Comment lines start with a `c` and must be written at the beginning of the file. The preamble specifies how many Boolean variables and how many disjunction constraints are used inside the DIMACS file (`p cnf <number of variables> <number of constraints>`). Variables are written as positive numbers. A negative number express the negation of the related variable. This format does not support Tristate logic or offers a direct mapping between the variables of the DIMACS file (numbers) and the variable names of the related Kconfig model. Hence, the tools use different techniques to encode Tristate, String and Numerical variables. All tools use comments to provide a mapping.

```
1  CONFIG_BOOL_VAL1 "CONFIG_CHOICE_1"
2  CONFIG_BOOL_VAL2 "CONFIG_CHOICE_1"
3  CONFIG_BOOL_VAR "CONFIG_BOOL_VAL2"
4  CONFIG_CHOICE_1 "((CONFIG_BOOL_VAL1 &&
       !CONFIG_BOOL_VAL2) || (!CONFIG_BOOL_VAL1
       && CONFIG_BOOL_VAL2))"
5  UNDERTAKER_SET ALWAYS_ON "CONFIG_CHOICE_1"
6  UNDERTAKER_SET SCHEMA_VERSION 1.1
7  CONFIG_X86 ""
8  CONFIG_n
9  CONFIG_y
10 CONFIG_m
```

Listing 5. rsf2model's Model translation of Listing 4 ("Config Options × `select`"): error in line 3.

Undertaker's **satyr** lists non-Boolean-/Tristate-config options inside the mapping. Constraints using such non-Boolean-/Tristate-elements are translated into separate variables in the form of `CONFIG___FREE__(NE | EQ)<Number>`. These variables are used correctly to translate constraints, but satyr does not model a dependency between this assignment variable and the related String-/Numerical-element. Tristate-elements are translated in the same way as in Model-files (cf. Section 5.1.2).

Kconfig Reader translates the boolean formulas of the Model-files into conjunctive normal form and stores them in DIMACS-files. These files provide the same mapping as the Model-files, except for the surrounding `def(...)`s (cf. Section 5.1.3). An example is given in Listing 7.

LVAT translates the extracts directly into DIMACS format. All kind of config options are translated as follows:

- `<NAME>` and `<NAME>_m` both being false means that the element is permanently deselected (`n`).
- `<NAME>` and `<NAME>_m` both being true means that the element is permanently selected (`y`).
- `<NAME>` being true and `<NAME>_m` being false means that the element is selected as a module (`m`). This state will only be allowed for Tristate config options and only if the Tristate semantics was enabled.
- `<NAME>` being false and `<NAME>_m` being true is an illegal state.

Value comparisons of Strings or Numbers are not translated. Further, LVAT creates constraints for nested config options of a choice, but does not create variables for the choice itself.

5.2 Examples

In this section, we show how "Config Options × `select`" is handled by the analysed tools to present some representative examples of wrong translations. In our analysis, only satyr was able to handle this corner case correctly. A detailed analysis of how all corner cases are handled by the different tools is given in our report [7].

```
1  #item BOOL_VAL1
2  #item BOOL_VAL2
3  (!def(BOOL_VAR)|def(BOOL_VAL2))
4  (!def(BOOL_VAR)|def(BOOL_VAL2))
5  #item BOOL_VAR
6  #item CHOICE_1
7  def(CHOICE_1)
8  #choice CHOICE_1
9  (def(BOOL_VAL2)|def(BOOL_VAL1)|!def(CHOICE_1))
10 (!def(BOOL_VAL2)|def(CHOICE_1))
11 (!def(BOOL_VAL1)|def(CHOICE_1))
12 (!def(BOOL_VAL2)|!def(BOOL_VAL1))
```

Listing 6. Kconfig Reader's Model translation of Figure 4 ("Config Options × `select`"): error in lines 3 and 4.

Listing 4 presents our test case for testing the translation of a `select` constraint pointing to a config option nested inside a choice. The `select` statement of line 3 is ignored by Kconfig, probably to prevent multiple selections of nested config options inside the same choice. In fact, this example is not only an artificial case. We also discovered a similar construct inside the Kconfig model of a recent version of the Linux kernel.

Listing 5 shows how rsf2model translates the example from Listing 4 into boolean formulas. Line 3 models an implication that if `BOOL_VAR` was selected, `BOOL_VAL2` has also to be selected. Thus, rsf2model translates the `select` statement of line 3 into boolean formulas, while Kconfig ignores this statement. Such translations could lead to an unsatisfiable model, if multiple `select` statements are pointing to different config options nested inside the same choice. While this translation is not correct, satyr translates this example correctly into DIMACS format. Thus, the correctness of the translation of the containing tool suite "Undertaker" differs depending on which sub–component was used.

Listings 6 and 7 show how Kconfig Reader translates the same model into its formats. Both translations model an implication between `BOOL_VAR` and `BOOL_VAL2`, which is not correct. Even if both results have the same structure in this case, this is in general not the case, as Model–files need not to be in conjunctive normal form. Further, Kconfig Reader adds a variable for controlling the Tristate semantics into the DIMACS–file and assigns it permanently to false (cf. lines 5 and 14).

5.3 Results

Table 3 summarizes the findings of our tool analysis. We marked some results inside the intermediate RSF–files as correct but problematic (Ⓟ). While dumpconf already simplifies the output for downstream analysis, it does not handle these corner cases. Thus, the resulting RSF-files are correct as they contain the same information as the original Kconfig models, but downstream analysis tools must be aware of the Kconfig behaviour as described in Section 4.2, otherwise the final result of the translation will be incorrect.

None of the analysed tools was able to handle all situations. There exist also some cases which were not handled by any tool correctly. These are the `option modules` attribute, which is used to control the Tristate semantics instead of the old mechanism and choices without a prompt. We did not find a choice without a prompt inside Linux v3.19. All tools are able to handle constraint precedence correctly (`select` over `depends on` / `if`), which is a fundamental concept of Kconfig. Further, they are all able to handle undefined config options inside of constraints.

Undertaker's satyr and the recent version of Kconfig Reader produce much more reliable results than the other tools. Both tools cannot handle the `option modules`, but at least for translating the

```
1  c 1 BOOL_VAR
2  c 3 CHOICE_1
3  c 2 BOOL_VAL2
4  c 4 BOOL_VAL1
5  c 5 MODULES
6  p cnf 5 8
7  -1 2 0
8  -1 2 0
9  3 0
10 2 4 -3 0
11 -2 3 0
12 -4 3 0
13 -2 -4 0
14 -5 0
```

Listing 7. Kconfig Reader's DIMACS translation of Figure 4 ("Config Options × `select`"): error in lines 7 and 8.

Case	Undertaker			Kconfig Reader					LVAT
	satyr	dumpconf	rsf2model	modified dumpconf			new dumpconf		vm2bool
	DIMACS	RSF	Model	DIMACS	RSF	Model	DIMACS	Model	DIMACS
Handling Attribute `option modules`	✗	✗	✗	✗	✗	✗	✗	✗	✗
Constraint Precedence	✓	✓	✗	✓	✓	✓	✓	✓	✓
Missing Config Options	✓	✓	✓	✓	✓	✓	✓	✓	✓
Config Options × `select`	✓	Ⓟ	✗	✗	Ⓟ	✗	✗	✗	✗
Boolean Config Options × `default`	✓	Ⓟ	✗	✓	Ⓟ	✓	✓	✓	✗
Tristate Config Options × `default`	✗	✓	✓	✓	✓	✓	✓	✓	✗
Tristate Choices × Boolean Config Options	✗	✓	✓	—	✓	—	✓	✓	✗
Boolean Choices × Tristate Config Options	✗	✓	✓	✗	✓	✗	✗	✗	✗
Choices × Hierarchies	✓	✗	✗	✗	✓	✗	✓	✓	✓
Choices × (broken) Hierarchies	✓	✗	✓	✓	✓	✓	✓	✓	✓
Choices × `if`	✓	✗	✗	✗	✗	✗	✓	✓	✗
Choices × Constraints	✗	✓	✗	—	✓	—	—	—	✗
Choices × `prompt`	✗	✗	✗	✗	✓	✗	✓	✓	—
`default` × `default`	✓	✗	✗	✓	✓	✓	✓	✓	✓

Table 3. Summary of tool analysis (—= translation aborted; ✗ = incorrect translation; Ⓟ= correct, but problematic for downstream analysis; ✓ = correct translation). We split "Config Options × `default`" and "Choices × Config Options" into two cases to provide a more fine grained analysis.

Kconfig model of Linux it is OK, as the variable for controlling the Tristate semantics is also named as `MODULES`. Also both tools are not able to handle one of the corner cases which we also found in a recent version of the Linux kernel. Further, satyr does not translate constraints based on Strings and Numbers correctly.

RSF–files produced by Kconfig Reader from 12.09.2014 can be used for writing own analysis tools. This tool can handle most of the concepts of Kconfig as it relies on a patched version of dumpconf, which in turn relies on menuconfig. However, it does not cover the `option modules` attribute and can not distinguish between a "recursive dependency" and a structured choice, if an `if`–statement is modelled inside a choice. Both observations should not be very problematic, since recent versions of Linux are still controlled over a config option named as `MODULES` and "recursive dependencies" are displayed as warnings by menuconfig and also during the translation of Kconfig Reader.

LVAT does not support most of the concepts. But LVAT is also the oldest tool, which we tested. Further, it was used to analyse Linux versions prior to v3.0 [8]. We did not analyse whether these old versions also contain some of the corner cases observed in Section 4.2. Thus, it still remains unclear how critical our observations are in respect to former results of LVAT.

6. Summary and Future Work

In this paper, we contributed to a better understanding of the Kconfig semantics and analysed existing Kconfig translators. We started with a compact introduction of the documented concepts of Kconfig and complemented this with a systematic analysis of undocumented interactions.

In the second part, we analysed the capabilities of existing analysis tools, used in scientific research. We started with an introduction of generated output formats and the resulting capabilities. For brevity, we showed only examples for one corner case, but presented the results of the complete analysis in Table 3. These results can be used to select an appropriate tool for performing own analysis, but also for improvement of these tools.

A significant amount of studies has been done with the analysed tools. As our research points to some semantic problems of the tool implementations, we believe a cautious review of these studies should be done as future work. However, we expect the impact will be limited. For example LVAT was used to analyse the structure of Kconfig models, like breadth and depth of generated Feature Models or the number of XOR groups [2]. At least the analysis regarding the breadth and depth of the model should be correct, because LVAT is considering constraint hierarchies correctly.

In this paper, we focused on the analysis of tools. As future work, we envision the analysis of the impact on theoretical work, like the formal semantics of the Kconfig language [16]. Further, a more in–depth analysis of the impact on published results of Kconfig based analysis seems required based on our results.

Acknowledgments

This work is partially supported by the Evoline project, funded by the DFG (German Research Foundation) under the Priority Programme SPP 1593: Design For Future — Managed Software Evolution. Any opinions expressed herein are solely by the authors and not of the DFG.

References

[1] T. Berger, S. She, R. Lotufo, A. Wasowski, and K. Czarnecki. Variability Modeling in the Systems Software Domain. Technical Report GSDLAB-TR 2012-07-06, Generative Software Development Laboratory, University of Waterloo, 2012.

[2] T. Berger, S. She, R. Lotufo, A. Wasowski, and K. Czarnecki. A Study of Variability Models and Languages in the Systems Software Domain.

IEEE Transactions on Software Engineering, 39(12):1611–1640, Dec 2013.

[3] CADOS Undertaker. `http://vamos.informatik.uni-erlangen.de/trac/undertaker`. Last visited 15.05.2015.

[4] C. Dietrich, R. Tartler, W. Schröder-Preikschat, and D. Lohmann. A Robust Approach for Variability Extraction from the Linux Build System. In *Proceedings of the 16th International Software Product Line Conference (SPLC '12)*, volume 1, pages 21–30, 2012.

[5] N. Dintzner, A. Van Deursen, and M. Pinzger. Extracting Feature Model Changes from the Linux Kernel Using FMDiff. In *Proceedings of the 8th International Workshop on Variability Modelling of Software-Intensive Systems (VaMos'14)*, pages 22:1–22:8, 2014.

[6] H. Eichelberger and K. Schmid. Mapping the design-space of textual variability modeling languages: A refined analysis. *International Journal of Software Tools for Technology Transfer*, pages 1–26, 2014.

[7] S. El-Sharkawy, A. Krafczyk, N. Asad, and K. Schmid. Analysing the KConfig Semantic and Related Analysis Tools. Technical Report 1/2015, SSE 1/15/E, Institute of Computer Science, University of Hildesheim, 2015. Available at `https://sse.uni-hildesheim.de/kconfig-study`.

[8] Extracts Repository. `https://code.google.com/p/linux-variability-analysis-tools/source/browse/?repo=extracts`, 2010. Last visited 14.06.2015.

[9] Kconfig Language Specification. `https://www.kernel.org/doc/Documentation/kbuild/kconfig-language.txt`, 2015. Last visited 04.03.2015.

[10] KconfigExtracts. `https://code.google.com/p/linux-variability-analysis-tools/wiki/KconfigExtracts`, 2012. Last visited 13.06.2015.

[11] kconfigreader. `https://github.com/ckaestne/kconfigreader`, 2015. Last visited 15.05.2015.

[12] 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 *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '11)*, pages 805–824, 2011.

[13] Linux Variability Analysis Tools (LVAT). `http://gsd.uwaterloo.ca/node/313`. Last visited 15.06.2015.

[14] S. Nadi, T. Berger, C. Kästner, and K. Czarnecki. Mining Configuration Constraints: Static Analyses and Empirical Results. In *Proceedings of the 36th International Conference on Software Engineering (ICSE 2014)*, pages 140–151, 2014.

[15] Satisfiability Suggested Format. Satisfiability Suggested Format. `ftp://dimacs.rutgers.edu/pub/challenge/satisfiability/doc/satformat.dvi`, 1993. Last visited 04.03.2015.

[16] S. She and T. Berger. Formal Semantics of the Kconfig Language. *Technical note, University of Waterloo*, 2010.

[17] J. Sincero, R. Tartler, D. Lohmann, and W. Schröder-Preikschat. Efficient Extraction and Analysis of Preprocessor-Based Variability. In E. Visser, editor, *Proceedings of the 9th International Conference on Generative Programming and Component Engineering (GPCE '10)*, pages 33–42, 2010.

[18] TypeChef. `http://ckaestne.github.io/TypeChef/`, 2013. Last visited 12.06.2015.

[19] C. Zengler and W. Küchlin. Encoding the Linux kernel Configuration in Propositional Logic. In *Proceedings of the 19th European Conference on Artificial Intelligence (ECAI 2010) Workshop on Configuration*, volume 2010, pages 51–56, 2010.