



**CHALMERS**  
UNIVERSITY OF TECHNOLOGY



UNIVERSITY OF GOTHENBURG

---

# **A Case Study of Interactive Conflict-Resolution Support**

in Software Configuration

Master's thesis in Software Engineering

DANIEL JONSSON



MASTER'S THESIS 2016

# A Case Study of Interactive Conflict-Resolution Support in Software Configuration

DANIEL JONSSON



**CHALMERS**  
UNIVERSITY OF TECHNOLOGY



**UNIVERSITY OF  
GOTHENBURG**

Software Engineering division, Department of Computer Science and Engineering  
CHALMERS UNIVERSITY OF TECHNOLOGY AND UNIVERSITY OF GOTHENBURG  
Gothenburg, Sweden 2016

A Case Study of Interactive Conflict-Resolution Support in Software Configuration  
DANIEL JONSSON

© DANIEL JONSSON, 2016.

Supervisor: Thorsten Berger, Chalmers University of Technology and University of Gothenburg

Supervisor: Sarah Nadi, TU Darmstadt

Examiner: Miroslaw Staron, Chalmers University of Technology and University of Gothenburg

Master's Thesis 2016

Software Engineering division, Department of Computer Science and Engineering

Chalmers University of Technology and University of Gothenburg

SE-412 96 Gothenburg

Sweden

Telephone +46 31 772 1000

Typeset in L<sup>A</sup>T<sub>E</sub>X

Gothenburg, Sweden 2016

A Case Study of Interactive Conflict-Resolution Support in Software Configuration  
DANIEL JONSSON  
Software Engineering division, Department of Computer Science and Engineering  
Chalmers University of Technology and University of Gothenburg

## Abstract

The Linux kernel is one of the largest highly configurable systems, with more than 13,000 configuration options. Although a description is provided for many of the configuration options, configuring the kernel has been identified as troublesome by both users and developers. To assist the user in satisfying unmet dependencies when configuring a system, an algorithm for resolving configuration conflicts called RangeFix has been proposed by academia. In this case study, we explore how RangeFix can be integrated with the Linux kernel configurator xconfig. We develop a prototype based on xconfig, where an existing Scala implementation of RangeFix is integrated to generate fixes and help the user resolve unmet dependencies. The workflow for configuring the kernel, supported by this prototype, is evaluated with Linux users through a survey. We find that the prototype is promising and can be useful in certain scenarios. We also evaluate the existing Scala implementation of RangeFix in terms of correctness and performance. Although the correctness is found to be good, it is much slower than the users' expectation. How RangeFix can be implemented to comply with the Linux kernel community's requirements is also explored in this case study. The result is a partial C implementation of RangeFix, based on a SAT solver. This C implementation is also evaluated in terms of correctness and performance. Even if our C implementation is incomplete, we find that this is a feasible way to implement the algorithm. When finished, it might perform well enough to provide interactive conflict-resolution support in the Linux kernel configurators.

Keywords: software configuration, conflict-resolution, configurators, Kconfig, xconfig, Linux, RangeFix, SAT.



# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Problem identification and motivation . . . . .	2
1.2	Objectives . . . . .	3
1.3	Research questions . . . . .	4
1.4	Research design . . . . .	4
<b>2</b>	<b>Background</b>	<b>6</b>
2.1	Feature modeling . . . . .	6
2.2	Kconfig . . . . .	10
2.2.1	Internal Kconfig infrastructure . . . . .	12
2.3	Constraint solvers . . . . .	14
2.4	Overview of available Kconfig tools . . . . .	15
2.5	RangeFix . . . . .	16
2.5.1	RangeFix’s three stages . . . . .	17
2.5.2	Generating diagnoses . . . . .	18
2.5.3	Encode a Kconfig model as an SMT problem . . . . .	19
2.6	Related works . . . . .	21
<b>3</b>	<b>Methodology</b>	<b>23</b>
3.1	Problem identification and motivation . . . . .	23
3.2	Objectives of a solution . . . . .	23
3.3	Design and development . . . . .	24
3.4	Demonstration . . . . .	25
3.5	Evaluation . . . . .	25
3.6	Communication . . . . .	26
<b>4</b>	<b>Design and development</b>	<b>27</b>
4.1	Encode a Kconfig model as a SAT problem . . . . .	27
4.1.1	Configuration option encoding . . . . .	28
4.1.2	Constraints encoding . . . . .	29
4.2	Read and set the configuration . . . . .	31
4.3	Generate unsatisfiable cores with SAT . . . . .	32
4.3.1	How unsatisfiable cores are generated with SMT . . . . .	33
4.3.2	How to generate unsatisfiable cores with SAT . . . . .	34
4.4	Generate diagnoses . . . . .	36
4.5	Simplify diagnoses . . . . .	38

4.6	Integrate with xconfig . . . . .	40
<b>5</b>	<b>Demonstration</b>	<b>42</b>
5.1	Configurator with Scala backend . . . . .	42
5.2	Configurator with C backend . . . . .	45
<b>6</b>	<b>Evaluation</b>	<b>47</b>
6.1	User survey . . . . .	47
6.1.1	Survey design . . . . .	48
6.1.2	Survey results . . . . .	48
6.2	The Scala implementation . . . . .	52
6.2.1	Evaluation design . . . . .	52
6.2.2	Correctness results . . . . .	54
6.2.3	Performance results . . . . .	59
6.3	The C implementation . . . . .	61
6.3.1	Evaluation design . . . . .	61
6.3.2	Correctness results . . . . .	63
6.3.3	Performance results . . . . .	65
6.4	Observations and conclusions . . . . .	67
6.4.1	User survey . . . . .	67
6.4.2	The Scala implementation . . . . .	68
6.4.3	The C implementation . . . . .	69
6.5	Threats to validity . . . . .	70
<b>7</b>	<b>Towards a SAT-based implementation in C</b>	<b>72</b>
7.1	Challenge #1: Integrating with xconfig . . . . .	72
7.2	Challenge #2: SAT encoding . . . . .	73
7.2.1	Proper tristate expression translation . . . . .	73
7.2.2	Use operators in conjunction . . . . .	79
7.3	Challenge #3: Realize diagnoses . . . . .	80
7.3.1	Using the internal Kconfig infrastructure for computing the configuration . . . . .	81
7.3.2	Implicitly configured configuration options . . . . .	81
7.4	Challenge #4: Realize fix generation . . . . .	82
7.4.1	Problem formulation . . . . .	82
7.4.2	Attempted approach . . . . .	83
7.4.3	Alternative approaches . . . . .	85
<b>8</b>	<b>Conclusion</b>	<b>87</b>
	<b>Bibliography</b>	<b>89</b>
<b>A</b>	<b>Survey questions</b>	<b>I</b>



# 1

## Introduction

Through software configuration, the users of a piece of software are able to customize it to their needs. There are software systems with a wide range of users, exposing an extensive set of configuration options to support a variety of hardware and software needs. These options might have complex dependencies between one another. Having an effective interactive configuration process that aids the user in resolving violations of one or more of these dependencies is therefore important.

For instance, the Linux kernel is one of the largest highly configurable open-source software systems in existence today [1, 2]. A screenshot of its configuration tool xconfig can be seen in Figure 1.1. It has more than 13,000 configuration options [3], also called features. Other examples of highly configurable systems are BusyBox, CoreBoot, eCos, and uClibc [2]. Highly configurable systems appear in software industries such as automotive, avionics, and communications equipment.

xconfig is not the only configurator that the Linux kernel comes bundled with. The Linux kernel also includes other configurators called config, menuconfig and gconfig that expose configuration of the same underlying feature model [4]. While config is a very basic prompt-based interface, the other configurators offer a menu-based interface, such as the one seen in Figure 1.1, using different front-end toolkits [4].

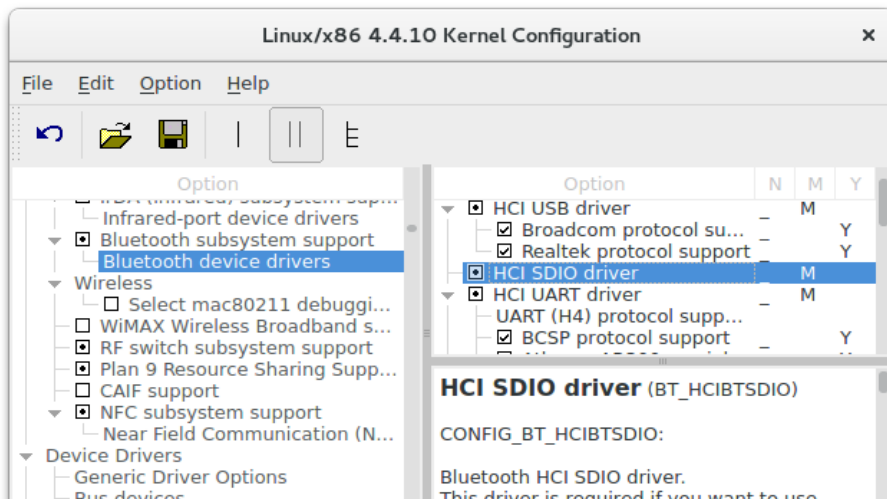


Figure 1.1: A screenshot of the Linux configuration tool xconfig.

## 1.1 Problem identification and motivation

Building feature models with Kconfig, which is the feature modeling language that the Linux kernel uses, occurs to a great extent. There are many people in the world who either interact with the language directly as developers, or indirectly using one of the many configurators. The Linux kernel has received contributions from more than 2,000 developers during a 15 months period [5]. Furthermore, between 2005 and 2015, nearly 1,000 companies contributed to the Linux kernel [5]. Taking into account the number of people using the Linux kernel, developing it, and the number of companies involved, it is the largest software development project in history [1]. Besides the Linux kernel, there are also several other open-source projects that use Kconfig for their feature model, including axTLS, BuildRoot, BusyBox, CoreBoot, EmbToolkit, Fiasco, Fraeetz, ToyBox, uClibc, uClinux-base and uClinux-dist [2]. Linux users configure the kernel for reasons such as personal use, server maintenance, system administration, development, embedded systems and virtual machines [6]. Since Kconfig is open-source, it is impossible to exactly quantify to what extent it is being used, but as noted here, there clearly are a lot of developers, administrators, projects and companies who interact with Kconfig in one or another way.

The task to configure the Linux kernel becomes more challenging as its feature model continues to grow. With each new release of the Linux kernel, the code base grows both in terms of source lines of code and configuration options [7, 8]. Between 2005 and 2015, the number of source lines of code in the kernel grew with 159%, or on average with 2.6% per release [8]. Today, there are more than 13,000 configuration options in the Linux kernel [3], making it one of the largest known feature models [2]. The number of Kconfig configuration options grows linearly with the number of lines of code, due to the development of the Linux kernel being predominantly feature-driven [9].

Hubaux et al. [6] have conducted a survey to identify challenges Linux and eCos users face when configuring their systems. 97 Linux users filled out the survey, roughly half of whom claimed to be experts with up to 20 years of experience with Linux. Half of the Linux user participants reported that they make between 20 and 50 changes to a default configuration, while some participants answered that they make more than 2,000 changes. 56% of the participants consider enabling/disabling a configuration option to be a problem in practice. 20% of the Linux user participants stated that they need at least a few dozen minutes on average to figure out how to activate an inactive option. To activate an inactive option, 46% of the participants said that they manually read and follow the constraints in the configuration option's help text. However, 19% of the Linux user participants complained that the help text documentation often is incomplete, hard to understand or incorrect. 26% participants said that they rely on their own expertise to activate an inactive option. In the paper by Hubaux et al., we find the following quote from one of the participants who provided a more detailed response to the survey:

"As far as consistency checking and helping determine inter-related dependencies on settings, I have long wished for a better kernel configuration tool [...], but it seems that the kernel guys learn their way around

the configurator by much exposure, and the rest of us have to just figure it out [...]"

Hubaux et al. [6] are not the only ones having identified the configuration process of the Linux kernel as lacking; there are also kernel developers who want to see improvements in this area. They have created a project called `kconfig-sat` with a wiki page [10], where information about the initiative is organized, and a mailing list [11], where discussions take place. Their plan is to integrate a boolean satisfiability solver, SAT solver for short, with the kernel configurators to assist the users in resolving dependencies. As a side note, they have also identified other areas in the kernel where a SAT solver could be beneficial, such as process scheduling and dependency resolution during boot up [12].

To illustrate the challenge in configuring the Linux kernel, let us look at a practical example. For instance, the user may want to enable the configuration option `RTLWIFI_DEBUG`, which provides debugging information to Realtek network adapters. But `RTLWIFI_DEBUG` depends on the configuration option `RTLWIFI`, which means that `RTLWIFI` has to be enabled before `RTLWIFI_DEBUG` can be enabled. However, `RTLWIFI` is invisible and does not appear in the configurator. To enable it, a Realtek network adapter must instead be enabled, for instance `RTL8192DE` or `RTL8821AE`. These latter two options are utilizing a reverse-dependency that enables `RTLWIFI` whenever they are enabled. To figure out why `RTLWIFI_DEBUG` is inactive and cannot be enabled, and to satisfy its constraints, is not straightforward with the current configurators. Resolving dependencies is often a time-consuming task where the user has to rely on his own expertise or manually follow the constraints described in the documentation [6].

To summarize, `Kconfig` is a feature modeling language that a large user base depends on in one way or another. The Linux kernel is interesting as a case study target due to its large feature model and popularity. Lastly, both users and developers of the Linux kernel have identified that a dependency-resolution tool would be beneficial.

## 1.2 Objectives

The objective of this Master's thesis is to conduct a case study where the feasibility of realizing an interactive conflict-resolution approach in one of the Linux kernel configurators is investigated. An approach would be attempted, and the effectiveness, scalability and practicability evaluated. By also collecting feedback from the Linux community, the usability of such a solution could be estimated. The results can be used to realize configuration processes for similar domains, or assess the feasibility of it. To achieve this objective, we address the following sub-goals:

- Generate a logical representation of the Linux kernel's feature model usable by reasoners.
- Find an approach to detect conflicts.
- Implement an approach to generate fixes, which will ultimately resolve the conflicts.

- Create a prototype that integrates the above into the Linux kernel configurator `xconfig`.

### 1.3 Research questions

The main research question to be addressed is:

- *How to realize scalable conflict-resolution for a configuration system as large as the Linux kernel configurator?* The feasibility of implementing a conflict-resolution algorithm from academia, which meets the requirements of the community, will be investigated.

The sub-question is:

- *How to encode the problem?* The constraints of the Linux kernel's feature model must be encoded in such a way that it is possible to do reasoning with them. The choice of constraint solver depends on what the Linux kernel community prefers.

### 1.4 Research design

Our goal is to realize interactive conflict-resolution support for one of the Linux kernel's configurators. This was initiated by a pre-study where available tools and options were investigated, with the goal of finding an appropriate algorithm for resolving unmet dependency. It was followed by the construction of a prototype where the usefulness of interactive conflict-resolution could be demonstrated. The feasibility of implementing the algorithm in C with a boolean satisfiability solver, which has a stronger community support, was also investigated. The last part of the thesis project was to evaluate the accuracy of the fixes, the scalability of the algorithm, and the usability of the prototype's user interface.

The research strategy for this project was a case study. In an exploratory manner, the objective to implement and evaluate interactive conflict-resolution in a highly configurable software has been researched. The context was software configuration, while the case was narrowed down to the Linux kernel, and the unit of analysis was this very project's produced artifacts. A frame of reference and data collection were achieved through a survey with kernel developers, where they provided their insights and feedback on the implementation. The case was intentionally selected within the context with the expectation of it to be revelatory due to its size and complexity.

The research method for this project was design science research, which can be divided into six activities [13]. The methodology is explained in further detail in Chapter 3, but a short overview is given here together with an outline of this thesis. The first activity of design science research is *problem identification and motivation*, where the specific research problem is defined and the value of a solution justified. This is elaborated in detail in Section 1.1. The second activity is *objectives of a solution*, where objectives of the project are inferred from the problem definition, which is done in detail in Chapter 3. The third activity is *design and development*,

where artifacts are produced with the aim of addressing the research question. The thought process behind the development and the produced artifacts are presented in Chapter 4. The fourth activity is *demonstration*, where the efficacy of the artifacts is showcased. This is done in Chapter 5, where the implementations are shown, and the workflow they support explained. The fifth activity is *evaluation*, where the artifacts are observed and measured to determine how well they solve the identified problem. This was done through correctness and performance analysis, as well as through a survey with kernel users and developers, and the results are presented in Chapter 6. The sixth activity is *communication*, where the problem and the artifacts are communicated, which is done through this report. The results have also been shared with the Linux developers. Furthermore, observed challenges that this thesis' C-based artifact has not yet been able to solve or implement are elaborated in Chapter 7. This provides information for the design and development activity for subsequent projects aimed at improving the effectiveness of that artifact.

# 2

## Background

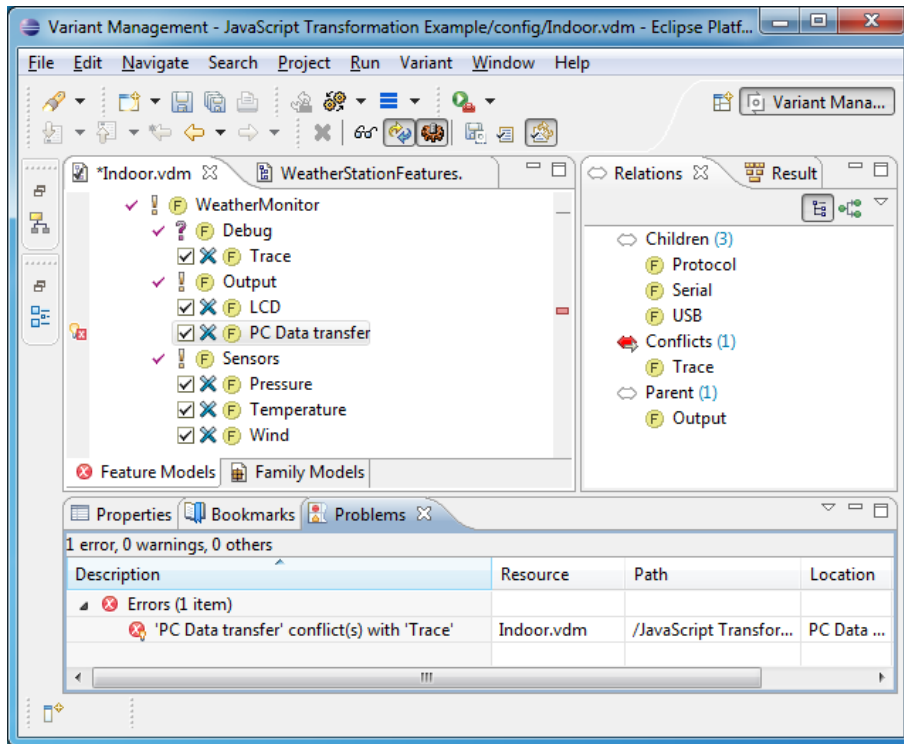
In this chapter, topics central for this thesis are explained. The first section shows how feature modeling looks like in some software systems. Next, an overview of the Kconfig feature modeling language is given. This is followed by the presentation of two types of constraint solvers. Next, an overview of the available Kconfig tools that were investigated for this thesis are presented. One such tool, RangeFix, is explained in further detail in its own section. Lastly, some related works are presented.

### 2.1 Feature modeling

A feature model is a graphical representation of commonalities and differences in a product line [14]. The feature model consists of a hierarchy of features [14], where a feature is an increment in product functionality [15]. Features are used to capture functionalities and distinguish products in a product line [15]. Furthermore, a feature model does also define constraints over the model to limit the number of valid combinations [14]. Each such valid feature combination is called a product, or an instance of the feature model [14].

Examples of feature modeling software are `pure::variants` from `pure-systems`, `Gears` from `BigLever`, and the `eCos Configuration Tool`. `pure::variants` is a commercial tool where the user can build a feature model consisting of features and restrictions between arbitrary features [16]. These restrictions are expressed with logical rules [16]. `pure::variants` has integrated conflict-resolution support, which is implemented through a conversion of the constraints into Prolog where the dependencies are evaluated [16]. A screenshot of `pure::variants` can be seen in Figure 2.1. The window in the screenshot contains a feature model in the left panel, relations in the right panel, and conflicts between the configuration options in the bottom panel. In the bottom panel we can see that there is a conflict between two of the configuration options. In Figure 2.2, a different view of `pure::variants`' feature model can be seen. It displays the relations and conflicts more visually. `Gears` is another commercial tool. It allows its user to build and manage a complete product line through a feature model [17]. `eCos` on the other hand is an open source real-time operating system which is configured through a configurator called `eCos Configuration Tool` [18]. The feature model is specified with a language called `Component Definition Language (CDL)` [18]. The `eCos Configuration Tool` also supports dependency-checking and reports on any detected conflicts, which makes it possible for its users to resolve the conflicts and ensure a consistent configuration [18]. A screenshot of the `eCos Configuration Tool` can be seen in Figure 2.3. The configuration in the screenshot is

## 2. Background



**Figure 2.1:** A screenshot of the pure::variants configurator.

in an unsatisfied state and in the upper-right corner of the window we can see that one conflict has been detected.

The Linux kernel is bundled with several configurators. These configurators can be seen as feature modeling tools [19]. All of them expose the same feature model, but with different front-ends. The most basic one is called `config`, a screenshot of it can be seen in Figure 2.4, which prompts the user with questions to configure the kernel. Another configurator is `menuconfig`, which can be seen in Figure 2.5. It has a terminal-based interface with menus that the user can navigate through and tick configuration options on and off to configure the kernel. One of the configurators with a graphical user interface is called `xconfig`, which can be seen in Figure 1.1. Common among all Linux configurators is that entering an invalid state is impossible due to only the valid configuration option values being selectable [6]. This means that no interactive conflict-resolution support in the configurators has been implemented, as opposed to eCos and pure::variants. However, there is also no built-in tool for helping the user to activate an inactive configuration option through dependency-resolution.

## 2. Background

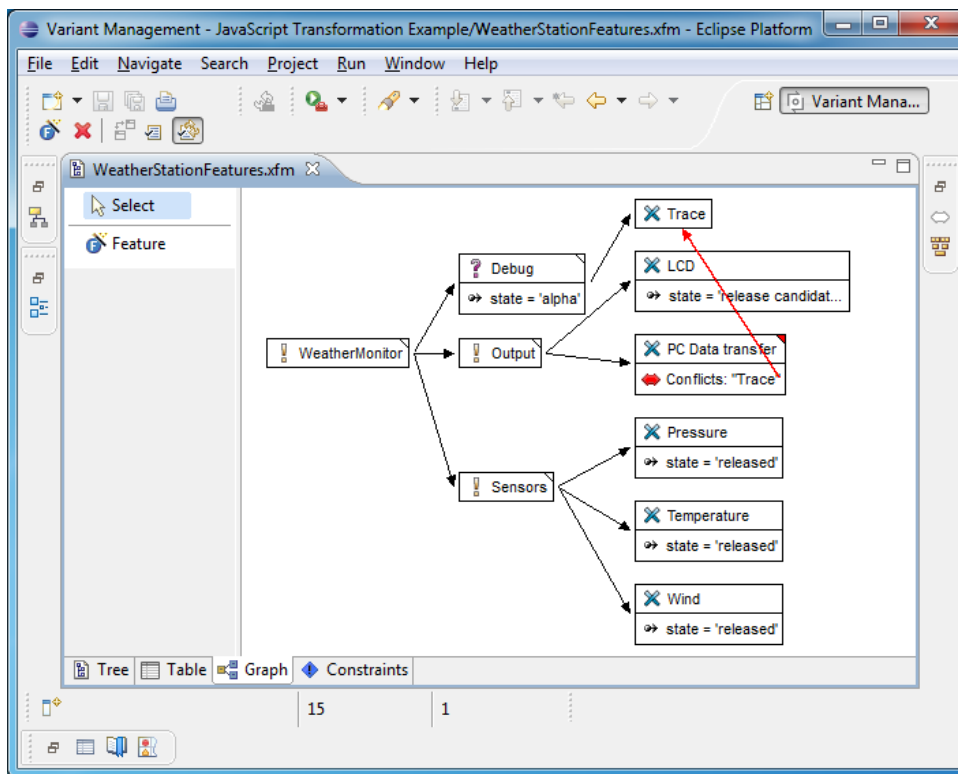


Figure 2.2: A different view of the feature model in pure::variants.

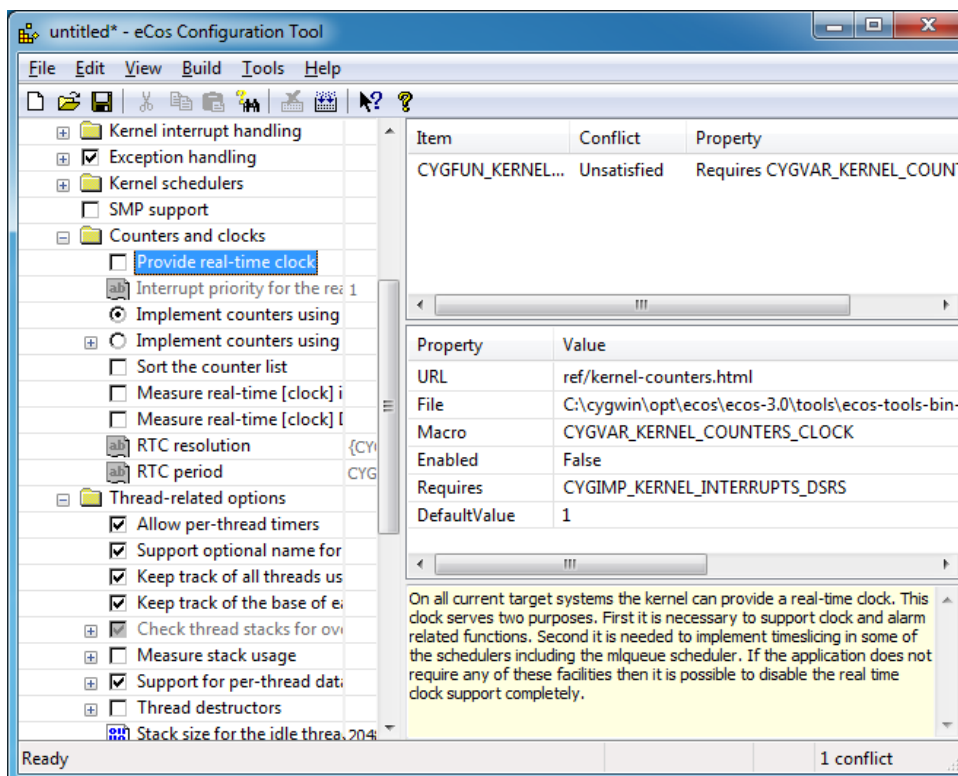
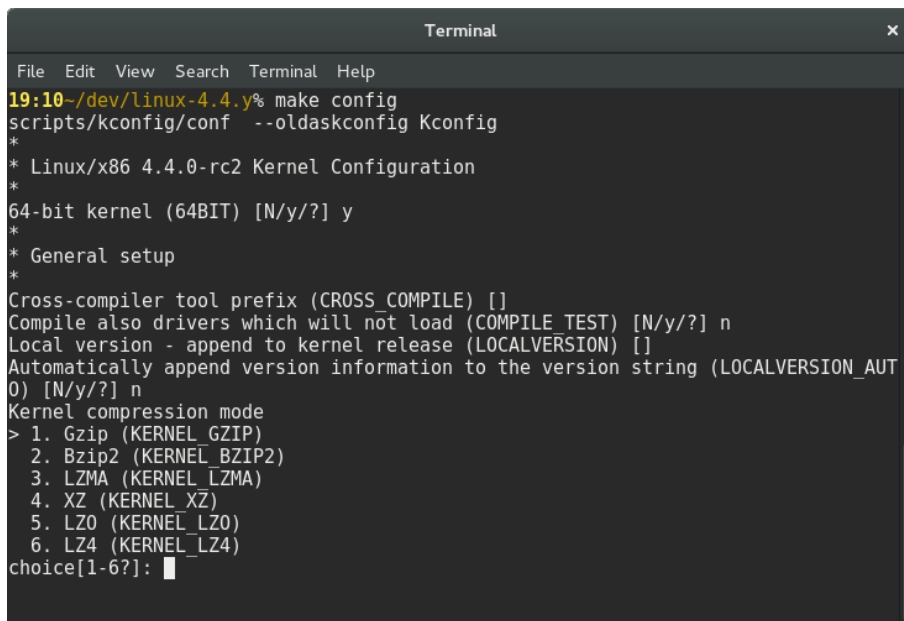


Figure 2.3: A screenshot of the eCos Configuration Tool.

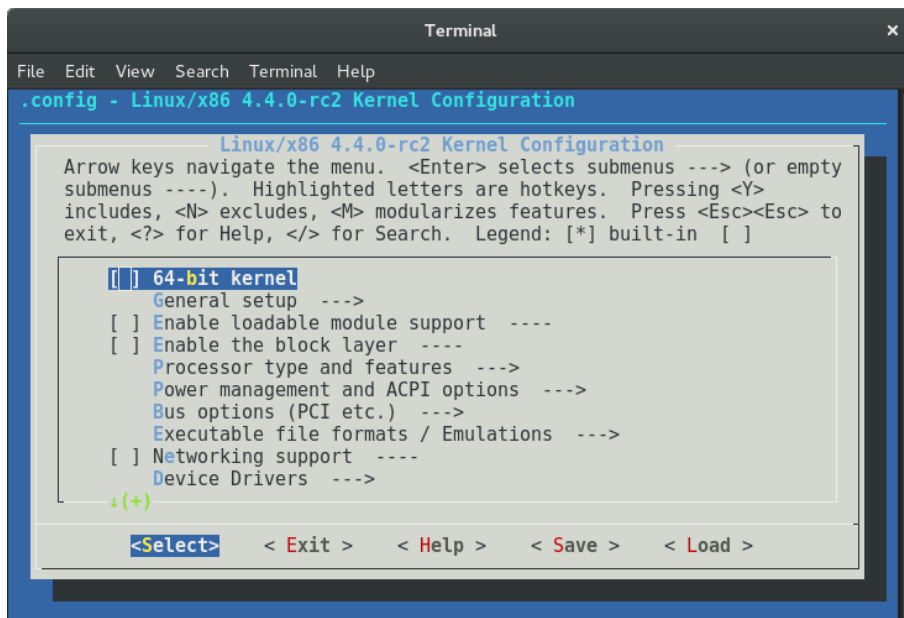


## 2. Background



```
Terminal
File Edit View Search Terminal Help
19:10~/dev/linux-4.4.y% make config
scripts/kconfig/conf --oldaskconfig Kconfig
*
* Linux/x86 4.4.0-rc2 Kernel Configuration
*
64-bit kernel (64BIT) [N/y/?] y
*
* General setup
*
Cross-compiler tool prefix (CROSS_COMPILE) []
Compile also drivers which will not load (COMPILE_TEST) [N/y/?] n
Local version - append to kernel release (LOCALVERSION) []
Automatically append version information to the version string (LOCALVERSION_AUTO) [N/y/?] n
Kernel compression mode
> 1. Gzip (KERNEL_GZIP)
  2. Bzip2 (KERNEL_BZIP2)
  3. LZMA (KERNEL_LZMA)
  4. XZ (KERNEL_XZ)
  5. LZ0 (KERNEL_LZ0)
  6. LZ4 (KERNEL_LZ4)
choice[1-6?]:
```

Figure 2.4: A screenshot of the Linux kernel configurator "config".



```
Terminal
File Edit View Search Terminal Help
.config - Linux/x86 4.4.0-rc2 Kernel Configuration

Linux/x86 4.4.0-rc2 Kernel Configuration
Arrow keys navigate the menu. <Enter> selects submenus ---> (or empty
submenus ----). Highlighted letters are hotkeys. Pressing <Y>
includes, <N> excludes, <M> modularizes features. Press <Esc><Esc> to
exit, <?> for Help, </> for Search. Legend: [*] built-in [ ]

[ ] 64-bit kernel
  General setup --->
  [ ] Enable loadable module support ----
  [ ] Enable the block layer ----
  Processor type and features --->
  Power management and ACPI options --->
  Bus options (PCI etc.) --->
  Executable file formats / Emulations --->
  [ ] Networking support ----
  Device Drivers --->
  (+)

<Select> <Exit> <Help> <Save> <Load>
```

Figure 2.5: A screenshot of the Linux kernel configurator menuconfig.

## 2.2 Kconfig

The Linux kernel's feature model is specified with a language called Kconfig, which at a high level is a collection of configuration options that are organized in a tree structure [20]. These configuration options are sometimes also called symbols, menu entries, or features. An example of a Kconfig file can be seen in Listing 2.1. The most relevant Kconfig concepts for this thesis, among the ones affecting Kconfig's variability management logic [21], are config options, attributes, choices, hierarchies, and propositional logic constraint expressions.

Config options are what the user modifies to configure the kernel. A configuration option is specified in Kconfig with the keyword `config`. Each configuration option has one of the five types `bool`, `tristate`, `string`, `hex` and `int`. Out of these, `tristate` needs some further explanation. A `tristate` behaves like a `bool`, except that it has three states: no, module, and yes. By using `tristate`, it is possible for the user to specify whether a configuration option, for instance a driver, should be compiled into the kernel or be compiled as a loadable module. 94% of the configuration options for the x86 architecture are either of the type `bool` or `tristate` [2].

Attributes are used to enhance configuration options with various properties. They are `prompt`, `default`, `range`, `visible if`, `depends on`, and `select`. `prompt` sets the name of the configuration option that appears in the configurator; if it is absent, the config is invisible. `default` sets the configuration option's default value. `range` is valid for the numerical types and specifies the range that the configuration option's value can be set within. `visible if` controls the visibility of the configuration option in the configurator. With `depends on`, it is possible to specify an expression that must evaluate to true for the user to be able to edit its value in the configurator. `select` is a reverse-dependency; if a configuration option is enabled and it is set to `select` a second configuration option, that second configuration option is forced to also be enabled.

Choice is a group of `bool` or `tristate` configuration options where only a single one may be set to yes, i.e. they are mutually exclusive. In the case when the type is `tristate`, any number of configuration options are also allowed to be set to module. A choice group is useful if there exists multiple drivers for a single hardware but only one can be compiled into the kernel while any number can be compiled as loadable modules [20].

Hierarchies are built by dependencies and menus. Through dependencies, a hierarchy can sometimes be inferred, where a configuration option becomes a child to the configuration option it depends on. A menu, specified with the keyword `menuconfig`, is a regular `config` except that it also has sub-options which gives a hint to the configurator how the configuration options should be presented. In configurators such as `menuconfig` and `xconfig`, a `menuconfig` results in a new level in the tree view.

Constraint logic expressions can be used together with most attributes. The supported operators for boolean and tristate logic are:

- *Constants*. There are three constants: no, `n` (`=0`); module, `m` (`=1`); and yes, `y` (`=2`).

**Listing 2.1:** A partial Kconfig model with six configuration options.

```
1 config MODVERSIONS
2     bool "Set version information on all module symbols"
3     depends on MODULES
4
5 config BT_MRVL_SDIO
6     tristate "Marvell BT-over-SDIO driver"
7     depends on BT_MRVL && MMC
8     select FW_LOADER
9     select WANT_DEV_COREDUMP
10
11 config NR_CPUS
12     int "Maximum number of CPUs (2-32)"
13     range 2 32
14     depends on (AGP || AGP=n) && !EMULATED_CMPXCHG && MMU && HAS_DMA
15     default "32" if ALPHA_GENERIC || ALPHA_MARVEL
16     default "4" if !ALPHA_GENERIC && !ALPHA_MARVEL
17     select I2C_ALGOBIT
18     select DMA_SHARED_BUFFER
19     help
20         MARVEL support can handle a maximum of 32
21         CPUs, all the others with working support
22         have a maximum of 4 CPUs.
23
24 config A
25     tristate "A prompt"
26     depends on B
27
28 config B
29     tristate "B prompt"
30
31 config C
32     tristate "C prompt"
33     select A
```

**Listing 2.2:** A Kconfig .config configuration file.

```
1 CONFIG_MODVERSIONS=y
2 CONFIG_BT_MRVL_SDIO=m
3 CONFIG_NR_CPUS=8
4 # CONFIG_A is not set
5 CONFIG_B=m
6 CONFIG_C=y
```

- *Negation*. Negation is achieved with `!<expression>`, which returns the result of `2 - <expression>`. `y` turns into `n` and vice-versa, while `m` is unaffected.
- *Equality*. The equality between two symbols, i.e. either a configuration option or a constant, is returned with `<symbol 1>=<symbol 2>`. For example, `A=m` returns `y` if the configuration option `A` is set to `m`, otherwise `n`.
- *Inequality*. This works like equality, but the other way around. It is specified with `<symbol 1>!=<symbol 2>` and returns `n` when the symbols are equal and `y` otherwise.
- *Max*. The max-operator returns the largest value of two expressions, and the operator is written `<expression 1> || <expression 2>`. For instance, if the expression is `n || m`, it will be evaluated to `m`.
- *Min*. The min-operator returns the smallest value of two expressions, and the operator is written `<expression 1> && <expression 2>`. For instance, if the expression is `y && m`, it will be evaluated to `m`.

Using these operators, it is possible to write complex expressions for controlling dependencies and other attributes. For instance, `depends on (A && B) || C` and `select A if B=m || C!=n`. More examples of constraint logic formulas can be seen in Listing 2.1.

There is also an attribute called `help` that does not affect Kconfig's variability management [21]. It is used for providing documentation about the configuration options, which can be displayed in the configurators for the users. A screenshot of how it looks like in `xconfig` for the configuration option `OMAP2_DSS` can be seen in Figure 2.6. The description of the configuration option is just "OMAP2+ Display Subsystem support." However, the help text in the configurator also contains other useful information.

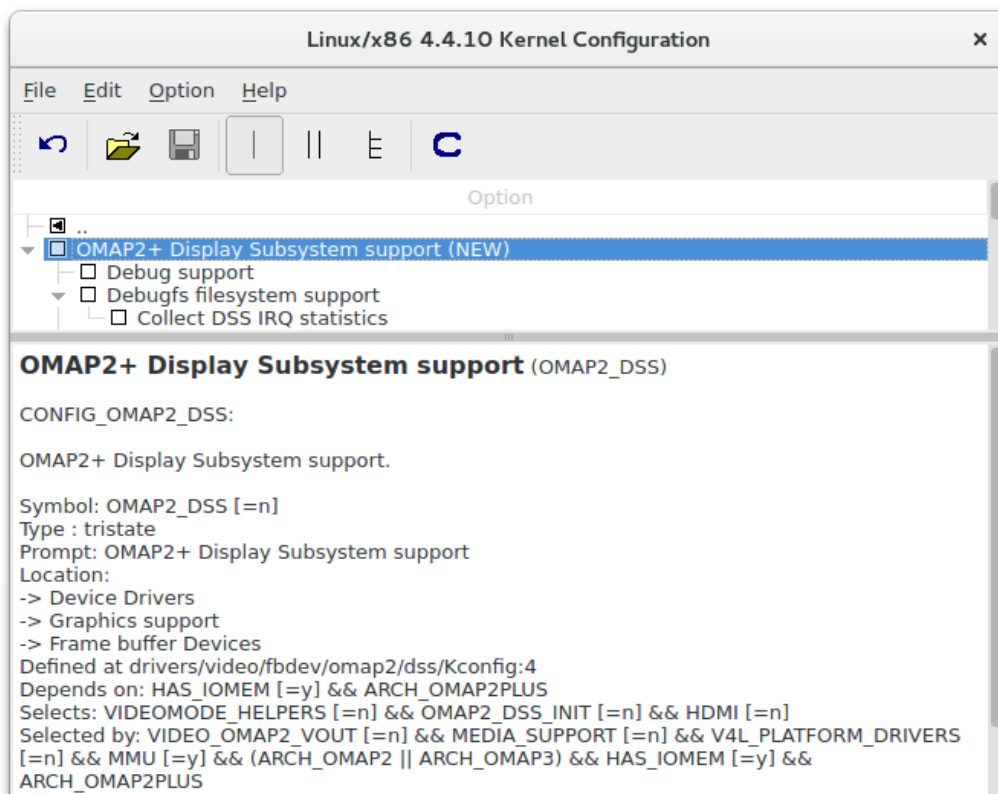
Dependencies and reverse-dependencies give rise to upper and lower bounds. Let us take a look at the three configuration options `A`, `B`, and `C` in Listing 2.1. If `B` is set to `n`, the value of `A` cannot be changed by the user from `n`. But if `B` is to `y` by the user, then `A` can be set to any tristate value. However, if `B` is set to `m`, then `A` can only be set to either `n` or `m`. This illustrates how `depends on` creates an upper bound. In a similar manner, `select` creates a lower bound. If `C` is set to `y`, then `A` is also forced to `y` without the user being able to change its value. If `C` is set to `m`, the user is free to set `A` to either `m` or `y` (given that `B` is set to `y`). But if `C` is set to `n`, the lower bound is `n`.

The upper and lower bounds can also be combined. For instance, if both `B` and `C` are set to `m`, then `A` is also set to `m`; the reason being that `B` creates the lower bound `m`, and `C` creates the upper bound `m`.

The configuration is stored in a file called `.config`. An example of such a file can be seen in Listing 2.2. A hashtag (`#`) creates a comment. Any configuration option that has not been assigned a value is implicitly set to `n`, if it does not have a `default` attribute that overrides it to something else.

### 2.2.1 Internal Kconfig infrastructure

The kernel source tree includes several configurators and they do all share the same code for reading and parsing Kconfig models and `.config` configuration files. The



**Figure 2.6:** Help text in xconfig for the configuration option `OMAP2_DSS`.

function with the signature `void conf_parse(const char *name)` takes the path to a Kconfig file as its parameter and loads the Kconfig model into a global variable with the signature `struct menu rootmenu`. The configuration is loaded with the function `int conf_read(const char *name)`, which takes the path to a `.config` file as its parameter. By loading a configuration, the `rootmenu` data structure is updated with value assignments from the configuration.

Some of the most important Kconfig C structs and their variables are shown in Figure 2.7. Each list item in the configurator maps to an instance of the struct `menu`. The list items are organized in a tree structure, where one's children are pointed at by `menu's` variable `list`. A `menu` can also, with its `sym` variable, point at a `symbol` struct, where the configuration option's data is found. A `symbol's` current configuration value is found in the variable `curr`. The configuration option's attributes are pointed at by `prop`, which are ordered as a single-linked list. The different configuration option types are found in the enum `symbol_type`, and the configuration option attribute types are in the enum `prop_type`.

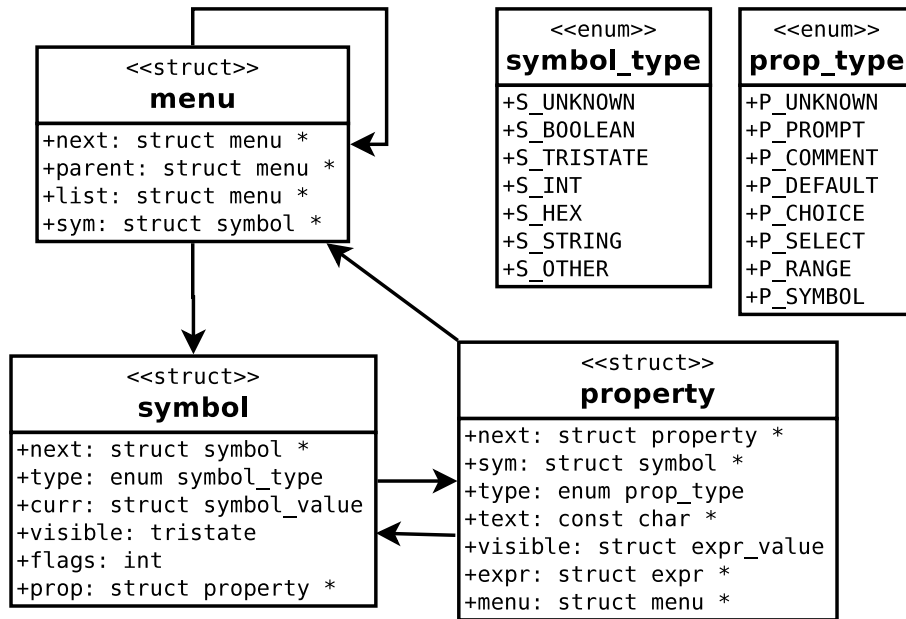


Figure 2.7: Internal Kconfig data structures.

## 2.3 Constraint solvers

There are several different types of solvers for working with constraint satisfaction problems. Two popular choices are SAT and SMT, which are explained in further detail below. The concept of unsatisfiable cores will also be explained.

SAT, an abbreviation of *boolean satisfiability problem*, is used for modeling boolean constraint problems. It only supports boolean typed variables, and the constraint formulas must often be written in CNF [22] (conjunctive normal form). A common format for these problems is DIMACS [23]. An example of a DIMACS encoded problem can be seen in Listing 2.3; it consists of three variables (1, 2 and 3) and the two clauses  $(1 \vee \neg 3) \wedge (2 \vee 3 \vee \neg 1)$ . The problem is satisfiable if there is an assignment of 1, 2 and 3 that makes all clauses evaluate to true. By running it through a SAT solver, we can find out if the problem is satisfiable and what a valid assignment of the variables would be in that case. The primary advantage with SAT is its very good performance which enables it to do extremely fast reasoning with thousands of variables [14].

SMT stands for *satisfiability modulo theories* and is a generalization of SAT. It provides a richer modeling language, where it is possible to work with integers, arithmetic, functions and more. An example of an SMT problem, written in the SMT-LIB language syntax, can be seen in Listing 2.4. In the example, a constant `a` is declared with the type `Int`, and a function `f`, taking two arguments and returning one value, is also declared. Two asserts are made over these declarations, and in the end the satisfiability of the problem is checked. As we can see in this small example, with SMT many problems can be modeled more easily than in SAT.

A concept related to constraint solvers is the notion of *unsatisfiable cores*, which is a subset of constraints in a configuration that in standalone are unsatisfiable.

**Listing 2.3:** An example of a SAT problem written in DIMACS.

```
1 p cnf 3 2
2 1 -3 0
3 2 3 -1 0
```

**Listing 2.4:** An example of an SMT problem written in SMT-LIB.

```
1 (declare-const a Int)
2 (declare-fun f (Int Bool) Int)
3 (assert (> a 10))
4 (assert (< (f a true) 100))
5 (check-sat)
```

Assume we have the boolean variables  $A$ ,  $B$  and  $C$ , and the following constraints:

$$A := true, B := false, C := true, A \leftrightarrow B, B \leftrightarrow C$$

The configuration is unsatisfied since some of the constraints are violated. One unsatisfiable core in this example is  $\{A := true, B := false, A \leftrightarrow B\}$ , which means that if we only consider the constraints in the core, the configuration is still unsatisfiable. The core is also minimal because if we remove any of the constraints from the core, it is possible to find an assignment of the variables that satisfies the constraints. For instance, if we remove  $B := false$ , we only have the constraints  $\{A := true, A \leftrightarrow B\}$  left, and we are able to find a valid assignment to  $A$ ,  $B$  and  $C$ , such as  $A := true, B := true, C := false$ .

## 2.4 Overview of available Kconfig tools

There are several tools and research projects available that can analyze Kconfig models in various ways. Part of the project has been to investigate these configuration options and see what could be re-used for this thesis project. These are listed below.

*Kconfigreader* [24] is a tool implemented in Scala that reads Kconfig files and converts them into CNF formulas in the DIMACS format for further reasoning. It relies on a utility called *dumpconf*, which is implemented in C, that utilizes Linux's Kconfig infrastructure to dump the internal representation as XML, which is then parsed by Kconfigreader.

*Undertaker* [25] is a tool implemented in C++ that parses Kconfig models and does configuration analysis on them. It can check the structure of preprocessor directives in the Linux kernel's source code against different configuration models to find blocks of features that cannot be selected or deselected.

*LVAT* [26, 27] (Linux Variability Analysis Tools) is a tool suite written in Scala for analyzing Kconfig models. It currently offers three different tools: propositional formula extractor, Kconfig statistics, and feature model translator. It relies on a utility called *exconfig* [28] (Linux Kconfig Extractor), which is implemented in C, that utilizes Linux's Kconfig infrastructure to dump the internal representation to an *.exconfig* file [29], which is then parsed by LVAT.

*RangeFix* [30, 31] is an algorithm for finding fixes to resolve configuration conflicts. A proof-of-concept with a command-line interface has been implemented in Scala that works with both Kconfig and eCos’ CDL files. The Kconfig version depends on LVAT, and takes an `.exconfig` file, a `.config` configuration file, a configuration option name and a requested value as inputs. If setting the configuration option to the requested value results in a conflict, RangeFix returns one or more fixes that the user can implement in her configuration to set the configuration option to the requested value and satisfy its dependencies. The algorithm depends on a constraint solver, and the Scala implementation utilizes the SMT solver Z3 [32].

*Satconfig* [33] is a tool for creating a valid `.config` configuration file from a partial configuration. The user only has to set the values of the configuration options she is interested in, and Satconfig will automatically fill in any blanks to satisfy their dependencies. However, if there are any conflicts, Satconfig quits since it is not able to resolve any conflicts. Satconfig is written in C and is integrated as an additional command in the Linux kernel’s build system. It started as a Google Summer of Code project, but it has not been merged upstream and is only available in a downstream fork of the kernel repository [33]. Satconfig translates the Linux kernel’s internal Kconfig data structure directly into CNF clauses and appends them to the SAT solver PicoSAT [34] by using its C API.

## 2.5 RangeFix

RangeFix was already introduced in the previous section. However, since this thesis bases a lot of its work on RangeFix, the algorithm needs a more detailed explanation.

RangeFix is a novel algorithm for finding minimal fixes to a configuration that resolve any unsatisfied constraints [30]. Since it is just an algorithm, it is not bound to any feature modeling language in particular. However, the algorithm has been tested with both Kconfig and CDL through an implementation in Scala.

The functionality of RangeFix is best illustrated by a small example, taken from the RangeFix paper [30]. Assume the following set of configuration options are declared in a feature model:

$$\{m : \text{Bool}, a : \text{Int}, b : \text{Int}\}$$

Furthermore, assume that the feature model contains the following three propositional logical constraints:

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

Our feature model now contains three variables ( $m$ ,  $a$  and  $b$ ) and three constraints over these configuration options. Continuing with the example, assume that there is also the following configuration:

$$\{m := \text{true}, a := 6, b := 5\}$$

Values have now been assigned to the feature model’s variables. However, these value assignments violate a couple of the constraints. More specifically, the first and



the last constraints are violated:

$$(\mathbf{true} \rightarrow 6 > 10) \wedge (\neg\mathbf{true} \rightarrow 5 > 10) \wedge (6 < 5)$$

The first is violated because  $(\mathbf{true} \rightarrow 6 > 10) = (\mathbf{true} \rightarrow \mathbf{false}) = \mathbf{false}$ . The third is violated because  $(6 < 5) = \mathbf{false}$ . Clearly, some of the variables need to have their values changed to satisfy the constraints. By applying the RangeFix algorithm to this problem, the following two fixes are generated:

- $[m := \mathbf{false}, b : b > 10]$
- $[(a, b) : a > 10 \wedge a < b]$

All conflicts will be resolved and the configuration satisfied if one of these two fixes is applied. In other words, the user has two options in how to resolve the conflicts and satisfy the constraints. If the first fix is selected, the assignment to  $m$  needs to be changed from  $\mathbf{true}$  to  $\mathbf{false}$ , and  $b$  from 5 to something greater than 10.  $a$  does not need to be changed in the first fix. If the second fix is selected, both  $a$  and  $b$  need to be changed in such a way that  $a > 10$  and  $a < b$  are true.

### 2.5.1 RangeFix's three stages

RangeFix generates fixes in three separate steps called *stages*. The variables that need to be changed are found during the first stage. These variables are organized in sets, called *diagnoses*. Returning to the previous example, the two located diagnoses were:

- $\{m, b\}$
- $\{a, b\}$

In the second stage, the constraints are transformed into *modified constraints*. The transformation is done for each diagnosis by retrieving the constraints affecting the diagnosis and replacing all variables in the constraints that are not part of the diagnosis with the configuration's value. Continuing with the running example, the two modified constraints were:

- $(m \rightarrow 6 > 10) \wedge (\neg m \rightarrow b > 10) \wedge (6 < b)$
- $(\mathbf{true} \rightarrow a > 10) \wedge (\neg\mathbf{true} \rightarrow b > 10) \wedge (a < b)$

In the first modified constraint, the applied diagnosis was  $\{m, b\}$ , and all other configuration options have therefore been substituted with their configuration value; in this case have all occurrences of  $a$  been substituted with 6. In the second modified constraint have all occurrences of  $m$  been substituted with  $\mathbf{true}$ .

The modified constraints are minimized into fixes during the last, and final, stage of RangeFix. The minimization is achieved through a process where several heuristic rules are applied to rewrite the modified constraints into simpler units. The end result of this stage are the fixes:

- $[m := \mathbf{false}, b : b > 10]$
- $[(a, b) : a > 10 \wedge a < b]$

## 2.5.2 Generating diagnoses

A diagnosis is

- a subset of configuration options that are violating one or more of the constraints, and is
- able to satisfy the constraints when the values of its configuration options are changed properly.

An important property of a diagnosis is that each configuration option in the diagnosis is part of an unsatisfiable core. Furthermore, for a diagnosis to be able to satisfy the configuration, it must contain at least one configuration option from each unsatisfiable core. To be able to understand how this works, a closer look at hard and soft constraints in conjunction with a satisfiability solver is needed.

Returning to the running example for this section, the constraints were:

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

These are considered to be hard constraints, because lessening them is not an option when seeking for a valid configuration. Continuing on, there was also the following configuration:

$$\{m := \mathbf{true}, a := 6, b := 5\}$$

These assignments are the soft constraints. It would be best if as many assignments as possible could be made, but if that is not possible due to constraint violations, some need to be edited.

Constraint solvers often only understand the concepts of variables and constraints, and there is usually no distinction made between hard and soft constraints. However, assuming that there is at least one assignment of the configuration options that do not cause any constraint violations, each unsatisfiable core must contain at least one soft constraint. Because it is only from the user's configuration choices that conflicts can be introduced. In the running example, the union of the constraints are:

$$(m \rightarrow a > 10) \wedge (\neg m \rightarrow b > 10) \wedge (a < b) \wedge (m := \mathbf{true}) \wedge (a := 6) \wedge (b := 5)$$

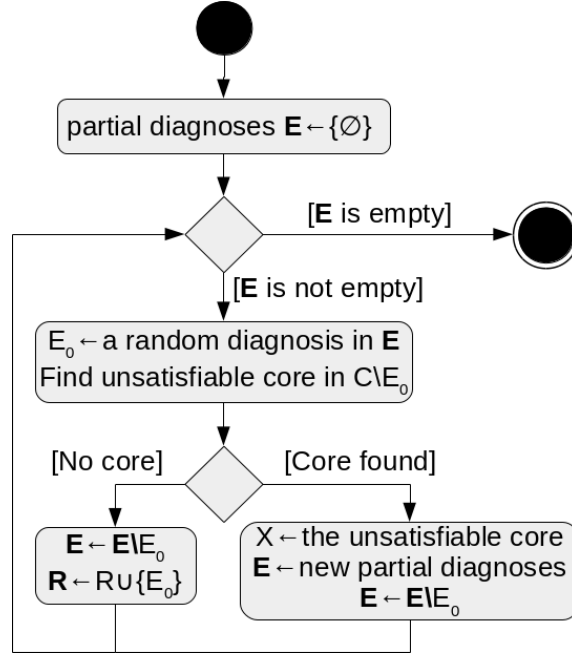
The minimal unsatisfiable cores are:

- $(m \rightarrow a > 10) \wedge (m := \mathbf{true}) \wedge (a := 6)$
- $(a < b) \wedge (a := 6) \wedge (b := 5)$
- $(m \rightarrow a > 10) \wedge (\neg m \rightarrow b > 10) \wedge (m := \mathbf{true}) \wedge (b := 5)$
- $(m \rightarrow a > 10) \wedge (\neg m \rightarrow b > 10) \wedge (a := 6) \wedge (b := 5)$
- $(m \rightarrow a > 10) \wedge (\neg m \rightarrow b > 10) \wedge (a < b) \wedge (b := 5)$

A diagnosis contains at least one soft constraint from each minimal unsatisfiable core. By selecting either  $(m := \mathbf{true}) \wedge (b := 5)$  or  $(a := 6) \wedge (b := 5)$ , all unsatisfiable cores are covered. In other words, by changing the values of either  $\{m, b\}$  or  $\{a, b\}$ , it is possible to satisfy the constraints.

In Figure 2.8, a visualization of the first stage of the RangeFix algorithm can be seen on a high level in the form of a flowchart. Starting with an empty partial diagnosis, the algorithm enters a loop where it tries to expand the partial diagnosis

into more partial diagnoses and make them as large as possible. When a partial diagnosis does not cause any more unsatisfiable cores to be detected, it is appended to the set  $R$  as a diagnosis. The important part to note here is that the constraint solver has to be invoked during each iteration to tell whether the partial diagnosis is large enough to eliminate all unsatisfiable cores; if not, it will extract an unsatisfiable core to construct one or more larger partial diagnoses.



**Figure 2.8:** A flowchart of the first stage of the RangeFix algorithm on a high level.

### 2.5.3 Encode a Kconfig model as an SMT problem

The Scala implementation of RangeFix utilizes an SMT solver to find and extract unsatisfiable cores. A brief overview of how a Kconfig model is encoded as an SMT problem is given in this section.

Boolean variables are encoded with SMT’s primitive type `Bool`, while tristates are encoded with an enum with three states. The enum is called `__enum__0`, and its definition is shown in Listing 2.5.

For each configuration option that appears in the Kconfig model, two SMT variables are allocated. The first variable contains the configuration option’s value, and its type maps to its corresponding type in the Kconfig model. For instance, a `tristate` in the Kconfig model maps to a `__enum__0` in SMT, while a `bool` in the Kconfig model maps to a `Bool` in SMT. The second variable is a `Bool`, regardless of

**Listing 2.5:** The tristate type declared with an enum.

```

1 (declare-datatypes () ((__enum__0
2   __s__enum_int__0_0 __s__enum_int__0_1 __s__enum_int__0_2)))

```

**Listing (2.6)** A Kconfig model with exconfig's syntax.

```
1 config A tristate {
2 }
3 config B boolean {
4 }
```

**Listing (2.7)** A corresponding .config configuration file.

```
1 CONFIG_A=n
2 CONFIG_C=n
```

**Listing (2.8)** Allocated SMT variables from Listing 2.6, and assert commands to control their value from and Listing 2.7.

```
1 (declare-const A __enum__0)
2 (declare-const B Bool)
3 (declare-const __gd__A Bool)
4 (declare-const __gd__B Bool)
5 (assert (or (not __gd__A) (= A __s__enum_int__0_0)))
6 (assert (or (not __gd__B) (= B false)))
```

**Listing 2.9:** The allocated SMT variables and the asserts to control their value source.

```
1 (assert (! __gd__A :named __ex____gd__A))
2 (assert (! __gd__B :named __ex____gd__B))
```

the configuration option's type, and it is used to determine the first variable's value source. If the second variable is set to `True`, the first variable is assigned to what is declared in the `.config` file. If the second variable is instead set to `False`, the SMT solver is free to determine the value of the first variable.

An example of a Kconfig model, a `.config` file and the corresponding SMT variables are shown in Listing 2.6, Listing 2.7 and Listing 2.8. In Listing 2.8, the SMT solver will use the values from the `.config` file when `__gd__A` and `__gd__B` are set to `True`. This is done with the two assert commands in Listing 2.9, which also name the variables to be able to extract the unsatisfiable cores that the configuration might give rise to.

The constraints are declared using a set of various functions. A minimal example for a configuration option `C` that has no dependencies is shown in Listing 2.10. This small example illustrates how the Scala implementation of RangeFix creates functions for each configuration option that defines its lower and upper bounds. At the last line, an assert is made to set `C`'s value to `yes` by specifying that the function `C_effective`'s return value should be equal to `__s__enum_int__0_2`.

**Listing 2.10:** An example of SMT functions.

```
1 (define-fun C__upperBound () Bool true)
2 (define-fun C__inherited () __enum__0 __s__enum_int__0_2)
3 (define-fun C__lowerBound () Bool false)
4 (define-fun C__rangedUserValue () Bool
5   (or (and C C__upperBound) C__lowerBound))
6 (define-fun C__default () Bool
7   (or (and false true) C__lowerBound))
8 (define-fun C__effective () Bool
9   (ite (= C__inherited __s__enum_int__0_0)
10    C__default C__rangedUserValue))
11 (assert (= __s__enum_int__0_2 C__effective))
```

## 2.6 Related works

**Usability of the Linux configurators.** Improving the usability of the Linux kernel configurators has been studied by Bak et al. [35]. In their study, they had identified the usability to be lacking, and used xconfig as an example to highlight these usability problems. They implemented a prototype that they called lkc, which would eliminate many of their identified usability problems. lkc was evaluated with users, and they received positive feedback saying that their changes were definitely an improvement over xconfig. Among the users' desired additional features was conflict-resolution.

**Formalizing the Kconfig language.** Since the Kconfig language lacks a formal specification, semantics of it have been described by studying the behavior exhibited in xconfig [26]. An experiment to formalize the language, by defining propositional formulas that describe all valid configurations of the Linux kernel, has also been made [36, 37], where implementing it with SAT solvers is highlighted as a possible application.

**Resolving configuration conflicts.** As mentioned in Section 2.4, there are several projects that have implemented various reasoners for Kconfig feature models. One of them was RangeFix [30], which is an algorithm for resolving configuration conflicts. It was also mentioned that RangeFix has a Scala implementation that works with Kconfig feature models.

**Feature model analysis.** Through a variety of papers, different aspects of the Linux kernel's feature model have been studied. One such aspect is the evolution of the kernel's feature model, which involves the growth of the number of features [9, 7]. Another aspect is finding zombie features, which are bugs where features cannot be either enabled or disabled at all [38]. How features are scattered in a non-modular way has also been researched [8].

**Knowledge based configurators.** Much work has been done for configurators for physical goods in the field of knowledge-based configuration, a subfield of AI, but

relatively little research has been done in the area of configurators for software [39]. In manufacturing, companies adopt configurators to gain commercial advantages such as flexibility, lower production costs and increased customer satisfaction [14]. These are built on models called product variant master, which is a set of hierarchically organised components that aims at capturing variability of the product range [14]. Abbasi et al. [40] did a study on 111 web-based product configurators and found that one can do a lot of parallels to software configurators. Since there exists a big overlap between the fields, an attempt at unifying these two fields has recently been made [39].

# 3

## Methodology

The methodology applied in this case study has been the one described by design science research [13]. Design science research is suitable for development projects and it prescribes a process of six activities. The six activities are 1. *problem identification and motivation*, 2. *objectives of a solution*, 3. *design and development*, 4. *demonstration*, 5. *evaluation*, and 6. *communication*. How these activities have been applied in this case study is expanded upon in the following sections in this chapter.

### 3.1 Problem identification and motivation

In Section 1.1, the relevancy of improving the configuration process of Kconfig feature models was argued for. Statistics from a user survey were presented, telling us that the configurators were indeed lacking in usability. Confirming the issues were kernel developers, who have started an initiative called kconfig-sat, where the aim is to add support for automatic dependency-resolution. From the gathered evidence, it is apparent that it would be beneficial if people who configure Kconfig based software could receive aid from the configurator with resolving unmet dependencies.

### 3.2 Objectives of a solution

With the problem having been identified, the next step was to formulate an objective on how to best address it. In Section 2.1, we presented examples of other feature modeling tools similar to the Linux kernel's configurator xconfig. Two of those were the eCos Configuration Tool and pure::variants, both of which have built-in support for conflict-resolution assistance. Using these solutions as an inspiration, implementing a similar solution for the Linux kernel seemed like a reasonable way to address the issue. We also noted in Section 1.1 that there are Linux users who have also arrived at the same conclusion—that adding support in the Kconfig configurators for resolving dependencies would be beneficial. The objective therefore became to implement an interactive conflict-resolution tool that integrates with one of the Kconfig configurators. Whenever the user attempts to enable a configuration option with unmet dependencies, the configurator could present a list of possible fixes that would resolve the unmet dependencies.

### 3.3 Design and development

The design and development activity was initiated by an investigation of available tools related to Kconfig. An overview of the tools that were investigated is presented in Section 2.4. Among these, the ones that appeared as having the highest potential in achieving the goal of the research questions were RangeFix and Satconfig. RangeFix is an algorithm for resolving configuration conflicts, which has an existing Scala implementation that works with Kconfig. Satconfig is tool for automatically completing a partial `.config` configuration file, that is implemented in C, and contains a mechanism for translating a Kconfig model into a SAT problem. By combining these projects, the plan was to realize a C implementation of RangeFix that utilizes Satconfig’s translation of Kconfig models into SAT problems. To make the conflict-resolution process interactive, the implementation of GUI additions in one of the Kconfig configurators to support such a workflow also had to be achieved.

The existing Scala implementation of RangeFix utilizes an SMT solver for encoding the Kconfig model and detecting unsatisfiable cores. Since SMT provides a richer modeling language than SAT, it is therefore more straightforward to model Kconfig constraints that involve strings and integers. Another strength with SMT in our context is that the Scala implementation of RangeFix has already been implemented using SMT. On the other hand, only 6% of the configuration options for x86 are of other types than bool and tristate [2], which means that for the majority of cases a SAT solver is enough. Furthermore, the Linux community acceptance factor for SMT is also much lower than for SAT, and many would like to see the SAT avenue investigated first [41]. Reasons for preferring SAT are its relative simplicity and its smaller code base (~10k lines of code for a SAT solver compared to ~100k lines of code for an SMT solver) [42]. The decision was therefore made to proceed with using a SAT solver in the C implementation of RangeFix.

After the selection of tools to use as a foundation, a prototype was implemented where the user would be able to perform conflict-resolution interactively. This involved modifying the Kconfig configurator `xconfig` with extra GUI elements to support the workflow. It was made to call the Scala implementation of RangeFix in the background, and print the returned fixes inside `xconfig`.

Next, the implementation of RangeFix in C took place. This involved examining the algorithm and Satconfig’s translation of Kconfig models into SAT problems. However, only the first stage of the RangeFix algorithm was implemented, which means that it would only be able to generate diagnoses and not complete fixes. A version of `xconfig` that uses the C implementation for conflict-resolution was also developed.

Details about our implementation of RangeFix in C, based on a SAT-solver, are shared in Chapter 4. How our implementation of RangeFix and the existing Scala implementation of RangeFix were integrated with `xconfig` is also told in that chapter.



## 3.4 Demonstration

From the design and development activity, two artifacts were produced. Both were versions of `xconfig`, but they were calling different implementations of `RangeFix` for dependency-resolution. One was a version of `xconfig` for version 2.6.32 of the Linux kernel that made calls to the Scala implementation of `RangeFix` for fix generation. The second was a version of `xconfig` for version 4.4 of the Linux kernel that generated diagnoses using our C implementation of `RangeFix`. These two artifacts are demonstrated in Chapter 5.

## 3.5 Evaluation

The evaluation of the two produced artifacts is presented in Chapter 6. The qualities performance, correctness and usability were evaluated.

Through a survey with Linux kernel developers and users, this thesis' direction was evaluated. In Section 6.1, the design and results of the survey are presented. The participants were asked to fill in the survey that can be seen in Appendix A. They were, among things, questioned on the method they currently use to enable a disabled configuration option, and the time it takes on average to accomplish that task. This established a baseline for how fast an interactive dependency-resolution solution would need to function. The participants were also shown a video of our version of `xconfig` that calls the Scala implementation of `RangeFix`, and through several questions they were asked to provide various kinds of feedback.

The Scala implementation of `RangeFix` was evaluated with respect to performance and correctness. The evaluation procedure and the results are found in Section 6.2. But to summarize the evaluation, a set of disabled configuration options were randomly sampled, and for each one the Scala implementation of `RangeFix` was invoked to generate fixes for enabling it. The time from when the program was started until it returned was recorded to determine the performance of the implementation. Furthermore, the time spent generating the diagnoses, and converting them to fixes, was also recorded. For each run, the correctness of the generated fixes was evaluated by testing them in `xconfig`. Depending on the quality of the generated fixes in this test, one of five different labels was given to summarize its correctness.

The C implementation of `RangeFix` was also evaluated with respect to performance and correctness. The evaluation, whose procedure and results are presented in Section 6.3, was carried out similarly to the previous one for the Scala implementation. However, since we were only able to implement in C the first stage of the algorithm, that yields the diagnoses, only that part was evaluated. As with the previous evaluation, a set of disabled configuration options were first randomly sampled. For each configuration option, the performance of generating the diagnoses was evaluated by measuring the running time of the program. The correctness of the diagnoses were evaluated by testing if it was possible to enable the configuration option by only modifying the values of the configuration options in them. Using the same five correctness labels as for the Scala implementation, the results were categorized in a comparable way.

By using the numbers collected in the survey as a baseline, it was possible to evaluate whether the Scala implementation of RangeFix performed well enough. Furthermore, by comparing the C implementation's running time against the Scala implementation's time spent at generating diagnoses, it was possible to determine if implementing it in C meant a performance improvement. With the performance and correctness statistics, it would also be possible to decide whether continuing implementing RangeFix in C with a SAT-solver is meaningful to explore in future projects.

## 3.6 Communication

This thesis is used as a medium for communicating the results from the project. It includes implementation details, demonstration of the produced artifacts, and evaluation of the artifacts. Furthermore, observations and ideas that might help in future iterations of developing an interactive conflict-resolution mechanism are shared in Chapter 7. The information has also been shared with the Linux developers.

# 4

## Design and development

Two artifacts have been produced to investigate the feasibility of realizing support for interactive conflict-resolution in the Kconfig configurator `xconfig`. Both are versions of the Kconfig configurator `xconfig`, but they utilize different implementations of `RangeFix` for assisting the user in satisfying unmet dependencies. One of them utilizes the existing Scala implementation of `RangeFix`, while the other utilizes our implementation of `RangeFix` in C that depends on a SAT solver.

The first five sections of this chapter describe design elements that make up our partial implementation of `RangeFix` in C. Together, they implement the first stage of the `RangeFix` algorithm. In Section 4.1, the encoding of a Kconfig model as a SAT problem is described. Next, in Section 4.2, how the user's configuration is read and translated into soft constraints in the SAT problem is explained. In Section 4.3, we take a look at how unsatisfiable cores are generated in the existing Scala implementation of `RangeFix` with an SMT solver, and how we achieve the same thing in C with a SAT solver. These sections are tied together in Section 4.4, where we explain how we generate the diagnoses for the first stage of the `RangeFix` algorithm. Lastly, in Section 4.5, we explain why and how we simplify the diagnoses by removing redundant configuration options.

The last section, Section 4.6, describes how both implementations have been integrated with `xconfig`. The Linux configurators are not engineered for letting the user cause conflicts, which meant that some compromises in the design had to be made. Furthermore, since the codebase for the existing Scala implementation has not been maintained for a long time, some further compromises had to be made with regards to the utilized Linux kernel versions.

### 4.1 Encode a Kconfig model as a SAT problem

In this section, our encoding of a Kconfig model as a SAT problem is presented. This encoding is to a large extent based on the encoding found in `Satconfig` [33], which is unpublished and to a large extent undocumented. The encoding can be divided into two separate phases. First, the configuration options, declared in the Kconfig model, are translated into boolean literals in the SAT problem. Next, the constraints imposed by the various Kconfig attributes are translated into CNF clauses in the SAT problem.

**Table 4.1:** Encoding a `tristate` with two literals.

Tristate value	Literal 1	Literal 2
n	0	0
y	1	0
m	1	1

**Table 4.2:** Allocated literals for each configuration option.

Literal	Used for	Allocated when
1	The option's value (yes or no)	Always present
2	The option's value (module or not)	Present only if tristate
3	If the option's value has been set by the user	Always present
4	If the option is <b>selected</b> by another symbol	Always present
5	If the option's dependencies are satisfied	Only present if the option has a prompt attribute
6	If the option's default constraint is satisfied	One for each default value
7	Disallow the option's value to be set manually	Always present

### 4.1.1 Configuration option encoding

The two configuration option types to consider are `bool` and `tristate`. The value of a configuration option with the type `bool` is simply encoded using a single literal. However, a configuration option with the type `tristate` has three states, and therefore needs two literals to encode its value. How the `tristate`'s states are encoded using two literals is depicted in Table 4.1.

To aid in the constraints encoding, a set of additional literals are allocated for each configuration option. Besides one literal for a `bool` or two literals for a `tristate`, literals for five additional properties are allocated. The additional literals encode if its value has been explicitly set by the user, if it has been **selected** by another configuration option, if its prompt's dependencies have been satisfied, if its defaults' expressions have been satisfied, and if the value must be set implicitly. The literals allocated for each configuration option, and the number of instances of them, are summarized in Table 4.2. The last literal, that forces the option's value to be set implicitly, is more of a helper that is used when simplifying a diagnosis by removing already indirectly set configuration option values.

With the Kconfig model declared in Listing 4.1, the literals listed in Table 4.3 are allocated. Two literals are allocated for each **A** and **C** to hold their values, since they are of the type `tristate`, while **B** only needs one. Each configuration option gets the two mandatory literals that say if its value has been set by the user and if it has been **selected** by another configuration option. All three configuration options

**Listing 4.1:** A Kconfig model.

```
1 config A
2     tristate "A prompt"
3
4 config B
5     bool "B prompt"
6     select A
7     default y
8
9 config C
10    tristate "C prompt"
11    depends on A && !B
```

have a prompt and are therefore allocated a literal each that says if the configuration option's dependencies have been satisfied. Lastly, since B has a default attribute, a literal is also allocated that says if its `if` expression has been satisfied.

### 4.1.2 Constraints encoding

Various propositional logic constraints can be inferred from a Kconfig model, and their purpose is to limit the configurations to what the Kconfig language permits. Since a SAT solver is being used, the constraints must be written as CNF clauses, which are also easily expressed in the DIMACS format.

#### 4.1.2.1 Tristate

A `tristate` has three states and is encoded with two literals, as depicted in Table 4.1. The state (0, 1) is invalid, and this is enforced by the constraint:

$$(\textit{Literal2} \rightarrow \textit{Literal1}) \equiv (\neg \textit{Literal2} \vee \textit{Literal1})$$

If the second literal is true and the first literal is false, which happens in the invalid state, the clause evaluates to false. Continuing with the Kconfig example in Listing 4.1 and its literals allocation depicted in Table 4.3, this constraint gives rise to the following two CNF clauses:

$$(\neg A_2 \vee A_1) \equiv (\neg 2 \vee 1)$$

$$(\neg C_2 \vee C_1) \equiv (\neg 14 \vee 13)$$

In DIMACS they correspond to `-2 1 0` and `-14 13 0`.

#### 4.1.2.2 Select

If a configuration option is activated and it has a `select` attribute, the configuration option it `selects` is also activated.

In the example in Listing 4.1, B has the attribute `select A`. This translates to the propositional formula  $(B_1 \rightarrow A_4) \equiv (\neg B_1 \vee A_4) \equiv (\neg 7 \vee 4)$ , which means that if B is set to y, A is selected. In DIMACS it is equal to `-7 4 0`.

**Table 4.3:** The literals allocated for the Kconfig model in Listing 4.1.

		Literal	Used for
A	$A_1$	1	Yes/no value
	$A_2$	2	Module/yes value
	$A_3$	3	If value is set by the user
	$A_4$	4	If the symbol has been selected
	$A_5$	5	If the prompt's dependency is fulfilled
	$A_6$	6	If the value must be implicitly set
B	$B_1$	7	Yes/no value
	$B_2$	8	If value is set by the user
	$B_3$	9	If the symbol has been selected
	$B_4$	10	If the prompt's dependency is fulfilled
	$B_5$	11	If the default's dependency is fulfilled
	$B_6$	12	If the value must be implicitly set
C	$C_1$	13	Yes/no value
	$C_2$	14	Module/yes value
	$C_3$	15	If value is set by the user
	$C_4$	16	If the symbol has been selected
	$C_5$	17	If the prompt's dependency is fulfilled
	$C_6$	18	If the value must be implicitly set

Furthermore, to enforce that whenever a configuration option is selected a lower bound is created, we need some more constraints. These are  $(A_4 \rightarrow A_1) \equiv (\neg A_4 \vee A_1)$ ,  $(B_3 \rightarrow B_1) \equiv (\neg B_3 \vee B_1)$ , and  $(C_4 \rightarrow C_1) \equiv (\neg C_4 \vee C_1)$ . In DIMACS, these constraint clauses are equal to  $-4 \ 1 \ 0$ ,  $-9 \ 7 \ 0$  and  $-16 \ 13 \ 0$ . For instance, when **A** is **selected**, which happens when  $A_4$  is true, it has to be set to either **m** or **y**. Note that this is a simplification that does not deal with the complete complexity of **select**'s lower bound.

#### 4.1.2.3 Prompt

A configuration option's dependency, specified with the attribute **depends on**, controls the visibility of the configuration option's prompt. If the dependency is not satisfied, then the prompt is invisible and the user is not able to change the configuration option's value.

All three configuration options in Listing 4.1 have a prompt. For **A** and **B**, the prompt is implicitly always enabled since no dependency has been declared. **A**'s literal that specifies if its prompt's dependency is fulfilled is 5. The CNF formula is therefore simply (5), and in DIMACS the clause is  $5 \ 0$ . For **B**, the DIMACS clause is equal to  $10 \ 0$  in an analogous way.

**C** on the other hand has the dependency **A && !B**, which translates into the propositional formula  $C_5 = A_1 \wedge \neg B_1 \equiv 17 = (1 \wedge \neg 7)$ . In CNF, it is equal to the three clauses  $(17 \vee \neg 1 \vee 7)$ ,  $(\neg 17 \vee 1)$  and  $(\neg 17 \vee \neg 7)$  [43]. In DIMACS, the clauses are equal to  $17 \ -1 \ 7 \ 0$ ,  $-17 \ 1 \ 0$  and  $-17 \ -7 \ 0$ . Note that this is a simplification that does not deal with the complete complexity of **depends on**'s upper bound.

#### 4.1.2.4 Default

A configuration option gets its `default` value if the `default` attribute's optional constraint is fulfilled, the option is not `selected` by another option and the user has not already assigned a value to it.

$A$  has the implicit default value `n`. It gets its default value if it is not `selected` by another symbol and the user has not given it a value. This translates into the propositional formula  $((\neg A_3 \wedge \neg A_4) \rightarrow \neg A_1) \equiv \{\text{De Morgan's law [44]}\} \equiv (\neg(A_3 \vee A_4) \rightarrow \neg A_1) \equiv (A_3 \vee A_4 \vee \neg A_1) \equiv (3 \vee 4 \vee \neg 1)$ . In DIMACS it is equal to `3 4 -1 0`.

$B$  has the explicit default value `y`. If it has not been `selected`, the user has not assigned a value to it, and the `default`'s constraint is fulfilled, it should fall back on its default value `y`. This yields us the constraint  $((\neg B_2 \wedge \neg B_3 \wedge B_5) \rightarrow B_1) \equiv (\neg(B_2 \vee B_3 \wedge \neg B_5) \rightarrow B_1) \equiv (B_2 \vee B_3 \vee \neg B_5 \wedge B_1) \equiv (8 \vee 9 \vee \neg 11 \wedge 7)$ , which in DIMACS is `8 9 -11 7 0`. Furthermore, since the `default` attribute lacks an `if` expression, it is always true, and we also get the additional constraint clause  $(B_5) \equiv (11)$ , which in DIMACS is `11 0`.

$C$  also has the implicit default value `n`. The CNF clause is equal to  $(15 \vee 16 \vee \neg 13)$  and the DIMACS formula is equal to `15 16 -13 0`.

#### 4.1.2.5 Must get its value from somewhere

The default value of a symbol is `n`. If it has no explicit default, no prompt and is not being `selected`, it falls back on the value `n`. Having any other value than `n` implies that it got its value from a source.

Returning to the Kconfig example and looking at the configuration option  $B$ , 10 says if its prompt is visible, 9 says if it is `selected`, 11 says if its default is satisfied, and 7 is its value. This translates to  $(7 \rightarrow (9 \vee 10 \vee 11)) \equiv (\neg 7 \vee 9 \vee 10 \vee 11)$ . In other words, to be able to set its value to `y`, one of its value sources must first be true. This constraint is created in an analogous way for  $A$  and  $C$ .

#### 4.1.2.6 Force an implicit value

This constraint does not originate from the Kconfig language. Its purpose is to make the seventh literal in Table 4.2 act as a toggle for the configuration option's value source. If the literal is set to true, the configuration option must get its value indirectly from either a default or from being selected. If it is false, it has no effect on the satisfiability.

If we look at configuration option  $B$  from the Kconfig example, its constraint is  $(B_6 \rightarrow (B_3 \vee B_5)) \equiv (12 \rightarrow (9 \vee 11)) \equiv (\neg 12 \vee 9 \vee 11)$ . Now, by setting literal 12 to true, it enforces configuration option  $B$  to obtain a value from its `default` or from being `selected`. The satisfiability solver will return `unsat` if this enforcement is not possible to make.

## 4.2 Read and set the configuration

The user's configuration is stored in the file `.config`. An excerpt of how it could look like can be seen in Listing 4.2. We can in the example see that `A` is set to `y` and

**Listing 4.2:** A `.config` excerpt.

```
1 #
2 # Linux/x86 4.4.10 Kernel Configuration
3 #
4 CONFIG_A=y
5 # CONFIG_B is not set
6 CONFIG_C=m
```

`C` is set to `m`. What is not so obvious is that `B` is explicitly set to `n`, even though it appears like the line about `CONFIG_B` is just a comment. However, it is only when a configuration option is completely absent from the configuration file that it falls back on its default value, specified in the `Kconfig` file.

We will now look at how the user's configuration is set in the SAT problem. Assume that the configuration in Listing 4.2 is for the `Kconfig` model declared in Listing 4.1, with the allocated literals in Table 4.3. From this model and configuration, we get the constraints  $(1 \wedge \neg 2)$ ,  $(\neg 7)$  and  $(13 \wedge 14)$ . These will enforce the values of the three configuration options `A`, `B` and `C`. Furthermore, we will also set the literals 3, 8 and 15 to true, which say that these configuration options got their values from the user. Combined, they translate into the following single-variable clauses in DIMACS:

```
1 0
-2 0
3 0
-7 0
8 0
13 0
14 0
15 0
```

### 4.3 Generate unsatisfiable cores with SAT

In this section, our method for extracting unsatisfiable cores in `C` with a SAT solver is presented. With a small example containing both hard and soft constraints, how the Scala implementation of `RangeFix` achieves this with an SMT solver is first examined. Next, how we achieve the same thing with SAT in `C` is presented. This highlights the contrast between the SMT and SAT syntax, and their respective capabilities.

An example of two hard and three soft constraints is depicted in Figure 4.1. A satisfiability solver will evaluate this to unsatisfiable, since  $C \rightarrow \neg B \equiv true \rightarrow \neg true \equiv true \rightarrow false \equiv false$ . To regain satisfiability, some of the soft constraints need to be edited.

The two minimal unsatisfiable cores in this example are  $\{2, 4, 5\}$  and  $\{1, 2, 3, 5\}$ . If any of the clauses that appears in a minimal unsatisfiable core is removed, the rest of the core's clauses become satisfiable. For instance, if clause 5 is removed,



- [1]  $(A \rightarrow B)$
- [2]  $(C \rightarrow \neg B)$
- [3]  $(A := true)$
- [4]  $(B := true)$
- [5]  $(C := true)$

**Figure 4.1:** Two hard constraints and three soft constraints.

**Listing 4.3:** Encoding the example in Figure 4.1 as an SMT problem.

```
1 (set-option :produce-unsat-cores true)
2 (declare-const a Bool)
3 (declare-const b Bool)
4 (declare-const c Bool)
5 (declare-const _a Bool)
6 (declare-const _b Bool)
7 (declare-const _c Bool)
8 (assert (=> a b))
9 (assert (=> c (not b)))
10 (assert (or (not _a) (= a true)))
11 (assert (or (not _b) (= b true)))
12 (assert (or (not _c) (= c true)))
13 (assert (! _a :named A))
14 (assert (! _b :named B))
15 (assert (! _c :named C))
16 (check-sat)
17 (get-unsat-core)
```

then the core  $\{2, 4, 5\}$  becomes satisfiable because  $C$  may be assigned to false.

### 4.3.1 How unsatisfiable cores are generated with SMT

The Scala implementation of the RangeFix algorithm depends on an SMT solver, more specifically the SMT solver Z3. The SMT-LIB language has a command called `get-unsat-core`, which responds with one unsatisfiable core if the previous call to `check-sat` returned `unsat`. All solvers do not support the `get-unsat-core` command [45], however, Z3 is one of those which supports it, making it suitable for the RangeFix algorithm.

The hard and soft constraints from Figure 4.1 are encoded with the commands depicted in Listing 4.3. The variables are declared using constants, while the constraints are declared with `assert` statements. Executing the SMT program prints:

```
unsat
(B C)
```

If line number 13 is deleted, it will instead print the other core:  $(A C)$ . An advantage, as illustrated in this example, is that with SMT it is easily controlled what variables are returned as parts of the unsatisfiable cores.

**Listing 4.4:** Encoding the example in Figure 4.1 as a SAT problem.

```
1 p cnf 3 5
2 -1 2 0
3 -2 -3 0
4 1 0
5 2 0
6 3 0
```

### 4.3.2 How to generate unsatisfiable cores with SAT

The propositional logical formulas in Figure 4.1 can also be expressed as the CNF clauses in Listing 4.4, encoded in the DIMACS syntax, making them compatible with SAT solvers. An example of such a SAT solver is PicoSAT [34]. By using logical equivalence [46], the *implies* connectives have been expressed with logical *or* instead. With these clauses declared, PicoSAT's function `picosat_sat` returns `PICOSAT_UNSATISFIABLE`. Having evaluated the satisfiability, iterating over the clauses and calling `picosat_coreclause` during each iteration, yields the responses `false`, `true`, `false`, `true` and `true`, i.e. the second, fourth and fifth clauses belong to the unsatisfiable core, which translates to the core  $\{2, 4, 5\}$ . If line number 5 is deleted, PicoSAT will instead return `true`, `true`, `true` and `true`, which translates to the core  $\{1, 2, 3, 5\}$ . There clearly exists no distinction between hard and soft constraints, and the unsatisfiable cores contain both constraint types. However, since it is known in what order the clauses were added, it is trivial to filter out the soft constraints from the unsatisfiable cores. By using this technique, it is possible to get the diagnoses  $\{B, C\}$  and  $\{A, C\}$ .

A code example where PicoSAT is utilized can be seen in Listing 4.5, which prints `B C` to the console. Where the soft constraints begin is stored in the variable `config_start`, which marks the spot of where to begin iterating over the clauses to identify the ones that belong to the unsatisfiable core.

**Listing 4.5:** Extracting an unsatisfiable core in C with PicoSAT.

```
1 #include <stdio.h>
2 #include "picosat.h"
3
4 #define number_of_configs 3
5
6 int main(int argc, char *argv[]) {
7     PicoSAT *ps = picosat_init();
8     picosat_enable_trace_generation(ps);
9
10    int a = picosat_inc_max_var(ps);
11    int b = picosat_inc_max_var(ps);
12    int c = picosat_inc_max_var(ps);
13    char *names[number_of_configs] = {"A", "B", "C"};
14
15    picosat_add_arg(ps, -a, b, 0);
16    picosat_add_arg(ps, -c, -b, 0);
17    int config_start = picosat_add_arg(ps, a, 0);
18    picosat_add_arg(ps, b, 0);
19    picosat_add_arg(ps, c, 0);
20
21    if (picosat_sat(ps, -1) == PICOSAT_UNSATISFIABLE) {
22        for (int i = config_start;
23            i < picosat_added_original_clauses(ps);
24            ++i)
25            if (picosat_coreclause(ps, i) == 1)
26                printf("%s ", names[i - config_start]);
27        printf("\n");
28    }
29    return 0;
30 }
```

## 4.4 Generate diagnoses

In this section, how the diagnoses are generated in the C implementation of RangeFix is examined. The algorithm works as explained in Section 2.5, however, some additional implementation details will be given here.

To start with, we will look at an example that illustrates how the diagnoses are generated by the algorithm. Assume that we have the Kconfig model in Listing 4.6 and the configuration in Listing 4.7. The soft constraints are  $\{A := \text{no}, B := \text{no}, D := \text{yes}\}$ . During the first iteration of RangeFix, the partial diagnosis is  $\{\}$  and the configuration consists of  $\{A, B, D\}$ . The core it finds is  $\{B, D\}$ , and the two partial diagnoses  $\{A\}$  and  $\{B\}$  are added. During the second iteration, it randomly selects a partial diagnosis to continue with. Assume that the partial diagnosis  $\{B\}$  gets picked; the configuration consists then of  $\{A, D\}$ . This returns the core  $\{A, D\}$ . The two partial diagnoses are now  $\{D\}$  and  $\{A, B\}$ . Next, it continues with the partial diagnosis  $\{D\}$ , which manages to satisfy the constraints and is added to the list of generated diagnoses. The remaining partial diagnosis,  $\{A, B\}$ , also manages to satisfy the constraints. The two produced diagnoses are therefore  $\{D\}$  and  $\{A, B\}$ . This makes sense, because we can either satisfy the constraints by setting  $D$  to  $n$ , or by setting  $A$  and  $B$  to  $y$ .

How the user starts with a small configuration and expands it to contain an additional configuration option is illustrated in Figure 4.3. We start with the set of all configuration options declared in the Kconfig model, named  $S$  in Figure 4.3a. The user has a configuration in her `.config` file, where she has configured a subset of all present configuration options. This is depicted in Figure 4.3b, where the configuration  $C$  is a subset of  $S$ . The configuration option the user wants to enable is called  $f$  and is depicted in Figure 4.3c.  $f$  is drawn outside the area of  $C$ , because the user has not yet assigned a value to it in her `.config` file. However, to enable  $f$ , its dependencies must also be satisfied. This means that some of the configuration options that  $f$  depends on must be edited too, illustrated by the green area in Figure 4.3d. Some of the dependencies have already been configured by the user and have to be updated, which is depicted by the intersection of the green and red

**Listing (4.6)** A Kconfig model.

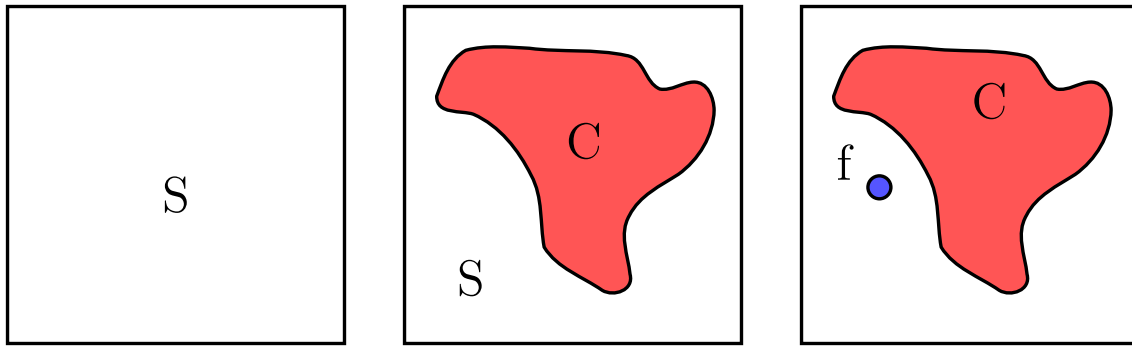
```

1 config A
2     bool "A"
3
4 config B
5     bool "B"
6     depends on A
7
8 config C
9     bool
10    default y if B
11
12 config D
13     bool "D"
14     depends on C
```

**Listing (4.7)** A configuration for the Kconfig model in Listing 4.6.

```

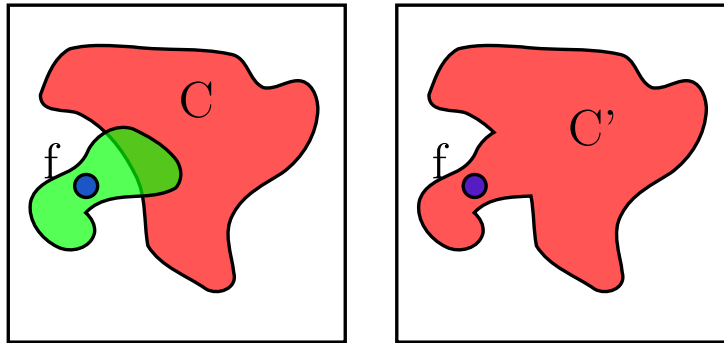
1 CONFIG_A=n
2 CONFIG_B=n
3 CONFIG_D=y
```



(a) The complete set of configuration options of the model  $S$ .

(b) The configuration  $C$  from `.config` is a subset of the whole set of declared configuration options.

(c) We want to change the value of a configuration option  $f$ .



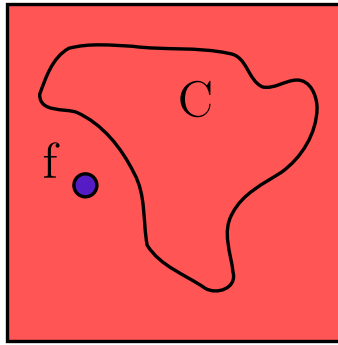
(d) A diagnosis which requires us to set new options' values and edit some of our previous configuration choices has been generated.

(e) Our new configuration  $C'$  now contains the configuration option  $f$ .

**Figure 4.3:** Visualizing how a configuration option's value is edited, a diagnosis computed and the configuration expanded.

areas. When  $f$  and its dependencies have been properly configured, we end up with a new configuration  $C'$ , shown in Figure 4.3e. Note that the new configuration  $C'$  contains a larger set of configuration options, including the configuration option  $f$ .

For this to work, we cannot only have the configuration options in  $C \cup \{f\}$  as our soft constraints, because then the unsatisfiable cores will only contain configuration options from this set. If the soft constraints do only consist of configuration options from  $C \cup \{f\}$ , i.e. the union of the red area and the blue dot in Figure 4.3c, the satisfiability solver can then freely infer values for the configuration options in  $S \setminus (C \cup \{f\})$ . The configuration options that will be found among the unsatisfiable cores are therefore only those in the green area in Figure 4.3d that overlap with  $C$ . To solve this, we include the whole set of configuration options from the feature model as soft constraints, as illustrated by Figure 4.4. The configuration options in  $S \setminus (C \cup \{f\})$  are set to their default values, which are computed by the internal



**Figure 4.4:** Visualizing how all configuration options in  $S$  are used as soft constraints.

**Listing (4.8)** A Kconfig model.

```

1 config A
2   bool "A"
3
4 config B
5   bool "B"
6   default y
7
8 config C
9   bool "C"

```

**Listing (4.9)** A configuration for the Kconfig model in Listing 4.8.

```

1 CONFIG_A=y

```

Kconfig infrastructure and read from the `rootmenu` data structure.

We will now look at an example of how the soft constraints look like. Assume that we have the Kconfig model in Listing 4.8 and the configuration in Listing 4.9. There are three configuration options, where one of them has been configured by the user in the configuration file. When the configuration is read by the internal Kconfig infrastructure, the values for all configuration options are computed. By reading the `rootmenu` data structure, we find the three configuration options' values. Even though only A has been explicitly assigned a value by the user in the configuration file, the two others' values are also easily retrieved by iterating over all configuration options using the Kconfig infrastructure's function `for_all_symbols`. The configuration, and the soft constraints, that the C implementation of RangeFix uses are then  $[A:=y, B:=y, C:=n]$ .

## 4.5 Simplify diagnoses

In the previous section, it was explained that all configuration options are used as soft constraints. It was also explained that the values used for these configuration options in the soft constraints are parsed from the configuration. This means that a diagnosis will contain all configuration options whose values deviate from the values parsed from the configuration. This includes configuration options that become implicitly `selected` or whose default value is changed.

Let us take a look at an example of how this looks like in practice. Starting

**Listing 4.10:** The definitions of the configuration options IMA and IMA\_DEFAULT\_HASH\_SHA512.

```
1 config IMA
2     bool "Integrity Measurement Architecture(IMA)"
3     select SECURITYFS
4     select CRYPTO
5     select CRYPTO_HMAC
6     select CRYPTO_MD5
7     select CRYPTO_SHA1
8     select CRYPTO_HASH_INFO
9     ...
10
11 choice
12     prompt "Default integrity hash algorithm"
13     depends on IMA
14     ...
15
16 ...
17
18 config IMA_DEFAULT_HASH_SHA512
19     bool "SHA512"
20     depends on CRYPTO_SHA512 && !IMA_TEMPLATE
21     ...
22
23 ...
24
25 endchoice
```

with an `allnoconfig` for Linux kernel 4.4.10, assume that we generate diagnoses for setting the configuration option `IMA_DEFAULT_HASH_SHA512` to `y`. A diagnosis for `IMA_DEFAULT_HASH_SHA512`, which contains all configuration options whose values are changed, is `[INTEGRITY, CRYPTO_SHA512, MULTIUSER, CRYPTO, IMA, CRYPTO_SHA1, CRYPTO_MD5, SECURITYFS, CRYPTO_HMAC, CRYPTO_MANAGER, TCG_TPM, TCG_TIS, SYSFS, SECURITY, IMA_NG_TEMPLATE]`. However, the user does not need to manually change all these configuration options by herself. The reason is that many of them are indirectly changed by other configuration options through `select` and `default` attributes. Looking at the definition of `IMA_DEFAULT_HASH_SHA512` in Listing 4.10, it is part of a `choice` group that depends on `IMA`. However, `IMA` selects many other configuration options, which therefore become part of the diagnosis. But, a better diagnosis would only contain the smallest set of configuration options that the user is required to edit. In this example, a simplified diagnosis is `[CRYPTO_SHA512, MULTIUSER, IMA, SYSFS, SECURITY]`. By only changing the values of these five configuration options, it is possible to satisfy the dependencies for `IMA_DEFAULT_HASH_SHA512`. However, as we have seen, other configuration options are also implicitly changed.

To simplify a diagnosis, our C implementation of `RangeFix` uses the seventh allocated configuration option literal, see Table 4.2 for a list of all allocated literals for each configuration option. When a diagnosis has been found, the program iterates over each configuration option in the diagnosis, sets the seventh literal to true and

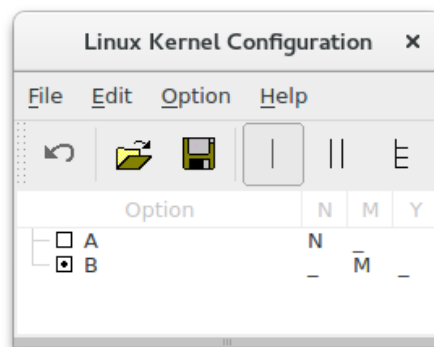
checks if the configuration is still satisfiable. If it is satisfiable, it means that the configuration option does not need to be explicitly changed by the user. But if it is not satisfiable, it means that the user must manually change this configuration option. Doing this process for each configuration option in a generated diagnosis, it is possible to remove the configuration options that the user does not need to change manually, which increases the usability of the diagnoses.

## 4.6 Integrate with xconfig

In Section 2.1, screenshots of the configurators `pure::variants` and the eCos Configuration Tool were shown. Both of them support the ability to enter an invalid state and later resolving any conflicts. To help the user resolve conflicts, they detect and show conflicts in a panel adjacent to the feature model. In the same section, it was also explained that the Kconfig configurators do not support the configuration to enter an invalid state. This is achieved by only presenting configuration alternatives that do not cause any conflicts. For instance, assume that there are two configuration options A and B, they are both tristates, and A depends on B. If B is set to `m`, the configurator does only let the user to set A to either `n` or `m`. This is shown in Figure 4.6, where "N" and "M" are the configuration options' current values, and the underscores show the valid states they may be changed to.

The Kconfig configurators are built to only allow the configuration to enter valid states, which has shaped the source code to a large extent. When building a GUI prototype with `xconfig`, it was therefore easier to leave that code as-is, and add the conflict-resolution as dependency-resolution instead. Rather than assisting the user with resolving conflicts, it would assist the user by generating fixes for satisfying unmet dependencies. This does not allow the user to enter an invalid configuration state, but does still achieve the goal of helping the user to enable disabled configuration options.

When this thesis project started, the existing Scala implementation of RangeFix had not seen major updates for the last few years. This meant that the latest version of the Linux kernel that it had been tested with was 2.6.32. LVAT, which the Scala implementation of RangeFix depends on, had also not been touched for many years.



**Figure 4.6:** Configuration option A depends on B, and is prohibited by the configurator to be set to `y`.



For instance, `exconfig`, which utilizes the internal `Kconfig` infrastructure to produce an `.exconfig` file that is used as input to `RangeFix`, did not work with version 4.4 of the Linux kernel, which was the latest kernel version at this project's start. To demonstrate and evaluate the concept of interactive dependency-resolution for `Kconfig`, implementing it for the deprecated kernel version 2.6.32 had a lower barrier.

For evaluating the feasibility of implementing a SAT-based conflict-resolution mechanism in C, there was no reason to use the deprecated 2.6.32 version of the Linux kernel. The latest version of `Satconfig`, which our C implementation of `RangeFix` utilizes, was based on version 4.4 of the Linux kernel. It was therefore easiest, and it felt more relevant, to use a modern and supported kernel version for evaluating the feasibility. The comparability might have been slightly better if the C implementation of `RangeFix` had been implemented for version 2.6.32 of the Linux kernel, as the Scala version. However, version 2.6.32 was first released in 2009 and has since then reached end-of-life.

# 5

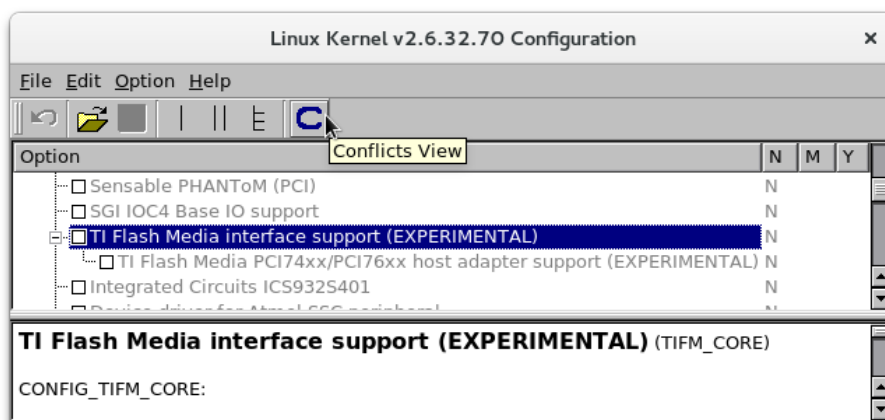
## Demonstration

Two artifacts have been produced. They are both versions of xconfig, but with different backends for the dependency-resolution support. The first uses the Scala implementation of RangeFix as its backend, while the second uses the C implementation of RangeFix as its backend.

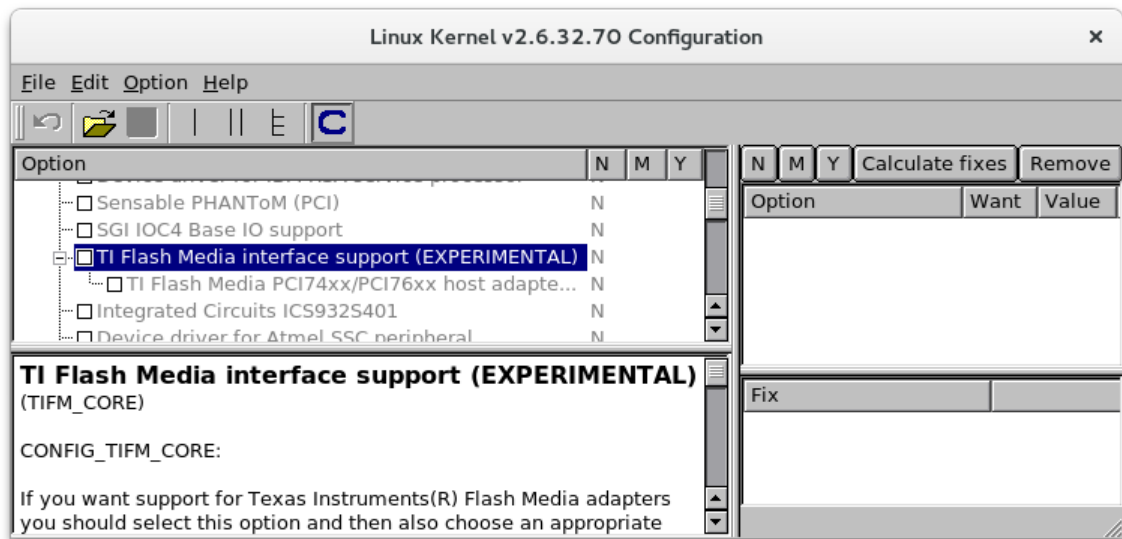
### 5.1 Configurator with Scala backend

In Figure 5.1, a screenshot of the modified version of xconfig that calls the Scala implementation of RangeFix can be seen. With its additional functionality, it is possible for the user to get assistance in resolving configuration options' dependencies. In the screenshot, the configuration option TIFM\_CORE, whose prompt is called "TI Flash Media interface support", has been marked by the user. It is currently set to no, indicated by the N in the column "N" to the right of it. Assume that the user wants to change the value of the configuration option to yes, but does not know how; by pressing the button in the toolbar that looks like a blue C, a panel with functionality for calculating fixes will be opened.

In Figure 5.2, the panel for assisting the user with calculating fixes has been opened. It consists of a toolbar, two lists and a status bar. The buttons in the toolbar let the user perform various actions, and will be explained in further detail in the following paragraphs. The upper list is for containing configuration options that the user wants to configure, while the lower list is for presenting generated fixes.



**Figure 5.1:** A screenshot of the Linux configuration tool xconfig, enhanced with interactive dependency-resolution hidden behind a toolbar button.



**Figure 5.2:** The functionality to support interactive dependency-resolution has been opened in a separate panel of the window.

Lastly, the status bar provides the user with feedback, such as telling if RangeFix is running in the background.

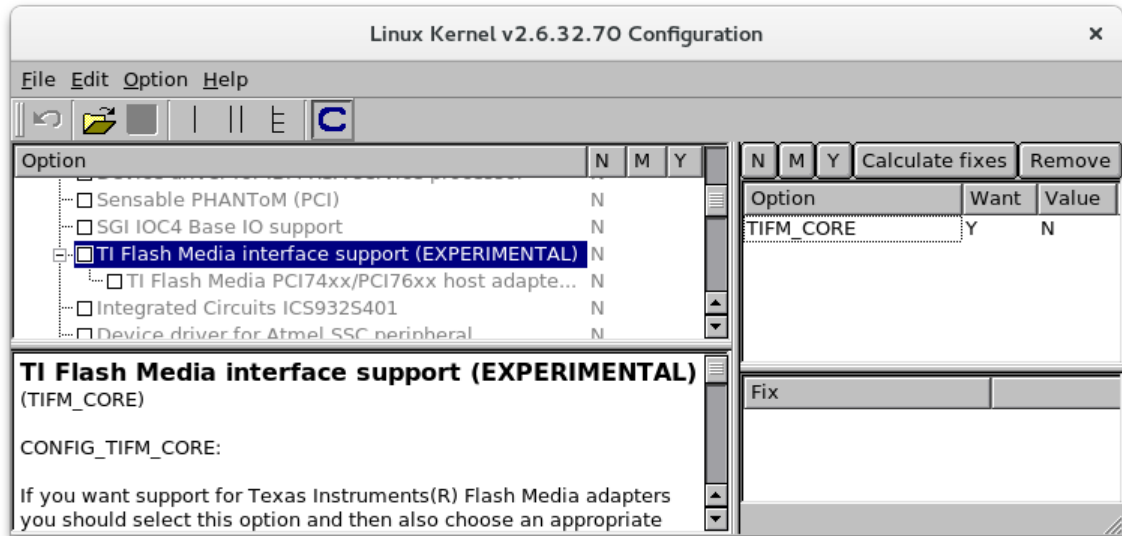
With the buttons "N", "M" and "Y", the user is able to say what value she wants for the marked configuration option in the left-hand panel. Assume that the user wants to change the value of the configuration option `TIFM_CORE` from `N` to `Y`. It is currently not possible due to unmet dependencies, indicated by the absence of an underscore in the column "Y" to the right of the marked configuration option. But by pressing the button "Y" in the toolbar to the right, the configuration option will be copied to the upper list of the right panel. In Figure 5.3, it is shown how it looks like after the user has pressed the button "Y". `TIFM_CORE` has been copied to the list of configuration options that the user wants to configure, its current value is `N`, and the value the user wants it to have is `Y`.

By selecting the configuration option in the upper-right list and pressing the "Calculate fixes" button, RangeFix will be initiated. After a while, when the commutation is completed, the fixes are returned and presented in the lower-right list. In Figure 5.4, fixes for the configuration option `TIFM_CORE` have been computed. Three fixes were found in this case, two being visible in the screenshot. Any one of these fixes is a valid way for resolving the dependencies and setting `TIFM_CORE` to `Y`.

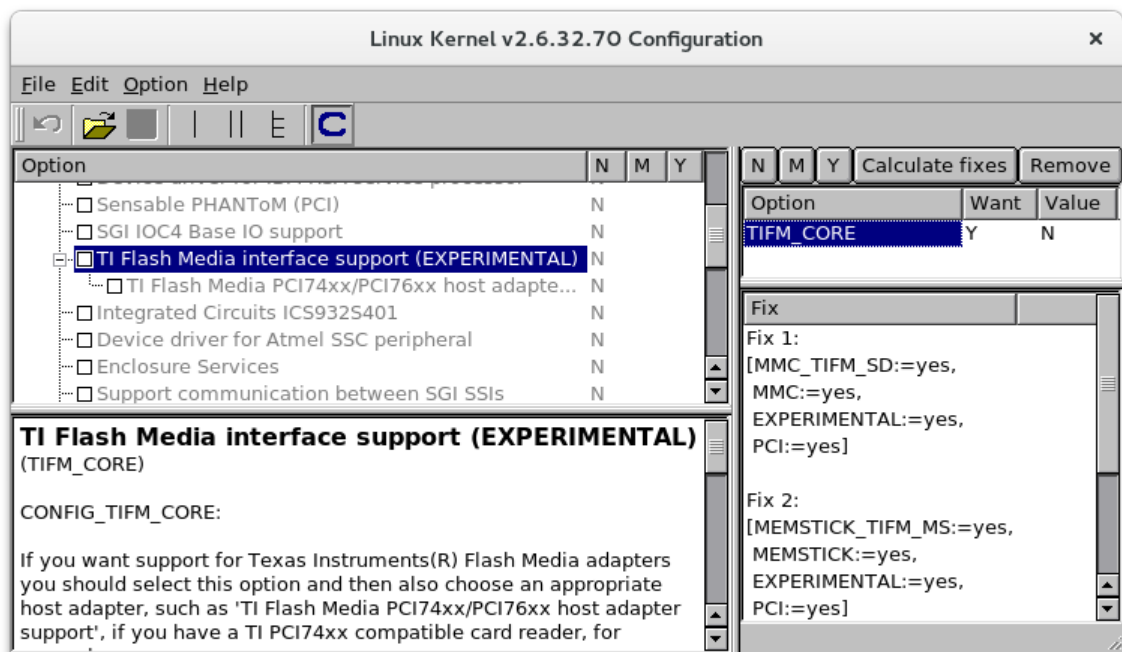
This interface allows the user to first locate the configuration options she wants to configure, and afterwards deal with their dependencies. An example where the user has added three configuration options to the upper-right list is shown in Figure 5.5. The user has marked two of those configuration options, and calculated fixes for them both at the same time. The visible fix is much larger than the previous two fixes in Figure 5.4, but it will on the other hand result in both `TIFM_CORE` and `BT_HCIBLUECARD` being set to yes if applied.

The "Remove" button in the toolbar deletes marked configuration options from the upper-right list. It is useful after a session where the user has been wanting to edit some configuration options, computed fixes for them, applied the fixes, and is

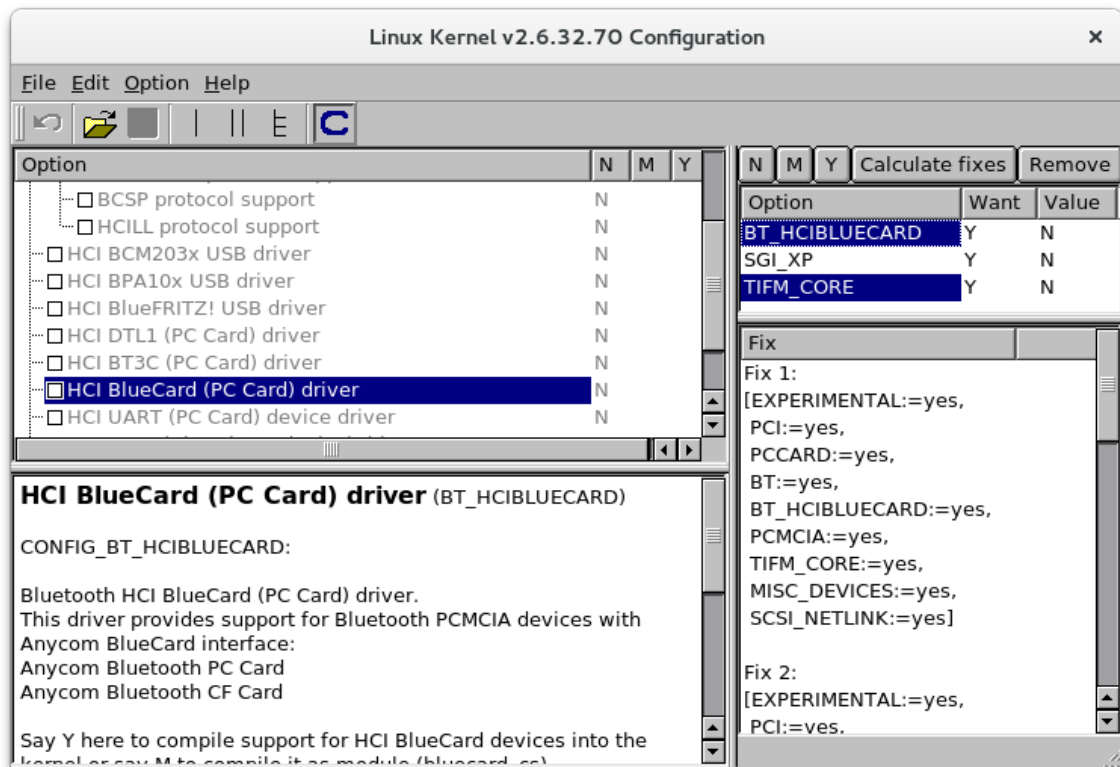
## 5. Demonstration



**Figure 5.3:** The user has added TIFM\_CORE to the list of configuration options she wants to configure.



**Figure 5.4:** Fixes for the disabled configuration option TIFM\_CORE have been generated.



**Figure 5.5:** Fixes for the both configuration options `TIFM_CORE` and `BT_HCIBLUECARD` have been generated.

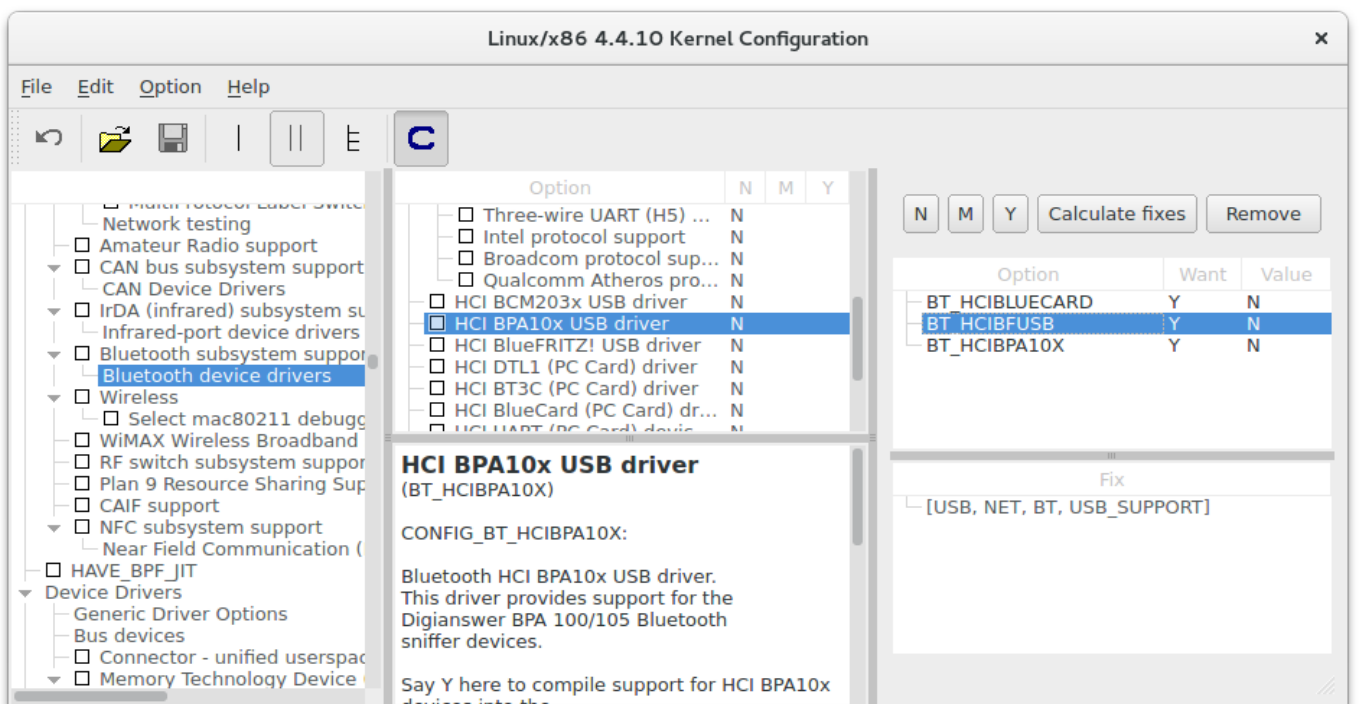
now done with configuring them.

For a more thorough demonstration of the workflow that this configurator enables, a video that showcases it has been recorded [47].

## 5.2 Configurator with C backend

In Figure 5.6, a screenshot of the modified version of `xconfig` that uses the partial C implementation of `RangeFix` as its backend for dependency-resolution can be seen. Overall, it works in the same way as the other version. However, the style of the widgets look a bit different, because this version of `xconfig` is compiled against Qt 4.8 or 5.x, while `xconfig` for version 2.6.32 of the kernel is compiled against Qt 3.

By clicking the button in the toolbar that resembles the letter C, the dependency-resolution panel in the right side of the window is opened. By selecting a configuration option and clicking one of the "N", "M" and "Y" buttons, the configuration option is added to the list of configuration options in the middle-right part of the window. These are the configuration options that the user wants to configure. Selecting one of the saved configuration options in that list and clicking the "Calculate fixes" button, triggers the C implementation of `RangeFix`. When `RangeFix` returns, the generated diagnoses are listed in the bottom-right corner of the window.



**Figure 5.6:** A screenshot of the Linux 4.4.10 configuration tool xconfig, enhanced with interactive conflict-resolution.

# 6

## Evaluation

The aim of this chapter is to evaluate this thesis' proposed solution. We started the thesis by making the case for why configuring the Linux kernel is in need of improvement, see Section 1.1. To resolve these issues, we found RangeFix, which integrated with one of the Linux kernel configurators would be able to aid the user in resolving dependencies. To demonstrate its capabilities, we designed and developed two versions of xconfig that both rely on the RangeFix algorithm. One uses the existing Scala implementation of RangeFix, while the second uses our preliminary reimplementaion of RangeFix in C. With these two artifacts, we want to evaluate 1. the users' opinions on such a mechanism as a solution to the configuration problem, 2. how well the existing Scala implementation of RangeFix performs, and 3. if it is feasible to achieve the same correctness and performance in C with a SAT solver.

The two artifacts have been evaluated in multiple ways. The modified version of xconfig that calls the Scala implementation of RangeFix has been shown to Linux users. Through a survey, they have provided us with their opinions and feedback. By getting the users' perspective, it is possible to discover if this is a solution to the problem that is worthwhile to explore further in future projects. Both implementations of RangeFix have also been evaluated in terms of correctness and performance. By evaluating the existing Scala implementation, we can asses its viability as a dependency-resolution mechanism in xconfig. We also get a reference point to compare the C implementation against. By evaluating our partial C implementation, we are able to find out the possibility of achieving a reimplementaion of the algorithm in C, which would have a greater acceptance among the kernel developers. Together with the survey results, we are also able to determine whether the two implementations produce satisfying results with regards to the users' expectations.

In this chapter, we will first present the user survey. It is followed by the evaluation of the existing Scala implementation. Next, the evaluation of our partial C implementation is presented. Observations and conclusions are then made from these data points. Lastly, threats to validity are also discussed.

### 6.1 User survey

In this section, our survey conducted with people familiar with configuring the Linux kernel is presented. The aim of the survey was to contribute to the evaluation of the configurator demonstrated in Section 5.1. Since the purpose of the artifact is to assist the users in their tasks, it is important to also get those users' assessment, which this survey is meant to do.

### 6.1.1 Survey design

**Survey goal.** The goal of the survey was to evaluate this case study’s direction. That encompassed the usability and functionality offered by xconfig with fix generation from the Scala implementation of RangeFix, demonstrated in Section 5.1. But also how well interactive fix generation has to function to be an improvement over the method they currently employ to resolve unmet dependencies.

**Survey questions.** The questions were designed to answer questions within several areas. To start with, questions to establish if the participants encounter issues while configuring the kernel were asked. This data would be compared to the data gathered by the survey by Hubaux et al. [6], presented in Section 1.1. The next area was if they consider the help text to configuration options to be useful. The help text, which was shown in Section 2.2, is the only aid the user is currently given in xconfig to configure the kernel. It is therefore interesting to evaluate its helpfulness. Next, the participants were asked how they resolve dependencies using the currently available tools. This was asked to identify usage patterns, but also to identify what workflows this thesis’ artifacts compete against. The next area was designed to establish what time it takes for the participants to resolve a configuration option’s dependencies. This is of interest because it establishes a baseline that an interactive conflict-resolution tool would need to be at least as fast as. In the last area, the appropriateness of the workflow supported by this thesis’ demonstrated artifacts was asked about. This would provide us with some firm feedback to evaluate the prototype and identify future work. The complete survey with all its questions can be found in Appendix A.

**Participants.** Linux users and developers being familiar with configuring the kernel were asked to participate in the survey. The survey was posted on the kconfig-sat project’s mailing list, the Linux kernel’s mailing list, and the discussion boards linuxquestions.org, bbs.archlinux.org and forums.gentoo.org. Eleven people participated in the survey.

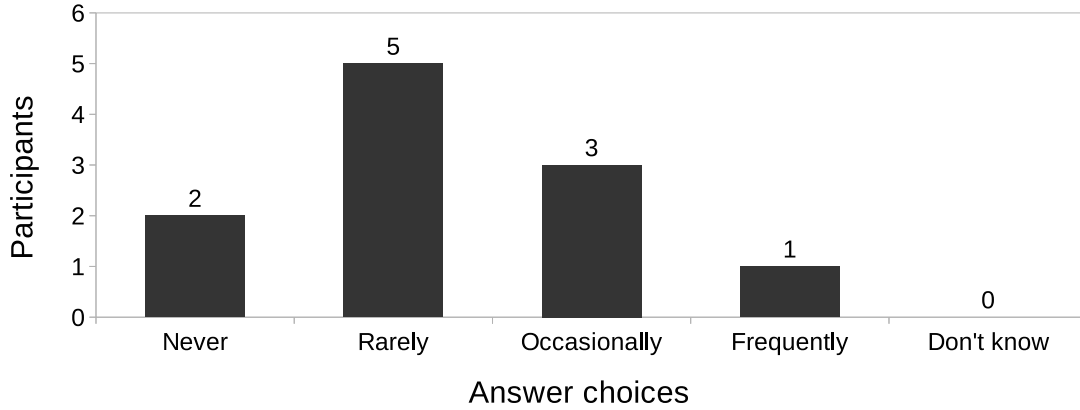
### 6.1.2 Survey results

The participants were first asked how often they run into issues when trying to change a configuration option’s value in one of the Linux kernel configurators. The exact question, answer choices, and participants’ answers can be seen in Figure 6.1. Seven of the participants did also elaborate on their answer in a free-form text field. Many of the answers were interesting and are provided in Table 6.1. It is apparent from these answers that there are occasions when changing a configuration option’s value is not a straightforward process. It might involve finding out what forces its lower bound, finding its unmet dependencies in the menus, and parsing its dependency expression. Furthermore, P2 mentioned that she finds it annoying that disabling a configuration option also disables and hides the configuration options that depend on it.

Each configuration option has a help text, which describes what it does and how to enable it. Participants were asked to answer whether they find this text helpful



Do you ever run into issues when trying to change a configuration option's value in xconfig/menuconfig during the Linux kernel configuration process?

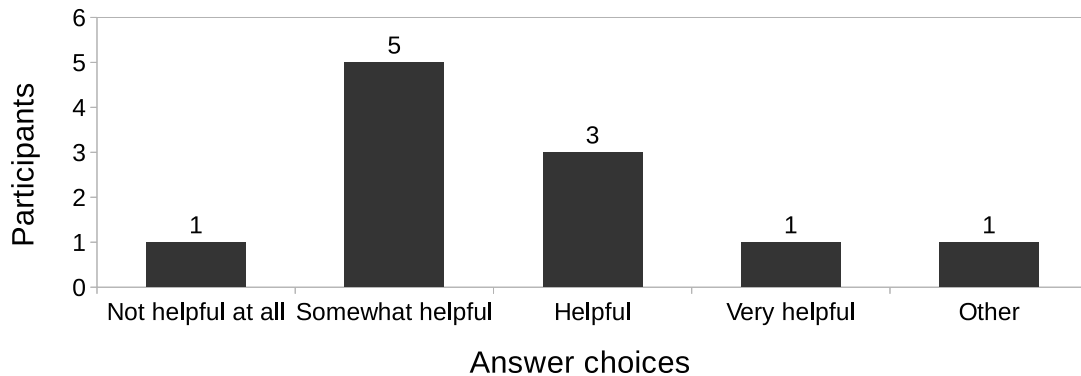


**Figure 6.1:** If the participants ever run into issues when changing a configuration option.

**Table 6.1:** Several participants explained their answer in Figure 6.1 in further detail.

"I frequently find that I can't find an option I want to enable/disable, because it's hidden by another option being set/unset. I also frequently find that I can't turn an option off because another option selects it. And finding the appropriate option typically requires carefully reading the full expressions involved, then searching for the relevant option by name, and then finding that option through the menu to change it." (P1)
"Other options depending on the disabled one are suddenly also disabled" (P2)
"Sometimes a new option isn't explained well enough for me to quickly determine if I want it or not." (P3)
"Sometimes I search for an option, find it, but cannot enable it because it depends on something that is disabled. Then I search for the dependencies, enable them and finally enable the option I wanted." (P4)
"Although rare, I do on occasion miss including an option that turns out needed." (P5)
"Only when I can't find the name of a driver or it has unmet dependencies" (P7)
"Finding the correct option in the menu" (P8)

How much help does the configuration option's help text provide when trying to change the current configuration to satisfy any missing dependencies needed to enable a particular option?



**Figure 6.2:** Participants' ratings of the helpfulness of the configuration options' help text.

when trying to satisfy any missing dependencies needed to enable a particular option. Their responses are shown in Figure 6.2. P4 picked the answer choice "Other" and left the response: "It lists the dependencies so I can find them. Could be better."

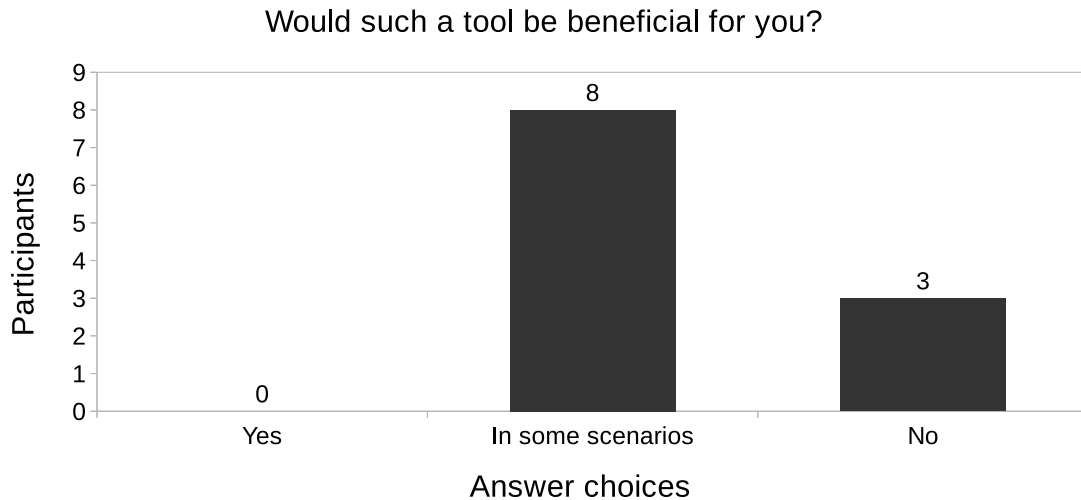
We next asked the participants about what method or tool they employ when trying to satisfy a configuration option's dependencies. Five participants replied that they use `menuconfig` to configure the kernel. To enable a configuration option, they read the option's help text within `menuconfig` and then use the built-in search functionality to locate its dependencies. P1 also explained that she sometimes resorts to reading the `Kconfig` files for figuring out the dependencies:

"[Manually I read] the dependencies in `menuconfig`'s help or by looking at the `Kconfig` file directly, search for the options that aren't enabled. `menuconfig` helps a bit there by noting the states of options, but complex expressions still require careful staring and manual evaluation." (P1)

P9 added that she uses the terminal command `sed` to manipulate configuration options' values directly in the `.config` file if she already knows the module's name. P5 explained that she finds dealing with mutual exclusive configuration options to be a challenge during this process:

"Dependencies are rarely an issue since Help helps by listing most. [...] The greatest problem can be conflicts ie- nVidia proprietary vs/ Rivafb and the like. Those are less well documented within the kernel Help sections." (P9)

The participants were then asked to indicate the longest, shortest, and typical time it takes for them to change a configuration option's value. The typical time among the participants was on average 101 seconds, ranging between 30 seconds and



**Figure 6.3:** If the participants think xconfig with fix generation would be beneficial.

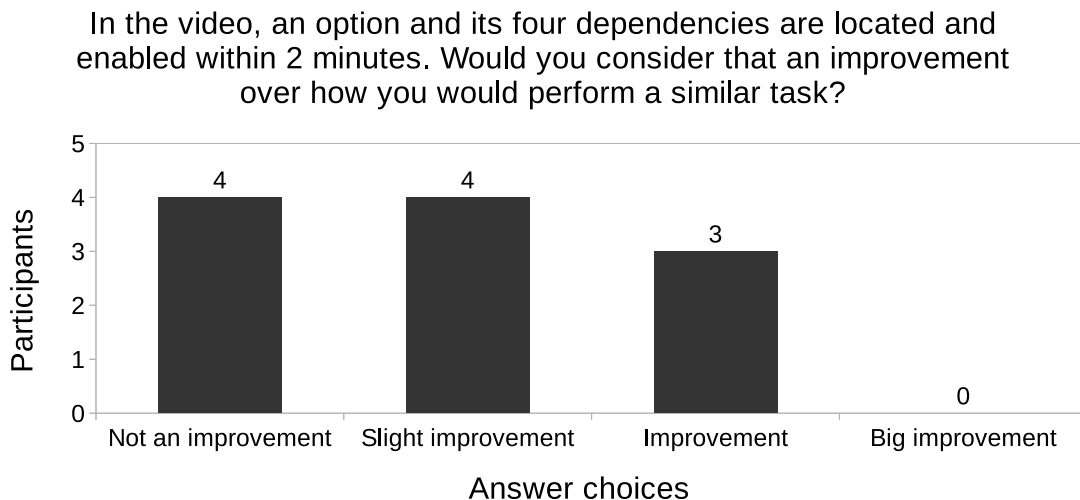
7 minutes. The shortest time was on average 20 seconds, ranging between 1 second and 2 minutes. The longest time was on average 11 minutes, ranging between 2 minutes and 1 hour, with most answers being between 2 and 10 minutes.

A video of the modified version of xconfig that calls the Scala implementation of RangeFix for generating fixes was then shown to the participants, followed by questions about it. They were first asked whether they would consider such a tool to be beneficial. The answers to the question are shown in Figure 6.3. The ability to expand the answer in a free-form text field was also given. Most participants wrote that it would be useful when attempting to edit either a configuration option whose dependencies have not been met or when editing a lot of configuration options, for instance when moving between major releases such as Linux 3.x and Linux 4.x. However, a couple of participants explained that the search functionality in menuconfig is already enough for them, and one participant asked whether this would be added to menuconfig too (as opposed to only xconfig).

They were then asked what they consider to be an ideal and maximum computation time they would wait for fixes to be generated. Most participants responded that they would prefer the fixes to be generated as soon as possible, within a few seconds or within a minute. The maximum amount of time some would tolerate was 2 seconds, 3 seconds, 5 seconds, or less than 10 seconds. Some could tolerate a longer computation time and said 3 minutes, 3-4 minutes, or 15 minutes. One person replied 5-10 seconds, but also added that she would be willing to wait longer and still save time in many scenarios.

The participants were then asked whether they consider the workflow demonstrated in the video to be an improvement over how they would achieve a similar task today. The responses are shown in Figure 6.4. 36% do not consider it to be an improvement, while 64% consider it to be an improvement of varying degree.

Lastly in the survey, it was possible to provide feedback about our prototype in a free-form text field. P1 asked why the fixes have to be applied manually, and thought it would be beneficial to have an "apply" button to automatically apply a fix. P3 said



**Figure 6.4:** If the participants think the supported fix generation workflow would be an improvement over how they would perform a similar task.

she would like a version of `make oldconfig`, but that takes a configuration option and outputs the sequence of the dependent configuration option changes needed to enable it. There is a Linux program called `lspci`, which prints detailed information about all PCI buses and devices in the computer; P7 said that she would like a tie-in with `lspci` that suggests configuration options that should be enabled based on the computer’s hardware.

## 6.2 The Scala implementation

RangeFix is an algorithm whose purpose it is to generate fixes that resolve configuration conflicts. In Section 5.1, it was demonstrated how the existing Scala implementation of RangeFix, which supports Kconfig, can assist the user with resolving unmet dependencies. With this functionality in place, it is therefore relevant to evaluate the quality of the fixes. In this section, we evaluate the quality in terms of correctness and performance. Whether a fix accurately tells the user how to enable a configuration option determines its correctness. The time it takes to run the Scala implementation of RangeFix to generate fixes determines its performance.

### 6.2.1 Evaluation design

In this subsection, the evaluation design for the Scala implementation of RangeFix is explained. First, the procedure for the evaluation is presented. Next, how the fixes were classified in terms of correctness is presented.

#### 6.2.1.1 Procedure

To evaluate the Scala implementation of RangeFix, a set of 200 disabled configuration options were sampled. The selection of configuration options was done randomly

from the set of configuration options fulfilling the following criteria:

- The configuration option's type is tristate. Required since it is the only input type that the existing Scala implementation of RangeFix supports.
- The configuration option has a prompt. Required since these are the only configuration options that the user can modify directly (invisible configuration options cannot be changed directly).
- The configuration option is currently set to `no`. Required since this state says that the configuration option is currently disabled.

The initial configuration was an `allnoconfig` with version 2.6.32.70 of the Linux kernel. That kernel version was selected since 2.6.32 was the last version that the Scala implementation of RangeFix had been tested with, as explained in Section 4.6. The total number of configuration options fulfilling the above criteria on the x86 platform with an `allnoconfig` is 3,416.

For each randomly selected configuration option, the existing Scala implementation of RangeFix was executed to calculate fixes for how to set it to `yes`. The running time was recorded, and the returned fixes were manually tested in `xconfig` to see if they were correct. It was observed that there were cases when the program got stuck and ran for a very long time. The program was therefore terminated if it did not finish within five minutes, which seemed like a reasonable upper limit.

### 6.2.1.2 Correctness classifications

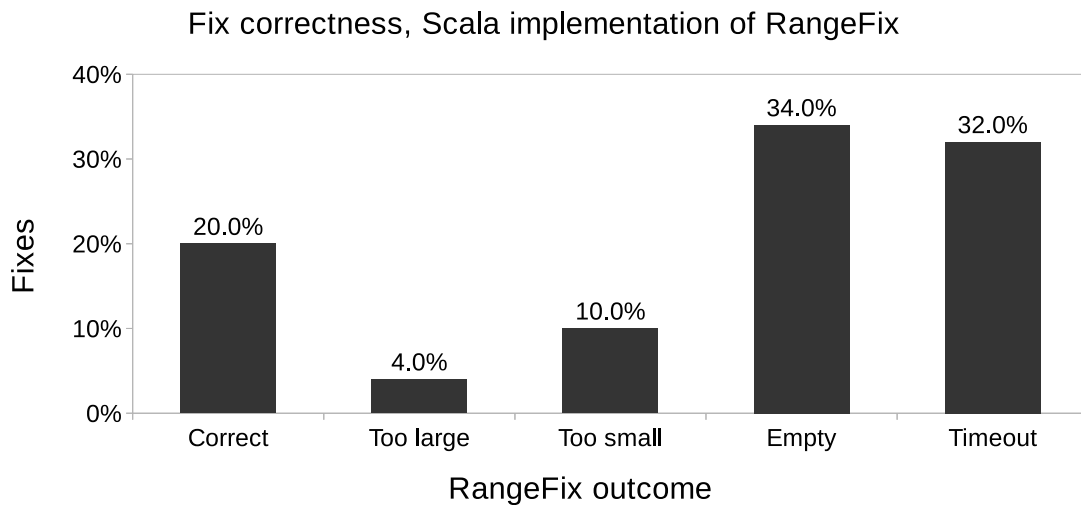
The correctness of a generated fix was determined by testing it in `xconfig`. We opted for classifying the generated fixes with five labels: `correct`, `too large`, `too small`, `empty`, and `timeout`. For each of the 200 configuration options, the generated fixes were classified with one of these labels. The classifications are explained in further detail below.

**Correct.** If a generated fix was minimal and satisfied the dependencies of the configuration option it was generated for, it was classified as correct. By being minimal, the fix does not contain any redundant assignments that the user does not have to make. Being minimal is a wanted property of a fix [30]. To satisfy the unmet dependencies is required for the fix to serve its purpose.

**Too large.** A too large fix was able to satisfy all dependencies of the configuration option, but contained redundant assignments. This means that the fix worked, since it was possible to enable the configuration option when the fix had been applied. But the fix lacked in usability since it was not minimal.

**Too small.** If a generated fix was too small, it meant that the fix did not contain the complete set of required assignments. In other words, the configuration option that the fix was generated for did still have unmet dependencies after the fix had been applied.

**Empty.** If the program returned, but with an empty set of assignments, and the configuration option had unmet dependencies, the result was classified as being



**Figure 6.5:** The results from calculating fixes for 200 disabled options.

empty. The distinction from being too small was made because the error probably originated from a different source.

**Timeout.** If the program did not return within five minutes, it was classified with the label timeout. This upper limit was implemented to ensure that the complete evaluation did not take too long time to run. Furthermore, five minutes felt generous and is probably a lot longer than a user would be willing to wait.

## 6.2.2 Correctness results

In this subsection, the correctness of the fixes, generated from the 200 runs, is evaluated. The results are summarized in Figure 6.5. These outcomes are discussed below, and various error sources are identified and explained.

### 6.2.2.1 Correct

In 20.0% of the cases, the generated fixes were correct. For instance, generating fixes for the configuration option `DRM_RADEON`, yielded the fix `[DRM_RADEON:=yes, DRM:=yes, PCI:=yes]`. This fix is possible to apply in `xconfig`, it is minimal, and successfully sets `DRM_RADEON` to `yes`.

However, one usability issue was found, caused by how `xconfig` is designed and a choice group no having a configuration option name. It happened when applying a fix for enabling `FB_HGA`. One of the four fixes contained the assignment `DRM_I915:=yes`, which is not possible to perform in `xconfig`, even though it has a prompt and the other assignments in the fix have already been applied. A screenshot of the configuration option `DRM_I915` in `xconfig` can be seen in Figure 6.6. The reason why it cannot be enabled is because it is part of a choice group, and its parent "Intel 830M, 845G, 852GM, 855GM, 865G" must first be enabled. However, as we can see in Listing 6.1, a `choice` has no symbol name, which means that it is

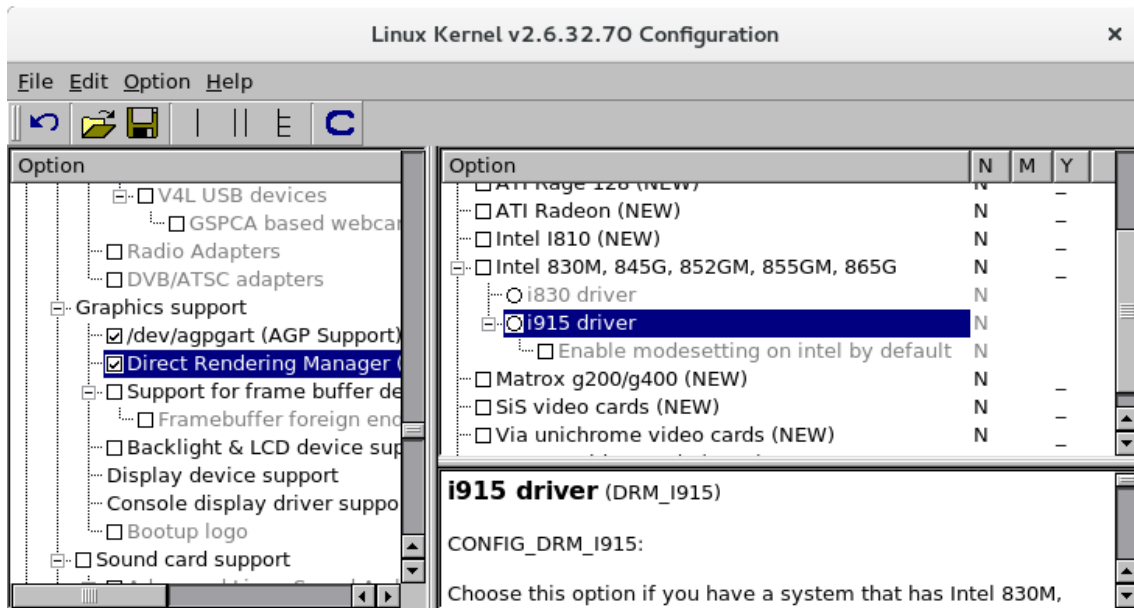


Figure 6.6: The option DRM\_I915 in xconfig.

not possible to refer to it in a fix. However, had the fix been applied automatically by the configurator, it would have worked without any issues.

### 6.2.2.2 Too large

In 4.0% of the cases, the fixes were correct but had usability issues where they contained redundant assignments. The redundancies were both in terms of duplicated fixes and unnecessary value assignments to configuration options, as explained in the following two paragraphs.

A case of duplicated fixes occurred with the configuration option IPDDP, where two of its fixes were identical. The returned fixes are shown in Listing 6.2, where we can see that the second and the fourth fixes contain the same assignments to the configuration options ATALK, IPDDP, DEV\_APPLETALK and NET. A theory to why this happened, could be that the diagnoses were originally different, but simplifying the constraints into fixes yielded the same results.

Another redundancy issue is a fix that contains an unnecessary assignment. In these cases, the assignment was superfluous because the configuration option would already get the same value from one of its `default` attributes. For instance, the fix returned for the configuration option SND\_MTPAV is equal to `[SND_DRIVERS:=yes, SND_MTPAV:=yes, SOUND:=yes, SND:=yes]`. Explicitly setting SND\_DRIVERS to `yes` is unnecessary due to its definition, which can be seen in Listing 6.3. SND\_DRIVERS is set by default to `yes` when its dependency on SND is met. Since setting SND to `yes` is part of the fix, it means that the user does not need to manually change that configuration option's value. Superfluous assignments were also found in the fixes for SND\_ICE1712, RTC\_DRV\_DS3234, RTC\_DRV\_PCAP and SND\_MTPAV.

**Listing 6.1:** The Kconfig definitions of DRM\_I830, DRM\_I915 and DRM\_I915\_KMS.

```
1 choice
2   prompt "Intel 830M, 845G, 852GM, 855GM, 865G"
3   depends on DRM && AGP && AGP_INTEL
4   optional
5
6   config DRM_I830
7     tristate "i830 driver"
8     ...
9
10  config DRM_I915
11    tristate "i915 driver"
12    depends on AGP_INTEL
13    ...
14
15  config DRM_I915_KMS
16    bool "Enable modesetting on intel by default"
17    depends on DRM_I915
18    ...
19
20 endchoice
```

**Listing 6.2:** Five fixes for setting IPDDP to yes.

```
1 [ATALK:=yes, IPDDP:=yes, DEV_APPLETALK:=yes,
2   SCSI_FC_ATTRS:!(SCSI_FC_ATTRS == no), ATA:!(ATA == no)]
3 [ATALK:=yes, IPDDP:=yes, DEV_APPLETALK:=yes, NET:=yes]
4 [ATALK:=yes, IPDDP:=yes, DEV_APPLETALK:=yes,
5   SCSI_FC_ATTRS:!(SCSI_FC_ATTRS == no), SCSI:!(SCSI == no)]
6 [ATALK:=yes, IPDDP:=yes, DEV_APPLETALK:=yes, NET:=yes]
7 [ATALK:=yes, IPDDP:=yes, DEV_APPLETALK:=yes, SCSI_NETLINK:=yes]
```

**Listing 6.3:** The Kconfig definition of SND\_DRIVERS.

```
1 if SND
2   ...
3
4   menuconfig SND_DRIVERS
5     bool "Generic sound devices"
6     default y
7     help
8       Support for generic sound devices.
9
10  ...
11 endif
```



**Listing 6.4:** The Kconfig definitions of 64BIT and X86\_32.

```

1 config 64BIT
2     bool "64-bit kernel" if ARCH = "x86"
3     default ARCH = "x86_64"
4     ---help---
5         Say yes to build a 64-bit kernel - formerly known as x86_64
6         Say no to build a 32-bit kernel - formerly known as i386
7
8 config X86_32
9     def_bool !64BIT

```

### 6.2.2.3 Too small

In 10.0% of the cases, the fixes missed assignments to satisfy the configuration option's dependencies. It was primarily caused by the implementation not taking into consideration if a configuration option is hidden or not. But in one case it occurred because a `default` attribute was not properly handled when generating the fix.

In many cases, configuration options without a prompt were included in the fixes, i.e. configuration options the user has no ability to change directly. For instance, wanting to set the configuration option `BATTERY_OLPC` to `yes` yields 18 fixes, all of which contain the fix unit assignment `X86_32:=yes`. However, the configuration option `X86_32` has no prompt, as we can see in Listing 6.4 (`def_bool` is a shorthand for declaring it as `bool` and setting its default value), and it is therefore not possible to apply any of the 18 fixes correctly. The same is true for `USB_SERIAL_MOS7720`, where its fixes contain `USB_ARCH_HAS_HCD`; `IPDDP`, where one of its five fixes contains `SCSI_NETLINK`; `PATA_WINBOND_VLB`, where three fixes contain `ISA`; and `I2C_PARPORT_LIGHT`, where one of its 16 fixes contains `FB_DDC`. To solve this, the program would need to take into account if a configuration option has a prompt or not, and only include those that have prompt in the fixes.

To explain this further, we will look at an example. In Listing 6.5, there are three configuration options: A, B and C. A depends on B, but B has no prompt. B is modified indirectly by changing the value of C, which selects B. Setting C to `yes` in `xconfig`, enables the ability to set A to `yes`. Saving this configuration in `xconfig` yields the following `.config` file:

```

CONFIG_A=y
CONFIG_B=y
CONFIG_C=y

```

Even though B has no prompt and cannot be assigned a value directly in `xconfig`, its indirect value is saved when the configuration is written to disk. Assume that we modify the `.config` directly to:

```

CONFIG_A=y
CONFIG_B=y

```

When this configuration is opened in `xconfig`, the configuration option A is turned off. Without doing any alterations in `xconfig`, saving it yields the following configuration:

```
# CONFIG_C is not set
```

This illustrates that values for invisible configuration options are written to the

**Listing 6.5:** A Kconfig model with three configuration options.

```

1  config A
2      tristate "A"
3      depends on B
4
5  config B
6      tristate
7
8  config C
9      tristate "C"
10     select B

```

configuration file. However, assigning values to invisible configuration options does not work in the configurator, nor via the configuration file.

The second found issue causing too small fixes, was that a fix would enable a conflicting configuration option through a `default` attribute. Generating fixes for the configuration option `ECHO` yielded the fix `[STAGING:=yes, ECHO:=yes]`. When enabling `STAGING` in `xconfig`, the configuration option `STAGING_EXCLUDE_BUILD` also gets enabled due to its `default y if STAGING` attribute. However, `ECHO` depends on the expression `STAGING && !STAGING_EXCLUDE_BUILD`, which means that the user does also need turn off `STAGING_EXCLUDE_BUILD`. The correct fix would therefore have been `[STAGING:=yes, ECHO:=yes, STAGING_EXCLUDE_BUILD:=no]`.

#### 6.2.2.4 Empty

In 34.0% of the cases, RangeFix returned without being able to find any fixes to the problem. Primarily, it was caused by architecture-specific dependencies being used in the configuration options shared between the architectures. But behaviour where diagnoses would only be found occasionally was also observed.

In many cases nothing was returned because of a dependency on a configuration option from another architecture. In the Linux kernel's source tree, there is a directory called `/arch` that contains a subdirectory for each supported architecture. These subdirectories contain architecture-specific configuration options that are only loaded depending on the target architecture. For instance, when configuring the Linux kernel for `x86`, the configuration options from the file `/arch/x86/Kconfig` are loaded and presented in `xconfig`. There are also directories with shared configuration options, for instance in the directory `/drivers`. However, these might still depend on architecture-specific configuration options.

For instance, when fixes were generated for the configuration `MTD_NAND_ATMEL`, nothing was returned. Taking a look at its Kconfig definition, we find that it depends on `ARCH_AT91 || AVR32`. These configuration options are only present if the user is configuring the kernel for either the architecture `ARM` or `AVR32`.

This is therefore not strictly an error. However, it is still a usability problem if there are configuration options where the program does not return anything. To solve this, the division between architecture-specific and architecture-independent configuration options would need to be stricter. Alternatively, RangeFix would need to detect if any of the conflicting configuration options contain an unavailable

architecture-specific dependency.

Sometimes no diagnoses were found. An example when this happened was for the configuration option `PCMCIA_3C589`. However, running RangeFix multiple times for the configuration option `PCMCIA_3C589`, it was found that it sometimes was able to find diagnoses. While sometimes it quit after only the first stage of the algorithm. This might be due to the order unsatisfiable cores are found in, or due to the order partial diagnoses are picked. Another possibility is that there is a bug somewhere in the code.

### 6.2.2.5 Timeout

The existing implementation of RangeFix taking more than five minutes to run happened in 32.0% of the cases. It was either caused by the diagnoses generation never completing, or by the fix generation not completing due to a too large constraint set.

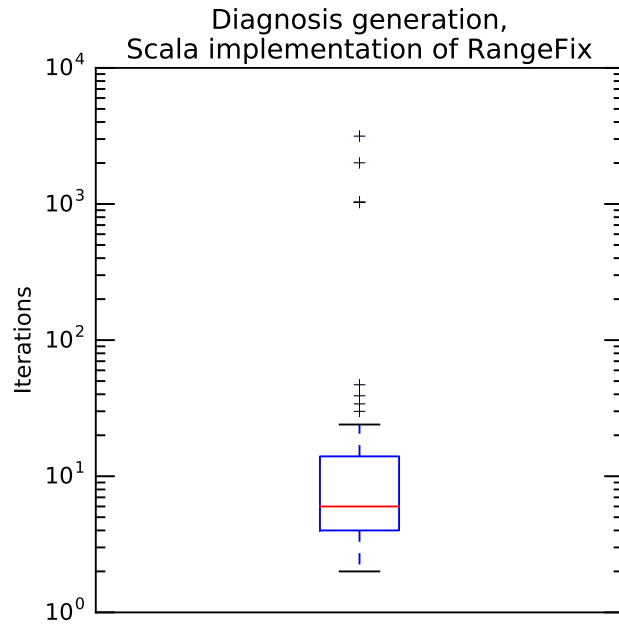
Sometimes the program timed out due to the diagnoses generation taking a very long time. For instance, when diagnoses for `NETFILTER_XT_MATCH_MARK` were computed, RangeFix never managed to find a diagnosis that was large enough to make the model satisfiable. As explained in Section 2.5, a diagnosis is required to contain at least one configuration option from each unsatisfiable core to be able to satisfy the constraints. At one point, it was working with a partial diagnosis that consisted of more than 300 configuration options, and it was still not large enough to contain one configuration option from each unsatisfiable core. There is probably a bug behind this issue, because it seems unreasonable that there would be hundreds of unsatisfiable cores.

In other cases, the third stage of the algorithm took a long time to complete. In this stage, the generated diagnoses are used in conjunction with the constraints to generate fixes. To improve the performance of this process, the program only uses the constraints that involve the configuration options in the generated diagnoses. However, the set of related constraints might still be very large. For instance, when fixes for `MISDN_NETJET` were computed, there were 20 diagnoses and each one consisted of roughly 7 configuration options. Furthermore, for each diagnosis, 1,375 related constraints were found. All these factors together made it very resource demanding to simplify the constraints into fixes.

## 6.2.3 Performance results

As we saw in Figure 6.5, 34% of the runs resulted in fixes (*correct*, *too large* and *too small* combined). We will now examine the performance of these runs in further detail.

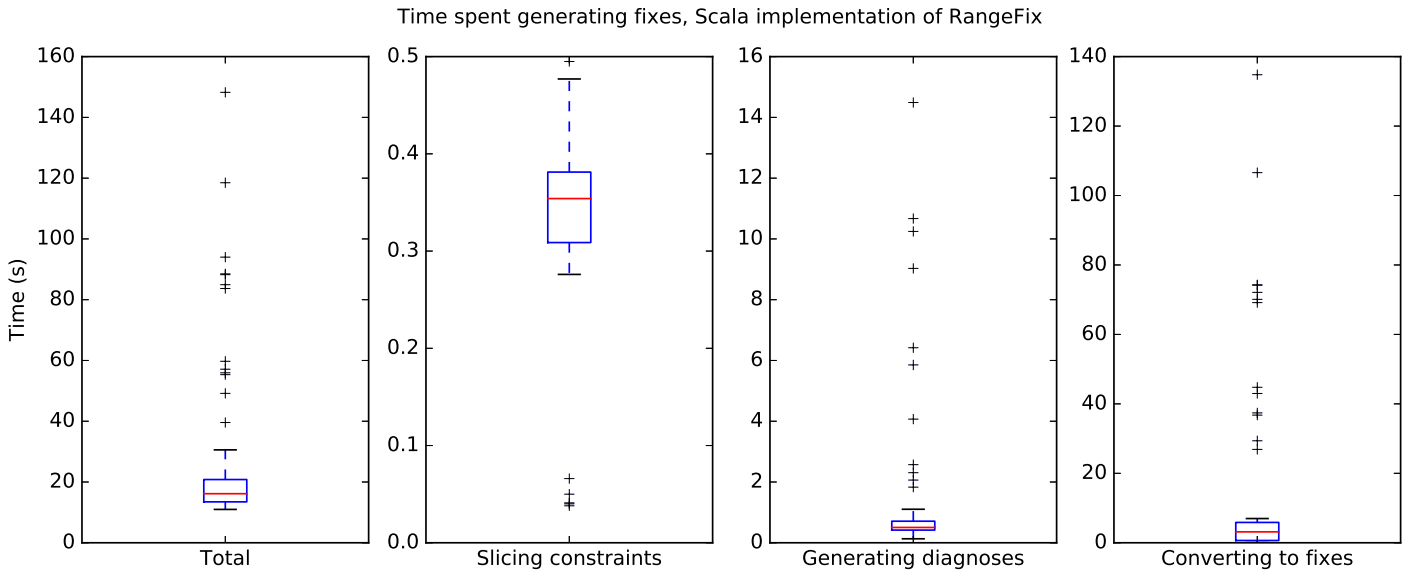
The number of iterations done in the first stage of RangeFix for generating the diagnoses is shown with a Tukey boxplot in Figure 6.7. The median is 6 iterations. 50% of the executions ran between 4 and 14 iterations. 75% ran between 2 and 14 iterations. Ignoring the outliers, 100% ran between 2 and 24 iterations. The outliers are 30, 34, 39, 47, 1029, 1029, 1030, 1030, 1031, 2006 and 3156. In the majority of cases, a small number of fixes were generated and these did also require relatively few iterations. However, there were some cases that required a lot of iterations,



**Figure 6.7:** The number of iterations done generating the diagnoses with the Scala implementation of RangeFix.

ignoring those executions that ran for more than five minutes. This explains why the mean is 159.5 number of iterations.

The running time of the program is illustrated with four Tukey boxplots in Figure 6.8. In the first plot, we can see that the median total running time, from that it is started until fixes are returned, is 16.15 seconds. The lower three quantiles, 75% of executions, finish between 10.97 and 20.81 seconds. However, the outliers are the reason behind the mean running time of 27.89 seconds. In the second plot, the running time for reducing the constraints is shown, which is a relatively quick activity. The median time is 0.35 seconds while the mean is 0.34 seconds. The third plot shows the running time of the activity for generating diagnoses, where the median is 0.50 seconds and the mean is 1.45 seconds. The fourth plot shows the running time for converting the diagnoses to fixes, where the median is 3.14 seconds and the mean is 14.14 seconds. The reason why the mean for the three activities do not add up to the total mean is because there are other activities, such as parsing the `.exconfig` input file, that also add some overhead to the program's total running time.



**Figure 6.8:** Time spent generating fixes with the Scala implementation of RangeFix.

## 6.3 The C implementation

In this thesis, the feasibility of implementing RangeFix in C using a SAT-based constraint solver is investigated. We have therefore built such a prototype of the algorithm, which implements the first stage of the algorithm and is therefore able to generate diagnoses. In Section 5.2, this partial C implementation of RangeFix was demonstrated. To be able to make an educated decision on how to proceed within the design science research methodology, it is necessary to evaluate the implementation's quality. In this section, we evaluate the quality in terms of correctness and performance. Whether a diagnosis accurately tells the user what configuration options whose values are required to be modified to enable a new configuration option determines its correctness. The time it takes to run the C implementation of RangeFix to generate diagnoses determines its performance.

### 6.3.1 Evaluation design

In this subsection, the evaluation design for the C implementation of RangeFix is explained. First, the procedure for the evaluation is presented. Next, how the diagnoses were classified in terms of correctness is presented.

#### 6.3.1.1 Procedure

To evaluate the C implementation of RangeFix, a set of 200 disabled configuration options were sampled. The selection of configuration options was done randomly from the set of configuration options fulfilling the following criteria:

- The configuration option's type is either bool or tristate. Required since these are the two input types that the C implementation of RangeFix supports.

- The configuration option has a prompt. Required since these are the only configuration options that the user can modify directly (invisible configuration options cannot be changed directly).
- The configuration option is currently set to `no`. Required since this state says that the configuration option is currently disabled.

The initial configuration was an `allnoconfig` with version 4.4.10 of the Linux kernel. The total number of configuration options fulfilling the above criteria on the x86 platform with an `allnoconfig` is 9,183. Why the kernel version differ from the one used in the evaluation of the Scala implementation is explained in Section 4.6.

For each randomly selected configuration option, the C implementation of RangeFix was executed to generate diagnoses for how to set it to `yes`. The running time was recorded, and the returned diagnoses were manually tested in `xconfig` for correctness. If it was possible to enable the configuration option by only modifying the values of the configuration options in the diagnoses, it was considered to be correct. If the program did not finish within 40 seconds, it was terminated.

The evaluation of our C implementation of RangeFix is limited to the diagnoses, as opposed to the evaluation of the existing Scala implementation of RangeFix, where the fixes were evaluated. The reason is that the C implementation is only a partial implementation of the RangeFix algorithm, since it is missing the second and the third stages of the algorithm. That is why the evaluation of it is limited to the diagnoses returned from the first stage.

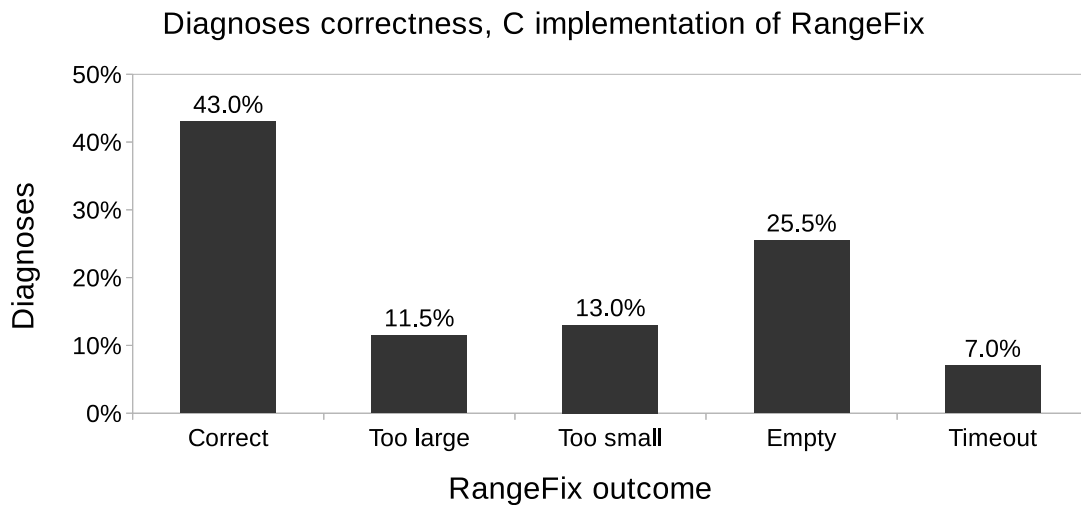
The timeout was set to only 40 seconds, as opposed to the five minutes that the existing Scala implementation of RangeFix got. This was due to a memory leak in the program, which caused it to crash after a few minutes. Efforts were made to locate the memory leak(s) through manual code inspection and analysis with Valgrind [48], but turned out unsuccessful.

### 6.3.1.2 Correctness classifications

The correctness of a generated diagnosis was determined by testing it in `xconfig`. As with the Scala implementation, we opted for classifying the generated diagnoses with the five labels: correct, too large, too small, empty, and timeout. For each of the 200 configuration options, the generated diagnoses were classified with one of these labels. The classifications are explained in further detail below.

**Correct.** If all generated diagnoses for a configuration option were minimal, and it was possible to enable the configuration option by only modifying the values of the configuration options in the returned diagnoses, it was classified as correct. By being minimal, a diagnosis contains only those configuration options that are required to edit to satisfy the unmet dependencies, which is a wanted property [30].

**Too large.** With a too large diagnosis, there were redundant configuration options that the user did not have to change. When one or more of the returned diagnoses contained redundant configuration options, the result was classified as being too large.



**Figure 6.9:** The results from calculating diagnoses for 200 disabled options.

**Too small.** If the returned diagnoses did not contain the complete set of required configuration options to edit the value of, it was classified as being too small. In other words, to enable the configuration option, the user would have needed to edit more configuration options than what were returned by the program.

**Empty.** If the program returned, but with an empty set of diagnoses, and the configuration option had unmet dependencies, the result was classified as being empty. As with the Scala implementation, the distinction from being too small was made because the error probably originated from a different source.

**Timeout.** If the program did not return within 40 seconds, it was classified with the label timeout. As already explained, this upper limit was implemented to limit the total running time. Furthermore, as mentioned previously, it had to be set to 40 seconds due to a memory leak that could not be located.

### 6.3.2 Correctness results

In this subsection, the correctness of the diagnoses, generated from the 200 runs, is evaluated. The results are summarized in Figure 6.9. These outcomes are discussed below, and various error sources are identified and explained.

#### 6.3.2.1 Correct

In 43.0% of the cases, the returned diagnoses were classified as correct. One such example was for the configuration option `IP_MULTIPLE_TABLES`, where the diagnosis `[IP_ADVANCED_ROUTER, NET, INET]` was generated. It was considered to be correct because it was possible to satisfy the dependencies to `IP_MULTIPLE_TABLES` by only modifying the values of the configuration options `IP_ADVANCED_ROUTER`, `NET` and `INET`.

### 6.3.2.2 Too large

In 11.5 % of the cases when the diagnoses were too large, it was because they were not simplified enough. As explained in Section 4.5, a step that was implemented in the C version of the algorithm was to remove any redundant configuration options before returning the diagnoses. It was possible to satisfy the dependencies by only modifying the values of the configuration options in the generated diagnoses. However, one or more of the diagnoses contained redundant configuration options that the user did not have touch.

For instance, generating diagnoses for `EFI_STUB` resulted in the following three intermediate diagnoses:

- [ACPI, EFI, NLS, PCI, RELOCATABLE, PNP, PCI\_GODIRECT]
- [ACPI, EFI, NLS, PCI, RELOCATABLE, PNP, PCI\_GOBIOS]
- [ACPI, EFI, NLS, PCI, RELOCATABLE, PNP, PCI\_GOOLPC, OLPC, GPIOLIB, OF]

These diagnoses were simplified into the following two diagnoses that were returned by the program:

- [EFI, PCI]
- [EFI, PCI, OLPC]

The second diagnosis is redundant, since it is possible to enable the configuration option `EFI_STUB` by only modifying the values of `EFI` and `PCI`.

Another example of this behaviour is the diagnoses returned for `BLK_DEV_CS5520`:

- [BLOCK, PCI, IDE]
- [BLOCK, PCI, IDE, DEFAULT\_NOOP]
- [BLOCK, PCI, IDE, PARTITION\_ADVANCED]
- [BLOCK, PCI, IDE, DEFAULT\_NOOP, PARTITION\_ADVANCED]

If the diagnosis [BLOCK, PCI, IDE] does already contain enough configuration options, it is of course unnecessary to also return variations of it that contain additional configuration options. A diagnosis should consist of a minimal set of configuration options whose values need to be modified.

### 6.3.2.3 Too small

In 13.0 % of the cases, when the diagnoses were too small, it was because they were simplified too much. During the simplification, redundant configuration options are removed from the generated diagnoses, as explained in Section 4.5. However, sometimes too many configuration options were mistakenly removed, which resulted in diagnoses that were missing necessary configuration options.

For instance, checking the logs from the diagnoses generation for the configuration option `NET_VENDOR_SIS`, we find that the following intermediate diagnoses were generated:

- [ETHERNET, NETDEVICES, PCI, NET, PCI\_GOMMCONFIG]
- [ETHERNET, NETDEVICES, PCI, NET, PCI\_GOANY]
- [ETHERNET, NETDEVICES, PCI, NET, PCI\_GODIRECT]
- [ETHERNET, NETDEVICES, PCI, NET, PCI\_GOBIOS]

They were simplified into the diagnosis [PCI, NET], which was returned by the program. However, only modifying the configuration options `PCI` and `NET` is not enough



to be able to enable `NET_VENDOR_SIS`. To satisfy its dependencies, `NETDEVICES` does also need to be part of the diagnosis, which means that `ETHERNET` and `PCI_*` were correctly removed, while `NETDEVICES` was incorrectly removed. This might be due to a bug in the SAT encoding of the Kconfig model or in the code that does the simplification.

#### 6.3.2.4 Empty

In 25.5 % of the cases, when an empty set of diagnoses was returned, it was due to the configuration option depended on a configuration option for another architecture. For instance, when diagnoses were generated for the configuration `LCS`, nothing was returned. Taking a look at its Kconfig definition, we find that it depends on the configuration option `S390`, which is only present if the user is configuring the kernel for IBM's S/390 mainframe architecture.

Another example was when diagnoses were generated for the configuration option `MTD_NAND_GPMI_NAND`. It depends on the expression `MTD && MTD_NAND && MXS_DMA`. Taking a closer look at these dependencies, we find that `MXS_DMA` depends on the expression `SOC_IMX23 || SOC_IMX28 || SOC_IMX6Q`, all of which are only present if the user is configuring the kernel for the ARM architecture. Even though the configuration options `MTD_NAND_GPMI_NAND` and `MXS_DMA` only serve a purpose on ARM, they are still present in xconfig when configuring the kernel for x86.

As with the correctness of the Scala implementation, this classification is therefore not strictly an error. However, as with the Scala implementation, it is a usability problem if there are cases when the conflict-resolution quits without any feedback and forces the user to find the error source by herself.

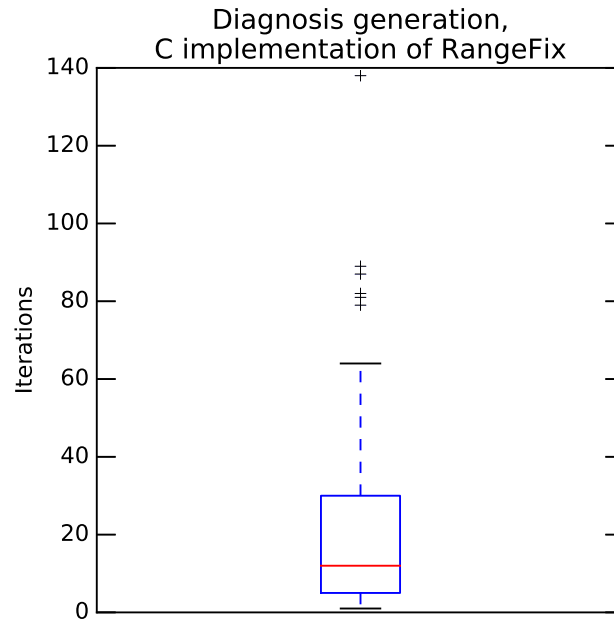
#### 6.3.2.5 Timeout

In 7.0 % of the cases, when running the program resulted in a timeout, it was because the generation of diagnoses never finished. One example when this happened was for the configuration option `IXGBEVF`, which depends on the expression `NETDEVICES && ETHERNET && NET_VENDOR_INTEL && PCI_MSI`. Examining the partial diagnoses generated during the program's execution, we find that they contain these configuration options, including the configuration options to satisfy their dependencies. Even though the partial diagnoses contain the necessary configuration options, the program believes it is still unsatisfiable and continues to run. It must therefore be something that wrongly triggers a conflict between the configuration options, that keeps the program running due to falsely detected unsatisfiable cores.

### 6.3.3 Performance results

67.5 % of the runs resulted in diagnoses being generated and returned (*correct*, *too small* and *too large* combined). We will examine the performance of these runs in further detail in this section.

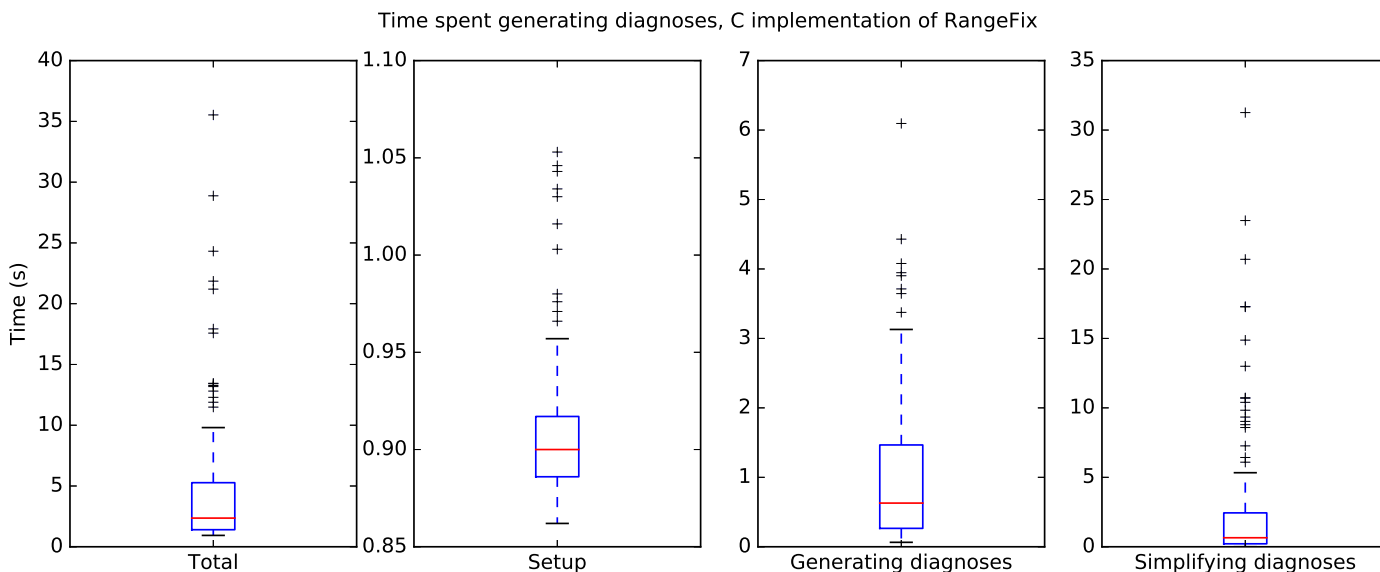
The number of iterations for generating the diagnoses is shown with a Tukey boxplot in Figure 6.10. The median is 12 iterations and the mean is 21.1 iterations.



**Figure 6.10:** The number of iterations done for generating diagnoses with the C implementation of RangeFix.

75% of the executions ran for less or equal to 30 iterations. There are also some outliers, that ran for 79, 81, 82, 87, 89 and 138 iterations.

The running time for the program is shown with four Tukey boxplots in Figure 6.11. In the first plot, we have the total running time for the program, from its initiation until diagnoses have been generated and are returned. The median is 2.4 seconds and the mean is 4.6 seconds. However, as can be seen in the plot, there are many outliers, the largest being 35.5 seconds. In the second plot we can see the time it takes to initialize PicoSAT with the translations of the constraints into PicoSAT clauses. This is a process whose mean and median are both 0.9 seconds. Since the constraints are always the same, there is not a big spread between the runs. In the third plot we have the time it takes to generate the diagnoses. This is also a relatively quick process, and its median is 0.6 seconds and its mean is 1.1 seconds. In the fourth plot we can see the time it takes for the program to simplify the diagnoses by removing any configuration options that the user does not need to set manually. The median for this activity is only 0.6 seconds, but its mean is 2.7 seconds due to many outliers that took as much as 31.3 seconds.



**Figure 6.11:** Time spent generating diagnoses with the C implementation of RangeFix.

## 6.4 Observations and conclusions

From the evaluations in the Sections 6.1, 6.2 and 6.3, we can make several observations. We will therefore comment on the results in the following subsections.

### 6.4.1 User survey

In the first question, the participants were asked if they ever run into issues while configuring the kernel. Of the 11 participants, 1 responded frequently, 3 occasionally, 5 rarely, and 2 never. This is somewhat comparable to the survey by Hubaux et al. [6], where they found that 56% of the participants considered enabling/disabling a configuration option to be a problem in practice. When we asked our participants to explain their answer further, reoccurring reasons were locating configuration options and satisfying dependencies.

To aid in configuring of the Linux kernel, a help text is provided to each configuration option. We asked the participants to rate the help text's usefulness. 1 replied that it is not helpful at all, 5 somewhat helpful, 3 helpful, and 1 very helpful. These numbers correlate somewhat to the ones mentioned in the previous paragraph.

We also asked the participants about the typical, shortest and longest time it takes them to configure a configuration option. The typical was on average 1.5 minutes, the shortest 20 seconds, and the longest 11 minutes. In the survey by Hubaux et al. [6], there are not a whole lot of raw numbers being presented. But we find that 20% of the Linux user participants stated that they need at least "a few dozen minutes" on average to figure out how to activate an inactive option. Our findings were not as bad.

To the survey participants, we showed our modified version of xconfig which has

functionality for generating and displaying fixes for satisfying unmet dependencies. 8 thought it would be beneficial in only some scenarios, and 3 thought it would not be beneficial. These numbers are not that surprising, since the implementation is currently fairly basic and requires one to manually apply the fixes. Furthermore, it is not always the case that one has to satisfy unmet dependencies to enable a configuration option. Therefore it makes sense that the participants would only consider the prototype to be beneficial in some scenarios.

The participants were also asked whether they considered the accomplishment in the video to be an improvement, where a configuration option and its four dependencies were located and enabled within 2 minutes. 4 participants replied it was not an improvement, 4 a slight improvement, 3 an improvement and nobody considered it to be a big improvement. These numbers correlate to the ones commented on in the previous paragraph, where 8 participants thought the tool would be beneficial in certain scenarios.

The participants were also given the opportunity to tell their preferred running time for generating fixes. They replied that they would prefer the fixes to be generated as soon as possible, which is not very surprising. However, since it is likely not possible to generate accurate fixes within zero amount of time, we also asked the maximum amount of time they would be willing to tolerate. The majority replied with maximums of 10 seconds or less.

#### 6.4.2 The Scala implementation

The correctness of the fixes from the Scala implementation was fairly good. In 34 % of the cases, fixes were returned. 59 % of these were correct, while 41 % were incorrect. The incorrect ones were primarily caused by the implementation lacking support for differentiating between configuration options with and without a prompt. Furthermore, there were some with minor usability issues with redundant configuration options. But overall, the returned fixes looked well.

The Scala implementation of RangeFix was experienced as being fairly slow. The quickest measured execution was 10.97 seconds, and 75 % of the executions finished within 20.81 seconds. As we can see, never was the program quicker than the users' expectation of it to take less than 10 seconds, which we found out in our user survey. Furthermore, the running time was rather irregular, with many outliers, which caused a mean running time of 27.89 seconds. In one instance the program even ran for more than 2 minutes.

Taking a closer look, we find that the performance irregularities stem from both the diagnoses generation and fix generation. Generating diagnoses was relatively quick, with a median of 0.50 seconds, while generating the fixes was slower, with a median of 3.14 seconds. But the mean times were 1.45 seconds and 14.14 seconds, respectively, which are a result of the frequent outliers.

We also recorded the number of iterations the algorithm did in its first stage, where the diagnoses are generated and the constraint solver continuously invoked to detect unsatisfiable cores. The median was 6 iterations, with the three lower quantiles being between 2 and 14 iterations. However, they ranged up to 3,156 iterations, with a mean of 159.5 iterations. This explains why the slowest time it

took to generate diagnoses was 14.5 seconds, compared to the mean of 1.45 seconds.

A source that also has a big negative impact on the performance of the Scala implementation is overhead from various activities, such as loading the application and parsing the `.exconfig` file. The mean for the total running time was 27.89 seconds, for generating diagnoses it was 1.45 seconds, and generating fixes 14.14 seconds. Subtracting those activities from the total, we find that the other activities consume on average 12.30 seconds.

### 6.4.3 The C implementation

The correctness of the C implementation's diagnoses was quite good. In 67.5% of the executions, diagnoses were generated. 64% of these were correct, while 36% were incorrect by being too small or big. When nothing was returned, it was because they contained architecture-specific configuration options, which happened in 25.5% of all cases, or because it ran for longer than the 40 seconds limit, which happened in 7.0% of all cases.

The performance of the diagnoses generation activity in the C implementation was comparable to the same activity in the Scala implementation. The mean running time for generating diagnoses in the C implementation was 1.1 seconds, and the median was 0.6 seconds. For the Scala implementation, the mean for generating diagnoses was 1.5 seconds, and the median 0.5 seconds.

The number of iterations done in the diagnoses generation activity differed somewhat. For the C implementation, the mean was 21.1 iterations, and the median was 12 iterations. For the Scala implementation, the mean was 159.5 iterations, and the median was 6 iterations. The three first quantiles for the C implementation ranged between 1 and 30 iterations, while the same statistic for the Scala implementation was between 2 and 14 iterations. The underlying algorithm is the same in both instances, however, factors such as the difference in kernel versions and timeout length might play a role.

The overhead from initiating the CNF clauses in PicoSAT was very small. The mean and median were both 0.9 seconds, and does only have to be done once at `xconfig`'s startup. In other words, when the `Kconfig` model has been translated into a SAT problem, the instance of the SAT problem can be reused for each subsequent time the user wants to generate diagnoses. This is possible since `xconfig` and the C implementation of `RangeFix` are both compiled into the same binary. The Scala implementation of `RangeFix`, on the other hand, has to be launched from scratch each time the user wants to generate fixes. A possible solution to reduce some overhead of starting and initializing the Scala implementation of `RangeFix`, would be to make it into a long-running background process that `xconfig` can communicate with through a socket.

The overall total running time for the C implementation was decent. The mean was 2.4 seconds, and the median was 4.6 seconds. However, the implementation is obviously lacking fix generation, which the evaluation of the Scala implementation proved to be relatively expensive. Furthermore, the running time of the C implementation does, as the Scala version, suffer from irregularities. There are many outliers, that in many cases cause a big negative impact on the running time. The

activity for simplifying the diagnoses, where redundant configuration options are removed, did also appear to be costly and consume a big proportion of the C implementation's total running time. Since the Scala implementation does not need such a step, it would be wise if the C implementation's diagnoses generation was improved in such a way that the need for simplifying the diagnoses is removed.

## 6.5 Threats to validity

There are internal threats to validity to be aware of. Since the survey was conducted by submitting it on various online discussion boards and mailing lists, it is based on a convenience sample, which might have an effect on the selection bias. For instance, it might be the case that people who engage on these sorts of forums are already experienced in configuring the kernel, and would therefore not appreciate additional assistance as much as a newcomer. On the other hand, the people who decided to participate in the survey might have done it because they have strong feelings about how the kernel is configured. It is therefore hard to draw any strong conclusions from such a sample.

Furthermore, the sample size was also fairly small, with only eleven participants. In the survey conducted by Hubaux et al. [6], where they utilized a similar method to get in contact with Linux users, they managed to get 97 participants. From a survey with only eleven participants, it is hard to make any quantitative claims.

There are also confounding variables that play a role in how the participants replied to the questions in the survey. One such confounding variable is the video presentation of this thesis' prototype. In the video, a rehearsed routine is carried out, which makes it look very easy. In addition, the video has also been tampered with, where elements have been sped up to shorten the length of the video. Such factors contribute to the participants' perception of the tool, and they become more inclined to give it a more positive rating than they would otherwise.

Another issues with the survey appears in the answer choices to question 13, see Appendix A and Figure 6.3. The choices the participants were able to pick from were "not an improvement", "slight improvement", "improvement" and "big improvement". However, what is missing is a negative scale, where the participants would be able to express a deterioration in the configurator. It is therefore not possible to distinguish between (a) those participants who considered it to not be an improvement and were neutral, and (b) those who considered it to have made the configurator worse. Four participants selected the answer choice "not an improvement", and it is unknown within which category of these two they are.

Many of the answer choices also suffer from being dependent on the participant's frame of reference. For instance, looking at the first question, see Appendix A and Figure 6.1, it asks whether they ever run into issues when configuring the kernel. However, one person might experience one issue per configuration session and consider it to be a frequent problem, while another person might experience ten issues per configuration session and consider it to only be a rare problem. Another example of where this problem appears is in question 8, see Figure 6.3, where the participant is supposed to select a answer choice to tell whether the prototype seems to be beneficial or not. However, answering that question depends heavily on the

participant's frame of reference. For instance, the participant might be paid by the hour to configure kernels and interprets a quicker process as something negative for herself. These two examples illustrate that the interpretation of the results can differ a lot from the participant's interpretation of the question.

Due to the discussed differences between the two implemented artifacts, where they work for different kernel versions and have differing degrees of completeness, there is also the case of instrument change to be aware of. If they had been implemented for the same kernel version, and had supported the same functionality, the comparability had been better. On the other hand, as discussed in Section 4.6, improving the comparability would have required more work.

Construct validity is also something to be aware of. The tests we created to measure the correctness, performance and usability qualities, might not have been optimal. There might be better ways to more accurately measure these qualities. For instance, in a controlled experiment it would have been possible to evaluate the usability of the demonstrated prototype in a much more accurate and structured way. It would have been possible to ask the participants to carry out a scenario, both in the regular configurator and in ours. By doing that, a proper benchmark would be established and any improvement or deterioration measured.

Since only conflict-resolution for the Linux kernel has been evaluated in this case study, the generalizability within the domain of software configurators remains unknown to some extent. This has a negative impact on the external validity. However, it can be generalized outside the Linux kernel to other projects that also use the Kconfig language for their feature model.

# 7

## Towards a SAT-based implementation in C

The Linux community would prefer to realize interactive conflict-resolution support in C and base it on a SAT solver. However, as is apparent by the evaluation of this thesis project's C implementation, there are still areas that need to be improved. In this chapter, we discuss identified challenges that need to be tackled to be able to finish the realization of such a mechanism. The challenges are categorized into four categories: 1. integrating the algorithm with xconfig, 2. encoding the Kconfig model as a SAT problem, 3. realizing the diagnoses generation, and 4. realizing fix generation.

### 7.1 Challenge #1: Integrating with xconfig

All Kconfig configurators bundled with the kernel restrict the user from entering an invalid configuration state [6], as opposed to other configurators that allow the user to enter an invalid state and use conflict-resolution to provide feedback and assistance in resolving the conflicts. In Section 2.1, the eCos Configuration Tool and `pure::variants` were discussed, and both those configurators use the latter approach where the configuration is allowed to temporarily enter an invalid state.

The source code of the Kconfig configurators would need a considerable overhaul to support a workflow where the user can make her desired configuration changes and afterwards fix any unsatisfied dependencies, because the configurators are currently engineered around the notion that no invalid states are allowed. As explained in Section 2.2.1, internally the Kconfig model's configuration options are loaded in a tree structure. Whenever the user modifies a configuration option's value, the configurator recalculates the values of any other configuration options related through `depends on`, `select` and `default` attributes. For instance, let say there are two boolean configuration options A and B, where A is set to `y` and B has an attribute `default y if A`; if the user changes the value of A to `n`, the value of B is also updated and is automatically set to `n`. (It falls back on `n` because an unconfigured configuration option without any satisfied `default` attributes is implicitly set to `n`.) The visibility of the configuration options is also determined by this data structure, where the evaluation of a configuration option's `depends on` expression determines its visibility.

Our C implementation of RangeFix gets its configuration values from iterating over the `symbol` structs in the internal Kconfig data structure, reading each



symbol's variable `curr`, which contains a configuration option's current value. The configuration options that the user wants to find fixes for are first modified in the Kconfig data structure, before diagnoses are generated. However, the Kconfig data structure is closely tied to the presentation of the configuration options in `xconfig`, which means that a better solution would isolate the data structure that the graphical user interface depends on from the data structure that RangeFix needs for its calculations.

Regardless of what language RangeFix is implemented in, the graphical interface of the dependency-resolution support in `xconfig` would also benefit from further work. The fixes could be presented in a more user friendly way, and applying the fixes could also be simplified. Rather than having to apply each assignment in a fix manually, it would be better if the user could, by the press of a button, choose a fix and have it applied automatically to the configuration. This would be easier to achieve if the algorithm is ported to C, because by using the Scala implementation, an additional step where the returned fixes are parsed as strings is necessary. If the underlying core of the configurator is also changed in such a way that invalid states are possible to enter, running conflict-resolution continuously to report on any detected conflicts could also benefit the user. With this prototype, the user has to manually initiate the fix calculation algorithm for a selected set of configuration options.

## 7.2 Challenge #2: SAT encoding

The first stage of RangeFix utilizes a constraint solver to find unsatisfiable cores. It is therefore crucial that the translation of the Kconfig model into a SAT problem is accurate. At the beginning of this thesis project, a decision was made to base it upon Satconfig, since that project had already implemented a translation of Kconfig models into SAT constraint clauses. However, Satconfig was found to contain some deficiencies in its encoding [49], which we propose a solution to in this section.

### 7.2.1 Proper tristate expression translation

Satconfig does not make any differentiation between a tristate's module and yes states. This means that lower and upper bounds that the Kconfig attributes `select` and `depends on` give rise to are not handled properly. For instance, if a tristate configuration option `depends on A && B`, only the first literal in A's and B's values will be considered. In Section 4.1, we saw that the first allocated literal for a configuration option represents if it has the value `y` or `n`, while the second literal represents if it has the value `y` and `m`. Since the second literal is ignored, it means that the dependency will currently be satisfied regardless of whether A and B are set to module or yes.

In a tristate expression, five operators are supported: NOT, `!A`; MIN, `A && B`; MAX, `A || B`; EQUAL, `A = B`; and NOT EQUAL, `A != B`. In the following sections, a truth table is presented for each operator together with a couple of CNF formulas that store the operator's truth value in a variable C. This will enable one to construct more complex expressions, as later illustrated in Section 7.2.2. By using this encoding for the dependency expressions, rather than the one currently

implemented in `Satconfig`, it should be possible to translate the expressions into SAT clauses that differentiate between the module and yes states.

### 7.2.1.1 NOT

The NOT operator produces the inverse of its input, as shown in Table 7.1. To store its output in a variable  $C$ , i.e.  $C := !A$ , two constraints need to be formulated—one constraint for each literal in  $C$ . By looking at the truth table in Table 7.1, we can see that the first literal in  $!A$ , i.e.  $(\neg A)_1$ , is equal to  $A_1 \rightarrow A_2$ . For instance, when  $A$  is set to `y`, its literals are  $A_1 = \text{true}$  and  $A_2 = \text{false}$ , which yields the value  $(\neg A)_1 = A_1 \rightarrow A_2 = \text{true} \rightarrow \text{false} = \text{false} = 0$ . In the truth table, we can see that the second literal in  $!A$ , i.e.  $(\neg A)_2$ , does not change from its original value, and is therefore equal to  $A_2$ . With these two observations, and by utilizing a Tseytin transformation [43], the CNF clauses for the two literals in  $C$  are:

$$C_1 = A_1 \rightarrow A_2 = \neg A_1 \vee A_2 \equiv (\neg A_1 \vee A_2 \vee \neg C_1) \wedge (A_1 \vee C_1) \wedge (\neg A_2 \vee C_1) \quad (7.1)$$

$$C_2 = A_2 \equiv (\neg A_2 \vee C_2) \wedge (A_2 \vee \neg C_2) \quad (7.2)$$

With these two constraints, it is possible to read the value of  $!A$  from the variable  $C$ .

**Table 7.1:** Truth table of the NOT operator.

A	!A
n 0 0	y 1 0
y 1 0	n 0 0
m 1 1	m 1 1

### 7.2.1.2 MIN

MIN works similarly to the logical AND operator ( $\wedge$ ) and returns the smallest value of its two operands, as depicted in Table 7.2. To be able to read the output from  $C$ , i.e.  $C := A \ \&\& \ B$ , constraints need to be declared. Finding the constraints is achieved by identifying patterns in the Karnaugh maps in Table 7.3 and Table 7.4. A Karnaugh map is used to simplify boolean algebra expressions, which it does by taking advantage of humans' pattern-recognition capability [50]. In the first Karnaugh map, we can easily spot that the first literal in  $A \ \&\& \ B$  is equal to 1 when both  $A_1$  and  $B_1$  are equal to 1. The second literal in  $A \ \&\& \ B$  is equal to 1 when either  $A_2$  or  $B_2$  is equal to 1. Formulating these observations as constraints for the two literals in  $C$ , with the help of Tseytin transformations [43], we get:

$$C_1 = A_1 \wedge B_1 \equiv (\neg A_1 \vee \neg B_1 \vee C_1) \wedge (A_1 \vee \neg C_1) \wedge (B_1 \vee \neg C_1) \quad (7.3)$$

$$C_2 = A_2 \vee B_2 \equiv (A_2 \vee B_2 \vee \neg C_2) \wedge (\neg A_2 \vee C_2) \wedge (\neg B_2 \vee C_2) \quad (7.4)$$

**Table 7.2:** Truth table of the MIN operator.

A	B	A && B
n 0 0	n 0 0	n 0 0
n 0 0	y 1 0	n 0 0
n 0 0	m 1 1	n 0 0
y 1 0	n 0 0	n 0 0
y 1 0	y 1 0	y 1 0
y 1 0	m 1 1	m 1 1
m 1 1	n 0 0	n 0 0
m 1 1	y 1 0	m 1 1
m 1 1	m 1 1	m 1 1

**Table 7.3:** Karnaugh map of the first literal in A && B.

		A			
		n 0 0	- 0 1	y 1 0	m 1 1
B	n 0 0	0	-	0	0
	- 0 1	-	-	-	-
	y 1 0	0	-	1	1
	m 1 1	0	-	1	1

**Table 7.4:** Karnaugh map of the second literal in A && B.

		A			
		n 0 0	- 0 1	y 1 0	m 1 1
B	n 0 0	0	-	0	0
	- 0 1	-	-	-	-
	y 1 0	0	-	0	1
	m 1 1	0	-	1	1

### 7.2.1.3 MAX

MAX works similarly to the logical OR operator ( $\vee$ ) and returns the largest value of its two operands, as depicted in Table 7.5. To be able to read the output from  $C$ , i.e.  $C := A \ || \ B$ , constraints need to be declared. To aid us in finding constraints that enforces that, we have the Karnaugh maps in Table 7.6 and Table 7.7. In Table 7.6 it is easy to spot the following pattern for  $C_1$ :

$$C_1 = A_1 \vee B_1 \equiv (A_1 \vee B_1 \vee \neg C_1) \wedge (\neg A_1 \vee C_1) \wedge (\neg B_1 \vee C_1) \quad (7.5)$$

$C_2$ , on the other hand, does not have such an easy formula. There are three regions in Table 7.7 where  $C_2$  is negative, which translates to the clauses:

$$(A_1 \vee B_1) \wedge (\neg A_1 \vee A_2) \wedge (\neg B_1 \vee B_2) \quad (7.6)$$

There are also three regions where  $C_2$  is positive:

$$(\neg A_1 \vee \neg A_2 \vee B_1) \wedge (A_1 \vee \neg B_1 \vee \neg B_2) \wedge (\neg A_2 \vee \neg B_2) \quad (7.7)$$

These clauses yield the following constraints that enforce the correct output value in  $C_2$ :

$$\begin{aligned}
 & (A_1 \vee B_1 \vee \neg C_2) \wedge \\
 & (\neg A_1 \vee A_2 \vee \neg C_2) \wedge \\
 & (\neg B_1 \vee B_2 \vee \neg C_2) \wedge \\
 & (\neg A_1 \vee \neg A_2 \vee B_1 \vee C_2) \wedge \\
 & (A_1 \vee \neg B_1 \vee \neg B_2 \vee C_2) \wedge \\
 & (\neg A_2 \vee \neg B_2 \vee C_2)
 \end{aligned} \tag{7.8}$$

**Table 7.5:** Truth table of the MAX operator.

A	B	A    B
n 0 0	n 0 0	n 0 0
n 0 0	y 1 0	y 1 0
n 0 0	m 1 1	m 1 1
y 1 0	n 0 0	y 1 0
y 1 0	y 1 0	y 1 0
y 1 0	m 1 1	y 1 0
m 1 1	n 0 0	m 1 1
m 1 1	y 1 0	y 1 0
m 1 1	m 1 1	m 1 1

**Table 7.6:** Karnaugh map of the first literal in A || B.

		A			
		n 0 0	- 0 1	y 1 0	m 1 1
B	n 0 0	0	-	1	1
	- 0 1	-	-	-	-
	y 1 0	1	-	1	1
	m 1 1	1	-	1	1

**Table 7.7:** Karnaugh map of the second literal in A || B.

		A			
		n 0 0	- 0 1	y 1 0	m 1 1
B	n 0 0	0	-	0	1
	- 0 1	-	-	-	-
	y 1 0	0	-	0	0
	m 1 1	1	-	0	1

#### 7.2.1.4 EQUAL

The EQUAL operator returns y if its two operands are equal and otherwise n, as depicted in Table 7.8. We want to construct a variable  $C$  which contains the value

of that expression, i.e.  $C := A = B$ . We will start with  $C_1$ , whose pattern in the Karnaugh map in Table 7.9 translates into the following formula:

$$C_1 = A_1 \leftrightarrow B_1 \wedge A_2 \leftrightarrow B_2 \quad (7.9)$$

All nine outcomes for  $C_1$  in the Karnaugh map in Table 7.9 translate into the following CNF clauses:

$$\begin{aligned} & (A_1 \vee A_2 \vee B_1 \vee B_2 \vee C_1) \wedge \\ & (A_1 \vee A_2 \vee \neg B_1 \vee B_2 \vee \neg C_1) \wedge \\ & (\neg A_1 \vee A_2 \vee \neg B_1 \vee \neg B_2 \vee \neg C_1) \wedge \\ & (\neg A_1 \vee A_2 \vee B_1 \vee B_2 \vee \neg C_1) \wedge \\ & (\neg A_1 \vee A_2 \vee \neg B_1 \vee B_2 \vee C_1) \wedge \\ & (\neg A_1 \vee A_2 \vee \neg B_1 \vee \neg B_2 \vee \neg C_1) \wedge \\ & (\neg A_1 \vee \neg A_2 \vee B_1 \vee B_2 \vee \neg C_1) \wedge \\ & (\neg A_1 \vee \neg A_2 \vee \neg B_1 \vee B_2 \vee \neg C_1) \wedge \\ & (\neg A_1 \vee \neg A_2 \vee \neg B_1 \vee \neg B_2 \vee C_1) \end{aligned} \quad (7.10)$$

However, since there are some undefined values in the truth table, there is room for minimizing them into smaller clauses. This minimization was achieved by a brute-force algorithm. The seven remaining CNF clauses are:

$$\begin{aligned} & (A_1 \vee A_2 \vee B_1 \vee B_2 \vee C_1) \wedge \\ & (A_1 \vee \neg B_1 \vee B_2 \vee \neg C_1) \wedge \\ & (A_2 \vee \neg B_2 \vee \neg C_1) \wedge \\ & (\neg A_1 \vee B_1 \vee \neg C_1) \wedge \\ & (\neg A_1 \vee A_2 \vee \neg B_1 \vee B_2 \vee C_1) \wedge \\ & (\neg A_2 \vee B_2 \vee \neg C_1) \wedge \\ & (\neg A_2 \vee \neg B_2 \vee C_1) \end{aligned} \quad (7.11)$$

Lastly,  $C_2$  is always negative, which is equal to the CNF clause:

$$(\neg C_2) \quad (7.12)$$

**Table 7.8:** Truth table of the EQUAL operator.

A	B	A = B
n 0 0	n 0 0	y 1 0
n 0 0	y 1 0	n 0 0
n 0 0	m 1 1	n 0 0
y 1 0	n 0 0	n 0 0
y 1 0	y 1 0	y 1 0
y 1 0	m 1 1	n 0 0
m 1 1	n 0 0	n 0 0
m 1 1	y 1 0	n 0 0
m 1 1	m 1 1	y 1 0

**Table 7.9:** Karnaugh map of the first literal in  $A = B$ .

		A			
		n 0 0	- 0 1	y 1 0	m 1 1
B	n 0 0	1	-	0	0
	- 0 1	-	-	-	-
	y 1 0	0	-	1	0
	m 1 1	0	-	0	1

### 7.2.1.5 UNEQUAL

The UNEQUAL operator returns *y* if its two operands are equal and otherwise *n*, as depicted in Table 7.10. We want to construct a variable *C* which contains the value of that expression, i.e.  $C := A \neq B$ . We will start with  $C_1$  whose pattern in the Karnaugh map in Table 7.11 translates into the following formula:

$$C_1 = \neg(A_1 \leftrightarrow B_1) \vee \neg(A_2 \leftrightarrow B_2) \tag{7.13}$$

Translating all nine outcomes for  $C_1$  from the Karnaugh map in Table 7.11 into CNF clauses yield us the following formula:

$$\begin{aligned}
 & (A_1 \vee A_2 \vee B_1 \vee B_2 \vee \neg C_1) \wedge \\
 & (A_1 \vee A_2 \vee \neg B_1 \vee B_2 \vee C_1) \wedge \\
 & (\neg A_1 \vee A_2 \vee \neg B_1 \vee \neg B_2 \vee C_1) \wedge \\
 & (\neg A_1 \vee A_2 \vee B_1 \vee B_2 \vee C_1) \wedge \\
 & (\neg A_1 \vee A_2 \vee \neg B_1 \vee B_2 \vee \neg C_1) \wedge \\
 & (\neg A_1 \vee A_2 \vee \neg B_1 \vee \neg B_2 \vee C_1) \wedge \\
 & (\neg A_1 \vee \neg A_2 \vee B_1 \vee B_2 \vee C_1) \wedge \\
 & (\neg A_1 \vee \neg A_2 \vee \neg B_1 \vee B_2 \vee C_1) \wedge \\
 & (\neg A_1 \vee \neg A_2 \vee \neg B_1 \vee \neg B_2 \vee \neg C_1)
 \end{aligned} \tag{7.14}$$

However, since there are undefined values in the truth table, there is room for minimizing them into smaller clauses. The minimization was achieved by a brute-

force algorithm, and the result is these seven clauses:

$$\begin{aligned}
 & (A_1 \vee A_2 \vee B_1 \vee B_2 \vee \neg C_1) \wedge \\
 & (A_1 \vee \neg B_1 \vee B_2 \vee C_1) \wedge \\
 & (A_2 \vee \neg B_2 \vee C_1) \wedge \\
 & (\neg A_1 \vee B_1 \vee C_1) \wedge \\
 & (\neg A_1 \vee A_2 \vee \neg B_1 \vee B_2 \vee \neg C_1) \wedge \\
 & (\neg A_2 \vee B_2 \vee C_1) \wedge \\
 & (\neg A_2 \vee \neg B_2 \vee \neg C_1)
 \end{aligned} \tag{7.15}$$

Lastly,  $C_2$  is always negative, which is equal to the CNF clause:

$$(\neg C_2) \tag{7.16}$$

**Table 7.10:** Truth table of the UNEQUAL operator.

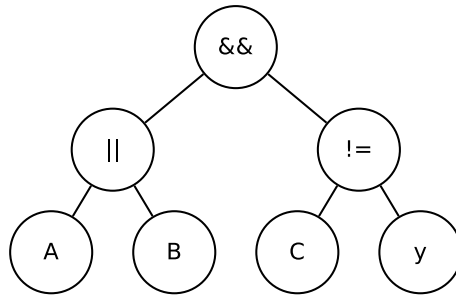
A	B	A != B
n 0 0	n 0 0	n 0 0
n 0 0	y 1 0	y 1 0
n 0 0	m 1 1	y 1 0
y 1 0	n 0 0	y 1 0
y 1 0	y 1 0	n 0 0
y 1 0	m 1 1	y 1 0
m 1 1	n 0 0	y 1 0
m 1 1	y 1 0	y 1 0
m 1 1	m 1 1	n 0 0

**Table 7.11:** Karnaugh map of the first literal in  $A \neq B$ .

		A			
		n 0 0	- 0 1	y 1 0	m 1 1
B	n 0 0	0	-	1	1
	- 0 1	-	-	-	-
	y 1 0	1	-	0	1
	m 1 1	1	-	1	0

## 7.2.2 Use operators in conjunction

Since each operator takes one or two variables as inputs and puts its output in a new variable, it is possible to recursively translate Kconfig expressions into CNF constraints. Let us illustrate how this works with an example,  $(A \parallel B) \&\& (C \neq y)$ . The expression can be viewed as a tree, as depicted in Figure 7.1. We will traverse the tree in depth-first post-order. Starting with  $(A \parallel B)$ , we will create a new auxiliary variable  $D$  which is equal to  $D := A \parallel B$ . By using the two CNF



**Figure 7.1:** The Kconfig expression  $(A \ || \ B) \ \&\& \ (C \ != \ y)$  drawn as a binary tree.

A	n
B	m
C	y
D	
E	y
F	
G	

(a) Initial state.

A	n
B	m
C	y
D	m
E	y
F	
G	

(b) Step 1.

A	n
B	m
C	y
D	m
E	y
F	n
G	

(c) Step 2.

A	n
B	m
C	y
D	m
E	y
F	n
G	n

(d) Step 3.

**Figure 7.2:** How variables, including auxiliary ones, created from the expression in Figure 7.1, are filled out and yields  $n$ .

constraints defined earlier for MAX operator, Formula 7.5 and Formula 7.8, it is possible to read the result of the operator from D. For  $(C \ != \ y)$ , we will use the CNF constraints for the NO EQUAL operator, an auxiliary variable E where the constant  $y$  is put, and an auxiliary variable F where the operator's result is put. The last operator is MIN, which is found in the bubble at the top of Figure 7.1. Rather than finding constraints for the complete expression  $(A \ || \ B) \ \&\& \ (C \ != \ y)$ , we will instead replace its operands with the auxiliary variables D and F, resulting in the expression  $D \ \&\& \ F$ , and use the two CNF constraints we already declared for the MIN operator. Once again we will create a new auxiliary variable, G, where the resulting value of the complete expression can be read. Assume we have the assignments  $A := n$ ,  $B := m$  and  $C := y$ , it results in the initial variable assignments as depicted by Figure 7.2a. The CNF constraint clauses will then force the SAT solver to generate the auxiliary variables' values as depicted by Figure 7.2a, Figure 7.2b and Figure 7.2c. The end result of the expression is read from G, which is equal to  $n$  in this case.

### 7.3 Challenge #3: Realize diagnoses

It is during the first stage of the RangeFix algorithm that diagnoses are generated. Although the first stage of the RangeFix algorithm has been implemented in C in this thesis, there are still open issues left that need to be resolved for xconfig to have a satisfying dependency-resolution mechanism.



### 7.3.1 Using the internal Kconfig infrastructure for computing the configuration

A convenient way to read the configuration, including configuration options that are implicitly set through `default` and `select` attributes, is to utilize the internal Kconfig infrastructure. In the RangeFix algorithm, a constraint solver is invoked to find an unsatisfiable core in  $(C \setminus E_0)$ , where  $C$  is the unsatisfied constraint set and  $E_0$  is a partial diagnosis. The most straightforward way to get the soft constraints, i.e. the current configuration, is to iterate over the `symbol` structs in the `rootmenu` data structure and copy the `curr` variable value of each configuration option. By using this method, it is easy to get soft constraints such as `[A:=n, B:=m, C:=y]`. With a set operation, it is then possible to compute  $(C \setminus E_0)$ , which the constraint solver is invoked with. Another advantage with this method is that the internal Kconfig infrastructure is utilized for parsing the `.config` file and calculating the default values and reverse-dependencies, which guarantees that the configuration is correctly interpreted.

The problem with utilizing the internal Kconfig infrastructure for computing the complete configuration, including implicitly set configuration options, is that the configuration will not be updated during the execution of the diagnoses generation algorithm. In reality, the complements of the partial diagnoses in the configuration cannot be expected to remain static during the execution of the algorithm. The inclusion of a configuration option in a partial diagnosis might affect the values of the configuration options in the complement of the partial diagnosis. The configuration should therefore instead be recomputed by the internal Kconfig infrastructure during each iteration of the algorithm. However, the C functions for parsing the Kconfig model and the `.config` file—`conf_parse` and `conf_read`—only work with the data structure that is stored in the global variable `rootmenu`. The same `rootmenu` data structure is also what the various Kconfig configurators depend on for displaying the configuration options' data, which means there can potentially be unexpected side effects if the same data structure is used for multiple purposes. Furthermore, many other miscellaneous functions in the source code are also tailored for there to only be one such global data structure, which means that making a separate copy of it would require some effort to implement.

### 7.3.2 Implicitly configured configuration options

Let us illustrate with an example how building a diagnosis might have ripple effects on the rest of the configuration. In Listing 7.1, there are three configuration options, where `A` depends on both `B` and `C`, and `C` defaults to `y` if `B` is equal to `n`. In Listing 7.2, two of the configuration options have been configured: `A` has been set to `y` and `B` has been set to `n`. `C` is implicitly set to `y` by its `default` attribute. This configuration does not satisfy the constraints, since `B` has to be set to `y` for `A` to be permitted to be set to `y`. During the first iteration of the diagnoses generation, the partial diagnosis will be an empty set, `{}`, and the soft constraints will be `[A:=y, B:=n, C:=y]`. Since there is a violation, an unsatisfiable core `{A, B}` will be found. The two partial diagnoses will therefore be `{A}` and `{B}`. Continuing with the partial

**Listing (7.1)** A Kconfig model where A depends on both B and C.

```
1 config A
2     tristate "A"
3     depends on B
4     depends on C
5
6 config B
7     tristate "B"
8
9 config C
10    tristate "C"
11    default y if B=n
```

**Listing (7.2)** A .config configuration file where the configuration options A and B in Listing 7.1 have been configured.

```
1 CONFIG_A=y
2 CONFIG_B=n
```

diagnosis {B} in the next iteration of the diagnoses generation, the soft constraints will be [A:=y, C:=y]. No unsatisfiable core will be found this time, since the configuration will be satisfiable with only these two soft constraints' assignments. The information we have at the end of this stage of RangeFix is that it is possible to satisfy the constraints if the value of configuration option B is changed. It is only later, in stage three, that a fix will be computed and we will find out that B has to be set to y to satisfy the configuration. However, that will clearly make C default to n since its if expression will not be satisfied anymore. Configuration option C can surely still be manually set to y by the user, but its default, when nothing has been assigned to it, is n if B!=n. How to properly handle this scenario remains an open issue.

## 7.4 Challenge #4: Realize fix generation

When the diagnoses have been generated, the next two stages of the RangeFix algorithm use the diagnoses to transform the feature model's constraints into fixes.

### 7.4.1 Problem formulation

The RangeFix algorithm has three stages and in each of them the constraints, defined by the Kconfig model, have to be employed in a somewhat different manner.

In the first stage of RangeFix, we need to be able to set the soft constraints, i.e. the values for a set of configuration options, and identify unsatisfiable cores among them. As long as these two abilities are available, it does not matter what form the model's constraints are in. The SAT encoding described in Section 4.1 creates a lot of extra literals, meant to simplify the definitions of the constraints. But at this stage of the algorithm, it is of no significance if the constraints are polluted by such miscellaneous literals.

In the second stage of RangeFix, we need to have the constraints in such a form that we are able to substitute configuration options with their values from the configuration. This means that we cannot work with the CNF clauses of PicoSAT literals from the previous stage. Such a clause might for instance be 3 4 7 0, which

does not necessarily map into meaningful configuration option names. Instead, the constraints need to be in a form on a level higher than clauses of numerical literals.

In the third stage of RangeFix, we need to simplify the modified constraints from the second stage into fixes. It is therefore important that the modified constraints are only made up of configuration option names and constants. No miscellaneous PicoSAT literals, created during the first stage, can be permitted to be part of these formulas. Literals, such as those telling if a default's expression is satisfied and so on, cannot be part of the fixes because they are not possible to represent in a concrete fix that the user can easily apply. The modified constraints that are inputs to this stage must therefore be clean from all such noise.

Note that it is only the first stage of the RangeFix algorithm that strictly depends on a constraint solver. In other words, it is only in the first stage that we need to transform the constraints into a SAT or SMT problem. In the second and third stages, we can have the constraints in any representation, as long as we are able to heuristically minimize them into meaningful assignments that can be interpreted as a fix by the user.

### 7.4.2 Attempted approach

The approach that was attempted in this thesis work, with unsuccessful outcome, was to mimic the examples given in the RangeFix paper. For each configuration option, constraints for `depends on` and `select` were formulated in such a way that they would be human-readable and have a single truth value. With these two properties, the belief was that it would be possible to simplify the constraints into fixes using heuristic rules, and without deniability evaluate if they were violated or not.

Assume we have the Kconfig model in Listing 7.3, we would then for `A depends on B` create the two constraints:

$$\begin{aligned} (A=y) \rightarrow (B=y) &\equiv !(A=y) \vee (B=y) \\ &\equiv A!=y \vee B=y \\ (A=m) \rightarrow (B=m \vee B=y) &\equiv !(A=m) \vee (B=m \vee B=y) \\ &\equiv A!=m \vee B!=n \end{aligned}$$

If `A` is set to `y`, it implies that its dependency is satisfied, i.e. that `B` is also set to `y`. But if `A` is only set to `m`, it implies that its dependency `B` is either `m` or `y`. The value of `B` creates an upper bound for `A`.

Continuing with the Kconfig model in Listing 7.3, the `B select C` attribute would give rise to the following two constraints:

$$\begin{aligned} (B=y) \rightarrow (C=y) &\equiv !(B=y) \vee (C=y) \\ &\equiv B!=y \vee C=y \\ (B=m) \rightarrow (C=m \vee C=y) &\equiv !(B=m) \vee (C=m \vee C=y) \\ &\equiv B!=m \vee C!=n \end{aligned}$$

If `B` is set to `y`, it also forces `C` to have the value `y`. But if `B` is set to `m`, `C` is forced to either `m` or `y`. The value of `B` creates a lower bound for `C`.

**Listing 7.3:** A Kconfig file with three configuration options.

```

1  config A
2      tristate "A prompt"
3      depends on B
4
5  config B
6      tristate "B prompt"
7      select C
8
9  config C
10     tristate "C prompt"
    
```

The complete set of constraints for the model in Listing 7.3 is therefore:

$$(A!=y \vee B=y) \wedge (A!=m \vee B!=n) \wedge (B!=y \vee C=y) \wedge (B!=m \vee C!=n)$$

These are the constraints that are needed for the second stage of RangeFix, where the constraints are transformed into modified constraints. For instance, if the diagnosis is  $\{A, B\}$  and the configuration is  $[C:=y]$ , the modified constraints are:

$$(A!=y \vee B=y) \wedge (A!=m \vee B!=n) \wedge (B!=y \vee y=y) \wedge (B!=m \vee y!=n)$$

Minimizing the modified constraints into a fix, by using some basic heuristics, yields:

$$\begin{aligned} (A!=y \vee B=y) \wedge (A!=m \vee B!=n) \wedge (B!=y \vee y) \wedge (B!=m \vee y) &\equiv \\ (A!=y \vee B=y) \wedge (A!=m \vee B!=n) \wedge (y) \wedge (y) &\equiv \\ (A!=y \vee B=y) \wedge (A!=m \vee B!=n) & \end{aligned}$$

However, there might be room for further improvements in how a fix is minimized.

The problem with this approach becomes apparent when a variable's value causes changes to the rest of the configuration. Assume the diagnosis is  $\{B\}$ , the configuration is  $[A:=y]$  and  $C$  is implicitly set to  $n$ , the fix is then:

$$\begin{aligned} (A!=y \vee B=y) \wedge (A!=m \vee B!=n) \wedge (B!=y \vee C=y) \wedge (B!=m \vee C!=n) &\equiv \\ (y!=y \vee B=y) \wedge (y!=m \vee B!=n) \wedge (B!=y \vee n=y) \wedge (B!=m \vee n!=n) &\equiv \\ (n \vee B=y) \wedge (y \vee B!=n) \wedge (B!=y \vee n) \wedge (B!=m \vee n) &\equiv \\ (B=y) \wedge (y) \wedge (B!=y) \wedge (B!=m) &\equiv \\ (B=y) \wedge (B!=y) \wedge (B!=m) & \end{aligned}$$

We can clearly see that there are contradicting assignments, even though it should be possible to set both  $A$  and  $B$  to  $y$  at the same time.

Setting  $B$  to  $y$  will also modify the value  $C$ , due to the reverse-dependency.  $C$  is in this example currently implicitly set to  $n$ , but setting  $B$  to  $y$  will also set  $C$  to  $y$ . The problem is that when we build the fix above, we first assume  $C$  to be equal to  $n$ , and leave  $B$  without a value. But when the constraints get minimized into a fix, we get contradicting assignments, which stem from the fact that  $C$  cannot be assumed to be  $n$  in all cases. It is from this example clear that it is not that simple to generate

human-readable propositional constraints from a Kconfig model that are possible to minimize into fixes.

Another problem with this approach is how one would express `default` values. A `default` is not a constraint per se, since it is just a fallback value for a configuration option that the user has not configured yet. A user is perfectly eligible to override a configuration option's default value by explicitly assigning a value to the configuration option in the `.config` file. However, changing one configuration option's value might satisfy another configuration option's `default if` expression. In other words, configuring an configuration option might also affect other configuration options that have not been configured by the user, that instead rely on their `default` value. How to express this concept in a fix is also unclear.

This approach was implemented and tested by us in C. But due to the described problem of implicitly configured configuration options, it was not possible to minimize the constraints into meaningful fixes.

### 7.4.3 Alternative approaches

In the previous subsection, we described that our attempted method to generate fixes by formulating and minimizing propositional formulas from the Kconfig constraints did not work out. Generating fixes by utilizing the constraint solver from the first stage of the RangeFix algorithm is an alternative approach that can be investigated. In the first stage of RangeFix, the Kconfig model is encoded using constraint clauses, and the configuration options' values are available through the constraint solver's variables. For each diagnosis, it should be possible to generate all permutations of value assignments to its configuration options. For each permutation, the constraint solver could be invoked to check if it is an assignment that does not cause any constraint violations. When each permutation has been tested, it would then be possible to write a fix that expresses the valid configuration option value assignments that satisfy the constraints.

For instance, assume that the diagnosis `{A, B, C}` has been found during the first stage of RangeFix. That means that it is possible to satisfy the configuration if those three configuration options are changed. However, we do not know yet how they need to be changed. If all three configuration options in the diagnosis are tristates, it means that there are  $3^3 = 27$  permutations to test. With the results from those tests, it might be possible to formulate a fix such as `[A = yes, B != no, C = mod]`.

A possible disadvantage with this approach is that it can be very resource demanding. If there are many diagnoses and each diagnosis consists of many configuration options, the number of times the constraint solver is invoked might be huge. On the other hand, the advantage with this method is that it is very simple, especially since the setup of the constraint solver can be reused from the first stage of RangeFix.

Another approach for generating fixes that can be investigated is the one that the existing Scala implementation of RangeFix utilizes. It uses propositional logic to write the constraints, but it has also added functionality to express if-then-else statements. For each configuration option, it creates two functions that calculate the

lower and upper bounds using its propositional logic language. However, how this approach works in detail is undocumented. Reverse-engineering would therefore be required if one wants to port it to another language. Furthermore, the Scala implementation of RangeFix uses a lot of functional programming techniques which are inherently difficult to copy directly to C, since C is a procedural programming language.

It would probably be beneficial for solving this problem if an intermediate form of the constraints was added. As the C implementation of RangeFix is built, it reads the Kconfig data structure built by the internal Kconfig infrastructure and immediately feeds PicoSAT with variables and clauses that make up the constraints imposed by the model. The Kconfig data structure could instead be used to build an intermediate format of the constraints that is easier to translate into forms that fit both purposes. By doing this, having to translate the same configuration options, dependencies, expressions and edge cases multiple times would be circumvented.

# 8

## Conclusion

In this case study, we have researched the feasibility of realizing scalable conflict-resolution for a configuration system as large as the Linux kernel configurator. In academia, we found the algorithm RangeFix, which had an existing implementation in Scala. By integrating this implementation of RangeFix with the Linux kernel configurator xconfig, we evaluated its suitability for resolving configuration conflicts. Through a survey, which Linux developers and users took part in, we evaluated whether such a system would be beneficial. We also studied how the algorithm could be implemented in C, utilizing a SAT-based constraint solver, which would achieve greater community acceptance.

The existing Scala implementation of RangeFix, we found to be unsuitable in its current shape to be integrated into the Linux kernel configurators. To start with, it is highly unlikely that a non-C solution to the problem will ever be accepted into the mainline Linux kernel branch. Especially a solution written in Scala, which requires one to install a 70 MB OpenJDK binary and a 110 MB Scala binary. Furthermore, the source code of the Scala implementation of RangeFix has not been maintained for quite some time, and compiling it requires old versions of Java and Scala. These old dependencies are not so easily available on a modern Linux distribution. The existing Scala implementation also depends on an SMT solver, however, the Linux community would prefer to base a solution on a SAT solver. The source code would also need to be further polished. It is currently only implemented as a proof of concept within a research project setting, and is therefore lacking in documentation and coding standards. It is also lacking in features, for instance, it is only possible to resolve unmet dependencies with inputs of the type tristate, not even boolean inputs are currently supported. In our evaluation, we also found the implementation to generally perform worse than the users' expectations. Furthermore, the running time is also irregular, and for some configuration options it may run for a very long time. Since we have to run the Scala implementation of RangeFix as an external program in xconfig, it also poses a challenge in C when its returned fixes, which can possibly contain complex expressions, must be parsed as strings. The existing Scala implementation of RangeFix works very well for demonstrating the capabilities of the algorithm, but it is not production-ready, which neither was its authors' intention.

Our C implementation of the RangeFix algorithm is neither production-ready. Only the first stage of the algorithm has been implemented, which means that it is only able to generate diagnoses, but not fixes. We based our work on Satconfig, which translates Kconfig models into SAT problems. However, Satconfig was found to contain deficiencies, such as not handling tristates in dependency expressions correctly. The performance did also suffer from the same irregularities as the Scala

implementation did, which might be due to complexities in the Kconfig model or a case that the RangeFix algorithm does not handle well. However, even if our C implementation of RangeFix is lacking in several regards, it did prove the principle. It is possible to implement scalable conflict-resolution support for a software system as large as the Linux kernel. The Linux community's requirements were respected, and the RangeFix algorithm appears to work well for conflict-resolution. However, our implementation would need more development before it would be acceptable into the mainline kernel repository.

A reoccurring problem is the many edge cases that exist in the Kconfig language. What the language lacks, is a complete specification. This explains why implemented parsers of the language do commonly suffer from deficiencies. In this thesis, we have observed that the existing implementation of RangeFix did not correctly handle assignment of values to configuration options without prompts, and Satconfig got tristates wrong. But other tools, such as Undertaker, Kconfigreader and LVAT, have also been found to make various errors [21].

To solve the problems inherent in implementing Kconfig language parsers and feature modeling tools, the language would likely need to be refined. A parallel can be drawn to the markup language Markdown, which has a plethora of community implementations with subtle differences. That sparked the initiative CommonMark [51], which attempts to unify the Markdown implementations by providing an extensive specification, examples, a reference implementation, and a test suite. A similar initiative for the Kconfig language would probably be healthy. Another suggestion is to simplify the Kconfig language, and design it for being more easily translated into logic propositional formulas that can be used by reasoners. Rather than trying to bolt advanced translations of Kconfig on top the language and the kernel's Kconfig infrastructure, it is probably wiser to design the language from the ground up for being more easily compatible with reasoners.



# Bibliography

- [1] S. Bhartiya, *Linux is the largest software development project on the planet: Greg Kroah-Hartman*, CIO, Ed., <http://www.cio.com/article/3069529/linux/linux-is-the-largest-software-development-project-on-the-planet-greg-kroah-hartman.html>, 2016.
- [2] T. Berger, S. She, R. Lotufo, A. Wasowski, and K. Czarnecki, “A study of variability models and languages in the systems software domain”, *Transactions on Software Engineering (TSE)*, vol. 39, no. 12, pp. 1611–1640, 2013.
- [3] N. Dintzner, A. Van Deursen, and M. Pinzger, “Analysing feature model changes using FMDiff”, 2015.
- [4] M. Hintermann, “Operating system components for an embedded Linux system”, *Technische Universitat Munchen*, 2007.
- [5] J. Brodtkin, *Linux has 2,000 new developers and gets 10,000 patches for each version*, Ars Technica, Ed., <http://arstechnica.com/information-technology/2015/02/linux-has-2000-new-developers-and-gets-10000-patches-for-each-version/>, 2015.
- [6] A. Hubaux, Y. Xiong, and K. Czarnecki, “A user survey of configuration challenges in Linux and eCos”, in *Proceedings of the Sixth International Workshop on Variability Modeling of Software-Intensive Systems*, ACM, 2012, pp. 149–155.
- [7] O. Koren, “A study of the Linux kernel evolution”, *ACM SIGOPS Operating Systems Review*, vol. 40, no. 2, pp. 110–112, 2006.
- [8] L. Passos, J. Padilla, T. Berger, S. Apel, K. Czarnecki, and M. T. Valente, “Feature scattering in the large: a longitudinal study of Linux kernel device drivers”, in *Proceedings of the 14th International Conference on Modularity*, ACM, 2015, pp. 81–92.
- [9] R. Lotufo, S. She, T. Berger, K. Czarnecki, and A. Wasowski, “Evolution of the Linux kernel variability model”, in *Software Product Lines: Going Beyond*, Springer, 2010, pp. 136–150.
- [10] *KernelProjects/kconfig-sat*, 2015. [Online]. Available: <http://kernelnewbies.org/KernelProjects/kconfig-sat>. [Accessed: 8-Feb-2016].
- [11] *Kconfig-sat*, 2016. [Online]. Available: <https://groups.google.com/forum/#forum/kconfig-sat>. [Accessed: 24-Apr-2016].
- [12] *KernelProjects/linux-sat*, 2016. [Online]. Available: <http://kernelnewbies.org/KernelProjects/linux-sat>. [Accessed: 24-Apr-2016].

- [13] K. Peffers, T. Tuunanen, C. E. Gengler, M. Rossi, W. Hui, V. Virtanen, and J. Bragge, “The design science research process: a model for producing and presenting information systems research”, in *Proceedings of the first international conference on design science research in information systems and technology (DESRIST 2006)*, 2006, pp. 83–106.
- [14] A. Hubaux *et al.*, “Feature-based configuration: collaborative, dependable, and controlled”, PhD thesis, FUNDP, 2012.
- [15] D. Batory, D. Benavides, and A. Ruiz-Cortes, “Automated analysis of feature models: challenges ahead”, *Communications of the ACM*, vol. 49, no. 12, pp. 45–47, 2006.
- [16] pure-systems GmbH, *Variant management with pure::variants*, 2006. [Online]. Available: <http://www.pure-systems.com/fileadmin/downloads/pv-whitepaper-en-04.pdf>. [Accessed: 14-Apr-2016].
- [17] BigLever Software, *Product line engineering solutions for systems and software*, 2015. [Online]. Available: [http://www.biglever.com/extras/BigLever\\_Solution\\_Brochure.pdf](http://www.biglever.com/extras/BigLever_Solution_Brochure.pdf). [Accessed: 14-Apr-2016].
- [18] eCos, *eCos user guide*, 2011. [Online]. Available: <http://ecos.sourceforge.org/docs-latest/user-guide/ecos-user-guide.html>. [Accessed: 14-Apr-2016].
- [19] J. Sincero and W. Schröder-Preikschat, “The Linux kernel configurator as a feature modeling tool.”, in *SPLC (2)*, Citeseer, 2008, pp. 257–260.
- [20] *Kconfig language*, 2016. [Online]. Available: <https://www.kernel.org/doc/Documentation/kbuild/kconfig-language.txt>. [Accessed: 13-Apr-2016].
- [21] S. El-Sharkawy, A. Krafczyk, and K. Schmid, “Analysing the Kconfig semantics and its analysis tools”, in *Proceedings of the 2015 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*, ACM, 2015, pp. 45–54.
- [22] *Conjunctive normal form*, 2016. [Online]. Available: [https://en.wikipedia.org/wiki/Conjunctive\\_normal\\_form](https://en.wikipedia.org/wiki/Conjunctive_normal_form). [Accessed: 14-Apr-2016].
- [23] *DIMACS challenge—satisfiability: suggested format*, 1993. [Online]. Available: <http://www.domagoj-babic.com/uploads/ResearchProjects/Spear/dimacs-cnf.pdf>. [Accessed: 30-May-2016].
- [24] C. Kaestner, *kconfigreader*, 2015. [Online]. Available: <https://github.com/ckaestne/kconfigreader>. [Accessed: 14-Apr-2016].
- [25] The VAMOS project, *undertaker*, 2016. [Online]. Available: <https://vamos.informatik.uni-erlangen.de/trac/undertaker>. [Accessed: 14-Apr-2016].
- [26] S. She and T. Berger, “Formal semantics of the Kconfig language”, *Technical note*, University of Waterloo, p. 24, 2010.
- [27] S. She, *LVAT*, 2013. [Online]. Available: <https://code.google.com/archive/p/linux-variability-analysis-tools/>. [Accessed: 14-Apr-2016].

- 
- [28] S. She, *exconfig*, 2013. [Online]. Available: <https://github.com/matachi/linux-variability-analysis-tools.exconfig>. [Accessed: 14-Apr-2016].
- [29] S. She, *exconfig extracts*, 2012. [Online]. Available: <https://github.com/matachi/linux-variability-analysis-tools.extracts>. [Accessed: 14-Apr-2016].
- [30] Y. Xiong, H. Zhang, A. Hubaux, S. She, J. Wang, and K. Czarnecki, “Range Fixes: interactive error resolution for software configuration”, *IEEE Transactions on Software Engineering*, 2014.
- [31] Y. Xiong and A. Hubaux, *RangeFix*, 2013. [Online]. Available: <https://github.com/matachi/rangeFix>. [Accessed: 14-Apr-2016].
- [32] Microsoft Research, *Z3*, 2016. [Online]. Available: <https://github.com/Z3Prover/z3>. [Accessed: 14-Apr-2016].
- [33] V. Nossum, *satconfig*, 2016. [Online]. Available: <https://github.com/vegard/linux-2.6/tree/v4.3+kconfig-sat>. [Accessed: 14-Apr-2016].
- [34] A. Biere and J. Kepler, *PicoSAT*, 2016. [Online]. Available: <http://fmv.jku.at/picosat/>. [Accessed: 14-Apr-2016].
- [35] K. Bak and K. Ali, “Improving usability of the Linux kernel configuration tools”, 2010.
- [36] C. Zengler and W. K uchlin, “Encoding the Linux kernel configuration in propositional logic”, in *Proceedings of the 19th European Conference on Artificial Intelligence (ECAI 2010) Workshop on Configuration*, Citeseer, vol. 2010, 2010, pp. 51–56.
- [37] M. Walch, R. Walter, and W. K uchlin, “Formal analysis of the Linux kernel configuration with SAT solving”, 2015.
- [38] R. Tartler, J. Sincero, W. Schr oder-Preikschat, and D. Lohmann, “Dead or alive: finding zombie features in the Linux kernel”, in *Proceedings of the First International Workshop on Feature-Oriented Software Development*, ACM, 2009, pp. 81–86.
- [39] A. Hubaux, D. Jannach, C. Drescher, L. Murta, T. M annist o, K. Czarnecki, P. Heymans, T. Nguyen, and M. Zanker, “Unifying software and product configuration: a research roadmap”, in *Proceedings of the Workshop on Configuration (ConfWS’12), Montpellier, France, 2012*, pp. 31–35.
- [40] E. K. Abbasi, A. Hubaux, M. Acher, Q. Boucher, and P. Heymans, “What’s in a web configurator? empirical results from 111 cases”, in *PRECISE-FUNDP, University of Namur*, 2012.
- [41] *Linux Kconfig-SAT integration wiki and mailing list*, 2015. [Online]. Available: [https://groups.google.com/d/topic/kconfig-sat/G6HA\\_3ecAQI/discussion](https://groups.google.com/d/topic/kconfig-sat/G6HA_3ecAQI/discussion). [Accessed: 28-Feb-2016].
- [42] A. Biere, *Fwd: Re: [kconfig-sat] Linux Kconfig-SAT integration wiki and mailing list*, 2015. [Online]. Available: [https://groups.google.com/d/msg/kconfig-sat/G6HA\\_3ecAQI/n71Nh1WmBQAJ](https://groups.google.com/d/msg/kconfig-sat/G6HA_3ecAQI/n71Nh1WmBQAJ). [Accessed: 14-Apr-2016].

- [43] *Tseytin transformation*, 2016. [Online]. Available: [https://en.wikipedia.org/wiki/Tseytin\\_transformation#Gate\\_Sub-expressions](https://en.wikipedia.org/wiki/Tseytin_transformation#Gate_Sub-expressions). [Accessed: 24-May-2016].
- [44] *De Morgan's laws*, 2016. [Online]. Available: [https://en.wikipedia.org/wiki/De\\_Morgan's\\_laws](https://en.wikipedia.org/wiki/De_Morgan's_laws). [Accessed: 6-May-2016].
- [45] *The SMT-LIBv2 language and tools: a tutorial*, 2013. [Online]. Available: <http://smtlib.github.io/jSMTLIB/SMTLIBTutorial.pdf>. [Accessed: 1-Mar-2016].
- [46] *Logical equivalence*, 2016. [Online]. Available: [https://en.wikipedia.org/wiki/Logical\\_equivalence](https://en.wikipedia.org/wiki/Logical_equivalence). [Accessed: 6-May-2016].
- [47] D. Jonsson, *Configuring Linux 2.6.32 using xconfig and RangeFix*, 2016. [Online]. Available: <https://www.youtube.com/watch?v=F8RZ8YpBeew>. [Accessed: 23-May-2016].
- [48] *Valgrind*, 2015. [Online]. Available: <http://valgrind.org/>. [Accessed: 30-May-2016].
- [49] D. Jonsson, *Editing the wiki to update my project status*, 2016. [Online]. Available: <https://groups.google.com/forum/#!msg/kconfig-sat/utCcD2R6sKU/2KXvqKG3HQAJ>. [Accessed: 2-Jun-2016].
- [50] *Karnaugh map*, 2016. [Online]. Available: [https://en.wikipedia.org/wiki/Karnaugh\\_map](https://en.wikipedia.org/wiki/Karnaugh_map). [Accessed: 2-Jun-2016].
- [51] *CommonMark*, 2016. [Online]. Available: <http://commonmark.org/>. [Accessed: 31-May-2016].

A

# Survey questions

## Configuring the Linux kernel

Hi,

We are a group of researchers from Chalmers University of Technology and TU Darmstadt, and we are part of the kconfig-sat project [1]. We are working on adding interactive dependency-resolution support to the Linux kernel configurator xconfig. We would be very thankful if you could take the time to fill out this short survey that will provide data to help us design and evaluate a better dependency-resolution mechanism. Filling out the survey should not take more than 5 minutes of your time.

It is possible to fill out the form without a Google account. No names, email-addresses or other identifying information will be collected or published.

Note that only the fields marked with \* are required to be filled out, however, we would of course appreciate if as many fields as possible are filled out.

Thank you for your time.

[1]: <http://kernelnewbies.org/KernelProjects/kconfig-sat>

\* Required

## Encountering configuration issues

---

1. Do you ever run into issues when trying to change a configuration option's value in xconfig/menuconfig during the Linux kernel configuration process? \*

*Mark only one oval.*

- Never
- Rarely
- Occasionally
- Frequently
- Don't know

2. Please explain your answer to the previous question, stating the issues you run into if applicable:

.....

.....

.....

.....

.....

## Help text's usefulness

---

3. How much help does the configuration option's help text provide when trying to change the current configuration to satisfy any missing dependencies needed to enable a particular option? \*

Mark only one oval.

- Not helpful at all
- Somewhat helpful
- Helpful
- Very helpful
- Other: .....

## Configuration method

---

4. When trying to enable a configuration option, please describe the method or tool you use to satisfy the option's dependencies: \*

.....  
.....  
.....  
.....  
.....

## Time required to change a configuration option's value

---

Please estimate in minutes (maximum, minimum, and average) the time it takes you to adjust a disabled configuration option's value when configuring the kernel.

5. Longest time it took you: \*

.....

6. Shortest time it took you: \*

.....

7. Typical time it takes you: \*

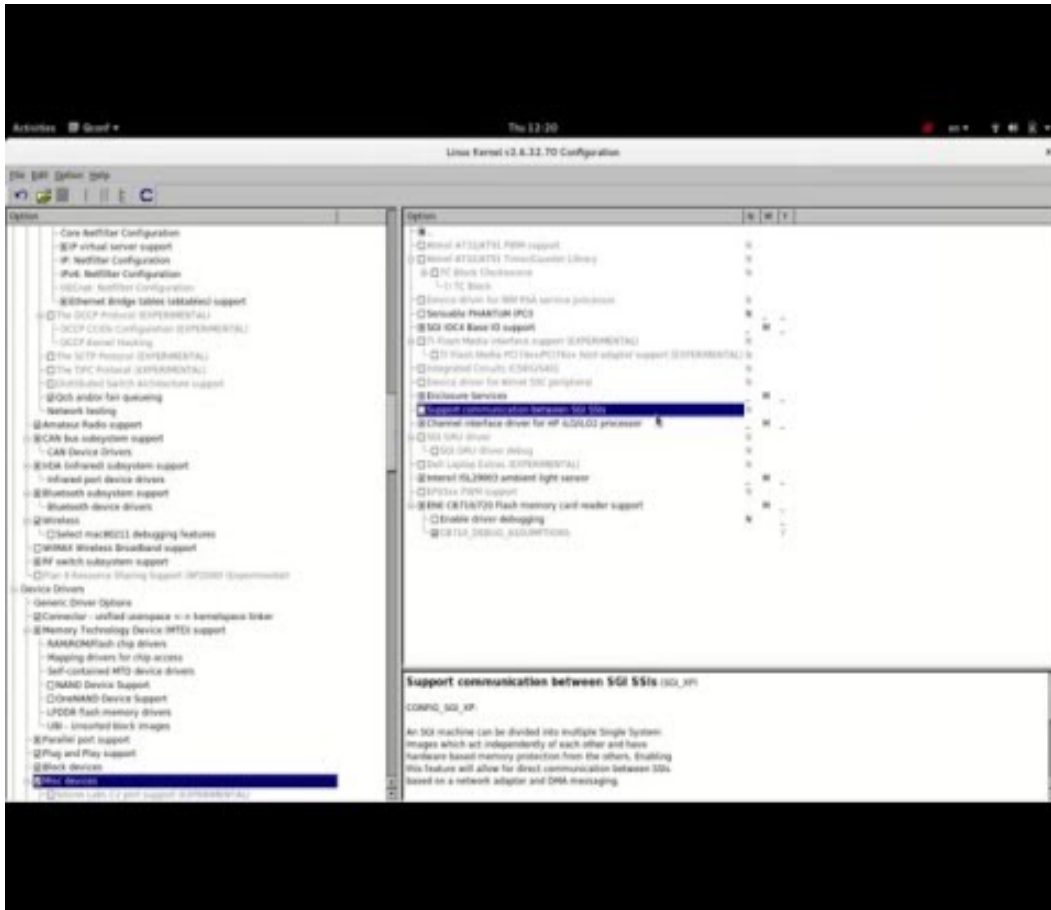
.....

## Configuration assistance

---

In the video below (45 seconds long) a modified version of xconfig is showcased. It is able to perform dependency resolution and display to the user what options need to be edited for the user to enable an option.

## Configuring the kernel using xconfig with dependency-resolution support



<http://youtube.com/watch?v=4oVzJMhn3Kw>

8. Would such a tool be beneficial for you? \*

Mark only one oval.

- Yes
- No
- In some scenarios



9. Can you describe a scenario where you would use such a tool?

.....  
.....  
.....  
.....  
.....

10. If not, why is such a tool not beneficial for you?

.....  
.....  
.....  
.....  
.....

---

Considering that the term "computation time" describes the time taken from the time you click "Calculate Fixes" until the list of fixes is displayed.

11. In your opinion, what is an ideal computation time? \*

.....

12. Since ideal scenarios do not always happen, what is the maximum amount of time you would be willing to wait for such a list of fixes?

.....

13. In the video, an option and its four dependencies are located and enabled within 2 minutes. Would you consider that an improvement over how you would perform a similar task? \*

*Mark only one oval.*

- Not an improvement
- Slight improvement
- Improvement
- Big improvement

14. Please provide any feedback you have about the tool (e.g., its functionality, layout, additional features you would like to see, or any other comments you might have).

.....  
.....  
.....  
.....  
.....

## Other

---

15. If you would be willing to be contacted for some follow-up questions, please provide us with your name and email address:

.....