The Kconfig Variability Framework as a Feature Model

Bachelor thesis of

Kaan Berk Yaman

at the Department of Informatics
KASTEL – Institute of Information Security and Dependability

Reviewer:	Prof. Dr. Ralf Reussner
Second reviewer:	Prof. Dr.-Ing. Anne Koziolek
Advisor:	M.Sc. Jan Wittler
Second advisor:	Dr. Christopher Gerking

13.12.2022 – 13.04.2023

I declare that I have developed and written the enclosed thesis completely by myself. I have submitted neither parts of nor the complete thesis as an examination elsewhere. I have not used any other than the aids that I have mentioned. I have marked all parts of the thesis that I have included from referenced literature, either in their original wording or paraphrasing their contents. This also applies to figures, sketches, images and similar depictions, as well as sources from the internet.

Karlsruhe, 31.03.2023

. . . .

(Kaan Berk Yaman)

Abstract

Variability in software is often managed using external tools. One such tool is Kconfig, which is utilised by the Linux kernel, a highly variable software project. Kconfig works with plain-text files that define the variability structure of the underlying software project. These plain-text files are called Kconfig files. Analysis of Kconfig files can provide meaningful insights for developers. Feature-oriented programming (FOP) is also used to manage variability in software. The variability structure of a software project is defined using feature models in FOP. There are tools for analysing feature models. However, it is not possible to use these tools for the analysis of Kconfig files, as currently no transformation between Kconfig files and feature models exist. In this thesis, we present a method to correctly transform Kconfig files into feature models, so that Kconfig files can be analysed with tools meant for feature models. We verify the correctness of our transformation by automatic and manual means. Our method transforms selected Kconfig files with non-trivial structure successfully into semantically equivalent feature models.

Zusammenfassung

Zur einfachen Handhabung von Softwarevariabilität werden oft externe Werkzeuge eingesetzt. Ein solches Werkzeug ist Kconfig, welches vom Linux-Kernel zur Erstellung von konkreten Softwarekonfigurationen benutzt wird. Kconfig arbeitet mit Textdateien, in denen die Variabilitätsstruktur des zugehörigen Softwareprojekts definiert wird. Diese Dateien werden oft als Kconfig-Dateien bezeichnet. Kconfig-Dateien können analysiert werden, um Probleme in der Variabilitätsstruktur festzustellen. Feature-orientierte Programmierung (FOP) wird auch zur besseren Handhabung von Softwarevariabilität eingesetzt. Die Variabilitätsstruktur eines Softwareprojekts wird im Umfang von FOP in einem sogenannten Feature-Modell dargestellt. Es gibt Werkzeuge, welche zur Analyse von Feature-Modellen verwendet werden können. Diese kann man jedoch nicht zur Analyse von Kconfig-Dateien nutzen, da bisher eine Transformation zwischen Kconfig-Dateien und Feature-Modellen fehlt. In dieser Arbeit stellen wir eine Methodik zur korrekten Transformation von Kconfig-Dateien in Feature-Modelle vor, sodass Werkzeuge zur Feature-Modell-Analyse auch auf Kconfig-Dateien angewandt werden können. Wir evaluieren die Korrektheit unserer Transformation mit automatischen und manuellen Vorgehen. Unsere Methodik kann ausgewählte Kconfig-Dateien mit nichttrivialer Struktur erfolgreich in semantisch äquivalente Feature-Modelle überführen.

Contents

Abstract i

Zusammenfassung iii

1. Introduction 1

2. Foundations 3
- 2.1. Kconfig . . . 3
- 2.2. Feature-oriented programming . . . 5

3. Related Work 9
- 3.1. Tools that transform Kconfig files . . . 9
- 3.2. Feature-oriented programming and Kconfig . . . 9

4. Concept 11
- 4.1. Scope of the transformations . . . 11
- 4.2. Transformation rules . . . 13
  - 4.2.1. Boolean symbols . . . 13
  - 4.2.2. Tristate symbols . . . 16
  - 4.2.3. Mixed-type dependencies . . . 18
  - 4.2.4. Reverse dependencies . . . 21
  - 4.2.5. Choice blocks . . . 21
  - 4.2.6. Menus . . . 25

5. Implementation 31
- 5.1. Implementation details . . . 31
- 5.2. Challenges during implementation . . . 31

6. Evaluation 33
- 6.1. Methods of evaluation . . . 33
- 6.2. Human readability of the created feature models . . . 34
- 6.3. Converting feature model configurations to Kconfig configurations . . . 34
- 6.4. Results . . . 35
- 6.5. Interpretation . . . 38
- 6.6. Threats to validity . . . 38
  - 6.6.1. External validity . . . 38
  - 6.6.2. Internal validity . . . 38

7. Conclusion **41**
7.1. Benefits 41
7.2. Future Work 41

Bibliography **45**

A. Appendix **51**

1. Introduction

Variability in software is becoming increasingly more important as computers become more ubiquitous. Different target platforms have different constraints for software developers to work around and moreover to consider whilst implementing new functionality [13]. External tools are often utilised to manage software variability [7].

The Kconfig framework has been initially developed for the Linux kernel but it has since then become a generic tool for managing software variability [16], being used in projects such as ZephyrOS [41] and NuttX [1]. Kconfig manages variability over preprocessor variables, also called *configuration symbols*, which are used to exclude or include source code whilst building software [35, 26]. A Kconfig file contains definitions of configuration symbols that occur in the underlying source code and how these configuration symbols are related to each other, such as dependencies between configuration symbols. The Kconfig framework additionally offers many different graphical and non-graphical interfaces [22] for the end user to utilise for the creation of concrete software configurations. The main feat of Kconfig is the fact that software configurations created using any of the front-ends offered by Kconfig are valid, as the framework considers the dependencies between configuration symbols and ensures that all selected configuration symbols have their dependencies properly resolved [21].

Because Kconfig files can get very complex, many tools have been developed to analyse Kconfig files and detect configuration defects. Examples to this are `undertaker` and the Linux Variability Analysis Tools [31].

Another tool, or so to say methodology, to manage software variability is the use of feature-oriented programming (FOP). In FOP, concrete software products are seen as a group of software features. The dependencies and relations between the features that are provided by a certain software project are described in a *feature model* [2]. A feature model implies a software product line (SPL), a set of software products that share a common set of managed features, these features being those that are defined in the feature model. If a concrete software product in a software product line is composed according to the underlying feature model, it will be valid, assuming the correctness of the feature model [19].

There are already many well-established tools for feature model analysis and manipulation [11]. One such tool is FeatureIDE [25, 39], which can be used to detect inconsistencies in feature models.

Although there is no mention of feature-oriented design in the official Kconfig documentation [21], the similarities between the Kconfig framework and the feature-oriented approach are clear; there are many works that argue that the Kconfig framework allows the definition of software product lines [29, 36] and Kconfig modules effectively correspond to features within a feature model [34, 10]. There is currently no way to use tools that

work with FOP feature models (e.g. FeatureIDE) to analyse Kconfig files, as a correct and well-defined transformation of Kconfig files to such feature models does not exist.

We want to close the gap between Kconfig files and feature models so that the tools that are used for analysing feature models can be also used on Kconfig files. Analysis of Kconfig files through automated tools is a hot topic in Linux kernel development and bringing feature models and Kconfig files together with a correct transformation of Kconfig files into feature models would allow developers using the Kconfig framework to seamlessly integrate feature model analysis tools into their workflow.

Our thesis aims to presents a method to transform Kconfig files into feature models so that the resulting feature models and the underlying Kconfig files are logically equivalent, i.e. every non-solution to the feature model corresponds to an invalid configuration of the Kconfig file, whilst every solution to the feature model corresponds to a valid configuration for the Kconfig file. We also aim to make the menu structure present in a Kconfig file, which we describe in detail in the next chapter, recognizable in the resulting feature model; hence it is our goal to develop a transformation scheme for Kconfig files that ensures preservation of *semantics* and *structure*.

This thesis is structured as follows: In the second chapter, we introduce the terminology around Kconfig and feature-oriented programming. In the third chapter, we give an overview of related work; not only papers but also various open-source projects that have relevancy for our thesis. Chapter 4 introduces the aforementioned method: In this chapter, we talk about important design decisions and present many working examples to showcase different transformation rules we have established. In chapter 5, we talk about Kfeature, a tool we have developed that implements the transformations introduced in chapter 4. The fifth chapter also discusses certain problems that have arisen during the implementation of the Kfeature tool. In chapter 6, we evaluate the correctness of our transformation rules. Chapter 7 concludes the thesis, with references to future work that may build upon our contribution.

2. Foundations

In this chapter, we give a brief overview of the Kconfig variability framework and introduce the basics of feature-oriented design/programming, with specific focus on what a feature model is and what structure a feature model has.

2.1. Kconfig

The Kconfig framework was initially designed and developed to manage variability in the Linux kernel: Over time, many different optional features and modules were added to the Linux code tree, and it was no longer meaningful to compile and include every single module whilst building the kernel [15]. Instead of compiling all modules, users can use one of the many graphical interfaces offered by Kconfig to select the modules and features they want to compile and include in the built kernel.

Because the Linux kernel is written in C, including and excluding of modules/features is done over preprocessor variables [35]: Code that belongs to a certain module is excluded if the preprocessor variable that corresponds this module is not set. An example of this can be seen in figure 1, the preprocessor variable `CONFIG_IP_PNP_DHCP` decides if the field `dhcp_client_identifier` gets initialized or not.

The Kconfig tool can parse and process Kconfig files, which contain definitions of configuration symbols. A configuration symbol is a preprocessor variable that excludes or includes a certain feature or module (as already shown in figure 1). Configuration symbols can depend on other configuration symbols; this is relevant for cases where a module or feature utilises another module or feature, so that these must also be built if the latter is included in the kernel [21]. For the example given in figure 1, the configuration symbol `IP_PNP_DHCP` depends on `IP_PNP`.

The syntax used in Kconfig files is not formally defined (there is a parser grammar [30] that is offered as-is without any analysis of the syntax itself), but certain code constructs are described in the official Linux kernel documentation [21]:

- **Configuration symbols.** These are defined with the `config` keyword. The header of a configuration symbol may contain multiple options, such as `depends on`, for defining dependencies between configuration symbols. Configuration symbols have types, such as `string`, `tristate` and `boolean`, although 95% of configuration symbols in the Linux kernel are either of `tristate` or `boolean` type [28]. `tristate` configuration symbols can assume three different values: `t`, `f` and `m`. In the Linux kernel, such `tristate` configuration symbols are used to manage features/modules that can be included in the built kernel without being explicitly active (`m` stands for *module*, i.e. build module but do not activate it) [9]. Example configuration symbol definitions with dependencies can be seen in figure 28.

```
1 #if defined(CONFIG_IP_PNP_DHCP)
2 static char dhcp_client_identifier[253] __initdata;
3 #endif
```

Listing 1: Extract from [17]

```
1  config IP_PNP_DHCP
2    bool "IP: DHCP support"
3    depends on IP_PNP
4    help
5      If you want your Linux box to mount its whole root file system (the
6      one containing the directory /) from some other computer over the
7      net via NFS and you want the IP address of your computer to be
8      discovered automatically at boot time using the DHCP protocol (a
9      special protocol designed for doing this job), say Y here. In case
10     the boot ROM of your network card was designed for booting Linux and
11     does DHCP itself, providing all necessary information on the kernel
12     command line, you can say N here.
13
14     If unsure, say Y. Note that if you want to use DHCP, a DHCP server
15     must be operating on your network.  Read
16     <file:Documentation/admin-guide/nfs/nfsroot.rst> for details.
```

Listing 2: Extract from [20]

Figure 1.: A preprocessor directive with the configuration symbol `IP_PNP_DHCP`. `IP_PNP_DHCP` is also defined in the respective Kconfig file. The `CONFIG_` prefix is used to discern between regular variables and configuration symbols [22].

- **Menu blocks.** A menu block is effectively a nested Kconfig file. The header of a menu block definition may contain dependencies, in this case the menu block is only visible when its dependencies are satisfied. An example for a menu block can be seen in figure 34: “`Menu block M`” depends on `SYMBOL_A` and a choice block and a tristate configuration symbol.
- **Choice blocks.** Configuration symbols contained within a choice block are mutually exclusive. Choice blocks may depend on other configuration symbols. An example of a choice block can be seen in figure 30: `CHOICE_D` contains two configuration symbols and depends on `SYMBOL_Y`.
- **If blocks.** The condition of the if block is appended as a dependency to all configuration symbols that are in the aforementioned if block.
- **Combining multiple Kconfig files.** The `source` keyword can be used to refer to external Kconfig files whilst parsing.

The list above is not exhaustive; the specifics of the individual constructs are further explored in chapter 4.

End users and developers can use the various interfaces [22] provided by Kconfig to construct configurations in accordance with the constaints defined in the underlying Kconfig file. One of these interfaces is `menuconfig`, which generates a graphical menu interface with the configuration symbols defined in the Kconfig file it was called on. How this menu is structured depends on multiple factors:

- The structure of the menu can be defined *explicitly* with menu blocks. In this case, everything contained in the menu block is hidden behind a submenu (see figure 2).
- Beyond normal configuration symbols, there exists so-called `menuconfig` symbols. Configuration symbols that depend on a `menuconfig` symbol are hidden behind the menu entry of the `menuconfig` symbol (see figure 3).
- If a configuration symbol depends on another configuration symbol, the menu entry of the depender is only visible when the dependee is selected (see figure 4).

```
1 menu "Menu block M"
2
3   config SYMBOL_B
4     bool "Configuration symbol B"
5
6   config SYMBOL_C
7     bool "Configuration symbol C"
8 endmenu
```

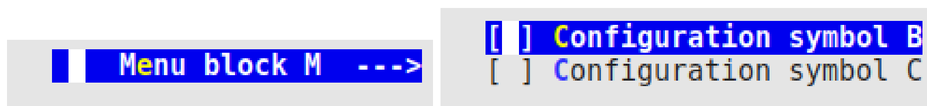


Figure 2.: The configuration symbols contained in the menu block are shown when the submenu is entered. The submenu itself appears as an entry in the main menu.

Once a configuration is generated, the resulting `.config` file (a plain textfile containing variable assignments) is processed further by Kbuild. Kbuild is an adapted version of GNU make [8]. We will not explain in detail how Kbuild works, as the focus of this thesis is exclusively on Kconfig.

2.2. Feature-oriented programming

Feature-oriented programming considers the feature as the fundamental building block of software. A concrete software product is a certain combination of features. What a feature exactly is, is not always well-defined, but in this thesis, we take the definition used by Kun Chen et al. [24] as reference:

> “A feature describes a product characteristic from user or customer views.”

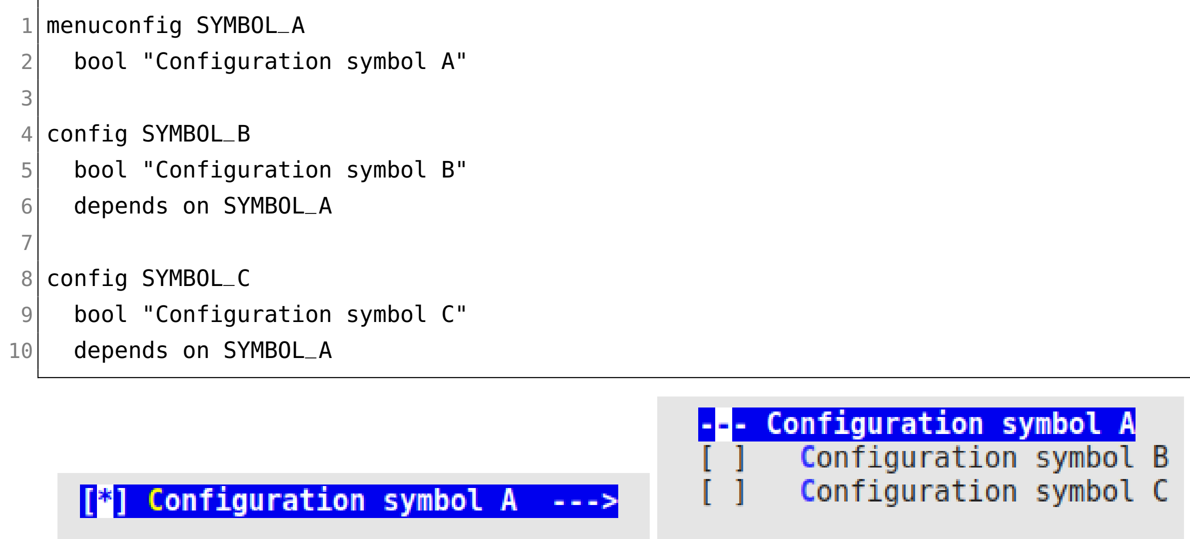


Figure 3.: Dependers of a `menuconfig` symbol are contained in a submenu hidden behind the menu entry of the `menuconfig` symbol.

The relationships between features are defined in a *feature model* [19]: A selection of features represent a valid software product if they fulfil the constraints presented by the feature model. The structure of a feature model is not standardized; although it is clear that feature models should have a tree-like structure. In this thesis, we will restrict ourselves to feature models as defined and used by Leich et al. [25]. Several authors [6, 5] call this family or "tradition" of feature models "FODA-like" or "FODA feature models", wherein FODA refers to Feature-Oriented Domain Analysis [18], a predecessor of FOP.

With this consideration, we would like to introduce some structures that occur in feature models:

- **Root feature.** Every feature model must have a root feature.
- **Parent-child relationships.** A child feature can only be selected if its parent feature is also selected. A feature can only have one parent feature.
- **Or groups.** The children of a feature may be put in an *or* group. In this case, at least one of the children must be selected if the parent feature is selected.
- **Alt groups.** The children of a feature may be put in an *alt* (alternative) group. In this case, exactly one of the children must be selected if the parent feature is selected.
- **Feature options.** Features may be optional, mandatory or abstract. Abstract features are used to bring the feature model in a certain structure or form, but they have no relevancy in the implementation level [38].
- **Cross-tree constraints.** A cross-tree constraint is a logical expression containing feature names as variables. A valid software product (= selection of features) must satisfy all cross-tree constraints.

```
config SYMBOL_A
  bool "Configuration symbol A"

config SYMBOL_B
  bool "Configuration symbol B"
  depends on SYMBOL_A

config SYMBOL_C
  bool "Configuration symbol C"
```

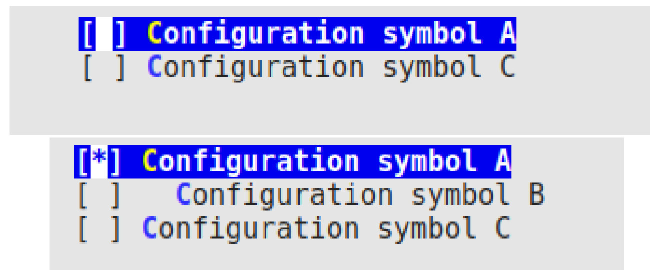


Figure 4.: The menu entry of the depender is only visible when the dependee is selected. Additionally the entry of the depender is placed right below the entry of the dependee; indented, graphically hinting a hierarchy between the two configuration symbols.

All of the constructs listed above were used in the feature model given in figure 5. Here, *Car* is the root feature. The feature *Motor* is mandatory and the children of *Motor* are in an or group, so that a car can have either an *ElectricMotor* or a *GasMotor* (or both, in that case, it is an hybrid car). The feature *Gearbox* is also mandatory but the children of *Gearbox* are in an alt group, so that a car can either have *Automatic* or *Manual* transmission, but not both. Additionally, there is a cross-tree constraint: If *TowHitch* is selected, *GasMotor* must also be selected. A possible solution to this feature model would be [*Car*, *Motor*, *ElectricMotor*, *Gearbox*, *Automatic*, *Chassis*], as this selection of features fulfil all the constraints given by the feature model.

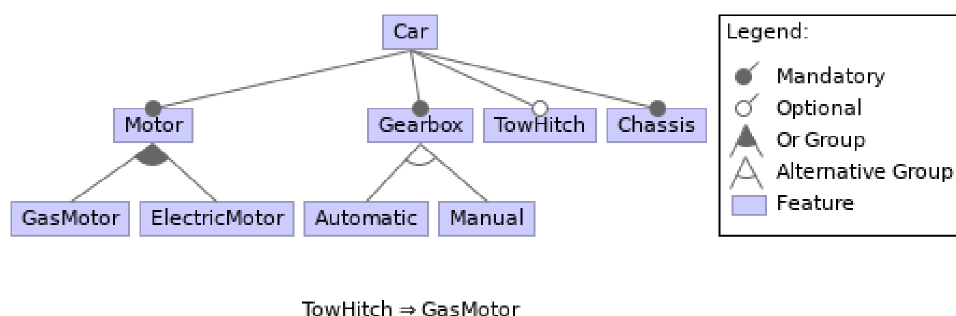


Figure 5.: A feature model for cars. Example adapted from Apel and Kästner [2]. Graphic generated with FeatureIDE.

3. Related Work

In this chapter, we give an overview of related work in areas relevant to this thesis. These works are the literary fundament that we will build upon going forward.

3.1. Tools that transform Kconfig files

Due to the lack of a formal language definition and a Kconfig ABI, tools that work with Kconfig files usually transform the Kconfig files they want to process. Seldom a Kconfig file is used as-is.

One such tool that transforms the Kconfig files it processes is `undertaker` [37]. `undertaker` parses Kconfig files and generates SAT problems using the Linux kernel source code to find "dead" code blocks (code blocks that can never appear in a valid kernel configuration) and "undead" code blocks (code blocks that appear in all valid kernel configurations) in the kernel code tree (hence the name `undertaker`). The results returned by `undertaker` can then be used to remove unnecessary configuration symbols and cut out redundant preprocessor directives in source code. `undertaker` utilises a well-defined formal transformation of Kconfig files to expressions in propositional logic. A similar transformation is done with the preprocessor directives in the source code and then the resulting logical expressions are compared to detect differences between the constraints defined in the Kconfig file and the constraints inferred by the preprocessor directives in code. Tartler et al. [37] define the approach used by `undertaker` as follows: "The variability constraints defined by both spaces are extracted separately into propositional formulas, which are then examined against each other to find inconsistencies we call configurability defects."

Fernandez-Amoros et al. [12] provide a method for Kconfig-to-logic translation: This translation is general-purpose, the paper's goal is similiar to ours: To bridge the gap between "logic engines" and Kconfig files. Oh et al. [28] use a related transformation and additionally introduce the tool `kmax`, which uses the expressions that are generated through the transformation to find out if the Kconfig file contains (or moreover defines) unsatisfiable constraints by invoking an external SAT solver. Transformation of tristate symbols seems to be a challenge for both: Aforementioned Fernandez-Amoros et al. [12] ignore tristate symbols and give no translation for these. Oh et al. [28] underapproximate tristate symbols by reducing them two states (`m` and `t`); prioritising tool performance over semantic correctness.

3.2. Feature-oriented programming and Kconfig

Many authors [28, 29, 27, 6] have already described (moreover recognized) the Linux kernel and its variants as a software product line, hence the link between Kconfig and

feature models is not new; although the official Kconfig documentation does not use any FOP terminology.

Sincero and Schröder-Preikschat [34] define a set of rudimentary mappings that can be used to transform certain feature models structures into Kconfig code. The semantic equivalence of the resulting Kconfig code and the initial feature model isn't verified within the scope of [34], although it is implied that the mappings can be used inversely to transform Kconfig code into feature models. Ultimately, Sincero and Schröder-Preikschat [34] argue that the Kconfig framework can be used as a feature modelling tool. She et al. [32] also suggest several "simplified mappings" between Kconfig and FOP concepts and even transform a small Kconfig file snippet to an equivalent feature model, but they do not give any instructions to reproduce this transformation for arbitrary Kconfig files. The mappings provided by [34, 32] can however be utilised whilst we develop our own transformation rules.

Dintzner, Deursen, and Pinzger [10] introduce the tool `fmdiff`: `fmdiff` can be used to compute or detect the effects of changing Kconfig files (eg. adding, removing and modifying of configuration symbols) on individual kernel variants. `fmdiff` does by this transforming the initial and the changed Kconfig file into feature models and then comparing the created feature models (using the Eclipse Modelling Framework and the EMF Compare tool). The paper unfortunately does not describe the transformation process in detail, and additionally, it uses a feature model structure/definition that greatly differs from the typical FODA-like notation, so that we cannot build upon the transformations used by `fmdiff` whilst working out our own transformation rules.

4. Concept

In this chapter, we document the considerations we had during the conception of this thesis and constructively introduce the transformation rules we have concieved.

4.1. Scope of the transformations

This thesis aims to present a method for the transformation of Kconfig files into feature models. The word "method" here refers to a group of atomic transformations that can be applied respectively to create a feature model that corresponds to the source Kconfig file. But what does "correspond" mean here exactly? Before we go further, we would like present a three-layer decomposition of Kconfig files:

- **The structural layer.** These are the Kconfig constructs that affect the menu interface generated by `menuconfig`.
- **The constraint layer.** These are the Kconfig constructs that define the constraints that need to be fulfilled by the configurations that are generated using a given Kconfig file. This is so to say the semantic content of the Kconfig file. All constructs in the constraint layer are also in the structural layer, but this does not apply in the other direction (eg. menu blocks without dependencies, help text in configuration symbols).
- **The template/default value layer.** These are the Kconfig constructs that suggest a certain default configuration for a given Kconfig file. Examples to such constructs: `imply` and `default` options in configuration symbols (both described in detail later in this section).

Our transformations will attempt to preserve the semantic and the structural content of Kconfig files, so that:

1. The resulting feature model contains the configuration constraints defined in the Kconfig file we have transformed.
2. The menu structure implied by the Kconfig file should be recognizable in the resulting feature model, as in, the feature model should be human-readable; we do not aim to model all the semantics of a Kconfig file via cross-tree constraints.

Before we give in-depth descriptions of the transformation rules, it is important to talk about *what* we aim to transform. There are certain aspects of a Kconfig file that we have decided to ignore; as in, we do not see these aspects of the Kconfig language relevant for

the transformation rules (in accordance to the two goals we have stated above) and/or due to time constraints, we have chosen to make certain assumptions about the Kconfig files we aim to transform:

- Kconfig allows configuration symbols to have default values [21]: This is done using the `default` keyword within symbol definitions. When the Kconfig configuration interface (e.g. `menuconfig`) is invoked, the default values of the configuration symbols are assumed, but this does not restrict the user from changing these values. The default value for a configuration symbol is to be understood as a *suggestion*. Default values do not contribute to the constraints defined by a Kconfig file. Hence our transformation rules ignore default values.

- In the Kconfig files found in the Linux kernel, there exists a configuration symbol called `MODULES` which controls module support: Tristate symbols can assume the value `m` only when `MODULES` is set to true, i.e. setting `MODULES` to `f` converts all tristate symbols into boolean symbols. For ease of transformation, we assume that module support is always given.

- The dependency type `imply`, which corresponds to a weak reverse dependency [21], is ignored. `imply` effectively defines a conditional default value, and we have already made the decision to ignore default values.

- The `help` keyword in configuration symbol definitions is ignored. Help text has no relevancy for the semantic content of a Kconfig file.

- Configuration symbols that are not of `boolean` or `tristate` type are ignored. This means we will ignore `string`, `hex` and `int` configuration symbols. These configuration symbols (if no range is set) can assume infinitely many different values. The naive approach of representing every state of such a configuration symbol as a seperate feature does not work, as that would create an infinitely large feature model. We would suggest that there is no possibility to transform such configuration symbols into features, but we cannot formally prove our statement. Due to time constraints, we choose to focus more on transformations we deem possible. Beyond this, as already mentioned in the foundations chapter, `boolean` and `tristate` configuration symbols make up the vast majority (about 95%) of all configuration symbols (in case of the Linux kernel), so that limiting ourselves to these two types does not (greatly) endanger the external validity of our transformations.

- Composite dependencies (dependencies expression that are not single symbols) are limited to true multiple dependencies (multiple non-composite dependency options in a configuration symbol definition). This is done to limit the scope of this thesis. Initially, we have attempted to work out a transformation rule for arbitrary dependency expressions, but ultimately the rule could not be implemented, nor be subsequently evaluated, so that we have scrapped it.

- We do not provide a transformation rule for `if` blocks. `if` blocks are used to group up dependencies, e.g. when two configuration symbols A, B depend on the same

configuration symbol *C*, they can be put in an `if` block, i.e. `if C ... endif`. Such an `if` block is more of a syntactical shorthand, so that the absence of a transformation rule for `if` blocks shouldn't limit the extent of our method.

Now that we have defined the scope of our transformations, we continue with the transformation rules we have developed in the next section.

4.2. Transformation rules

In this section, we introduce the eight transformation rules we have developed in extent of this thesis. For each rule, we first document the considerations we had whilst establishing the respective rule and motivate the transformation in a constructive manner, so that the reader can better understand the design decisions we have made in scope of each transformation rule.

4.2.1. Boolean symbols

Individual boolean symbols can be set to true or false. Hence boolean configuration symbols are transformed to features. This is one of the mappings already suggested by She et al. [32].

A feature model must however have a single root feature [2]. We have come up with two different approaches to ensure that a root feature is always present:

- Each transformation rule assumes that a mandatory feature called "Kconfig" already exists in the target feature model. Every feature (before considering dependencies) depends on "Kconfig", consequently configuration symbols without any further dependencies are child features of "Kconfig".
- Some Kconfig files might contain a `mainmenu` entry [21]. This can be seen as the root of a Kconfig file. This `mainmenu` entry is hence transformed to a mandatory feature that acts as the root of the resulting feature model.

Because not all Kconfig files have a `mainmenu` entry, we have chosen the first approach going forward: All transformation rules presented in the following subsections make the assumption that a mandatory root feature called "Kconfig" exists in the target feature model.

4.2.1.1. Dependencies between boolean symbols

In the official Kconfig documentation, a dependency between two configuration symbols is described as an "upper bound on the depender" [21]. Within the context of two boolean symbols, when symbol *A* depends on *B*, the value of *A* is bounded above by the value of B. As boolean symbols can only assume the values `t` and `f`, this dependency implies the following value matrix:

Value of symbol A	Value of symbol B	Dependency satisified?
f	f	yes
f	t	yes
t	f	no
t	t	yes

This value matrix corresponds to the truth table of the logical expression $A \implies B$. In this case, a dependency between two boolean symbols is transformed to a child-parent relationship between the features that correspond to the boolean symbols: Feature `A` is an optional child of feature `B`. The reverse transformation (from an optional parent-child relationship between two features to two configuration symbols with a dependency between them) has already been proposed by Sincero and Schröder-Preikschat [34].

4.2.1.2. Multiple dependencies between boolean symbols

In case of multiple dependencies, it is important to mention that we *must* preserve the tree structure of the feature model. A feature cannot have multiple parents, hence the transformation we have described in the previous paragraph cannot be used for boolean configuration symbols with multiple dependencies. If A depends on B and C, there are several approaches to modelling this dependency within the context of the target feature model:

- The multiple dependency can be transformed to a cross-tree constraint. In this case, no structural changes must be done to the feature model to represent the dependency.
- The multiple dependency can be broken up into individual dependencies. In this case, one of the dependencies is structurally represented (as in, one of the dependencies is transformed into a child-parent relation in the feature model) whilst the remaining dependencies are represented as cross-tree constraints.

Both of these approaches have their advantages and disadvantages:

- One of the goals of our transformation is human-readability. Making the assumption that dependencies that are structurally represented are easier to comprehend than those that are "hidden" behind cross-tree constraints, it is preferable to transform dependencies to structural relationships as much as it is possible to do so. With this consideration, it is not feasible to transform a multiple dependency into a cross-tree constraint without further inspection of the dependency.
- When a composite dependency is broken up into its individual dependencies (which should be always possible in case of true multiple dependencies), a heuristic must be used to decide which of these sub-dependencies should be transformed to a structural relationship.

We have hence decided to break up multiple dependencies into their atomic parts and use an heuristic that prioritizes human-readability to choose the partial dependency we should represent structurally. In this case we have chosen to use a heuristic that attempts

to avoid branch dominance, so that the sub-dependency with the dependee with the least depth is represented structurally; i.e. the depender becomes the child feature of the dependee with the least depth. This heuristic should help increase human-readability by distributing nodes evenly, which is often proposed as a metric of graph readability [4].

Another metric of human-readability, or moreover a graphical property that should increase the readability of a graph, is the clustering of related nodes [4]. In this case, if a certain feature has many children, this makes it evident to the observer of the graph that this feature is a *gateway* feature. This same argument can be presented for branch depth: If a certain branch is particularly deep, this makes it evident to the observer that the root feature of this branch is a gateway feature, so to say, this branch represents many transitive dependencies.

In this case, we make a compromise: We allow clusters to be formed naturally in breadth, but we try distribute nodes evenly in depth by making the depender the child of the dependee with the least depth, representing further dependencies as cross-tree constraints. This heuristic is affected by menus, this is discussed in-depth in section 4.2.6.

The rule for transforming boolean symbols is as follows:

Rule 1: Whilst transforming Kconfig files into feature models, transform a boolean symbol A as follows:

1. Create a feature called `A` and add this as a child of the root feature of the feature model.
2. See respective sub-rules if A has a dependency.

Rule 1.1: If A depends on another boolean symbol B:

1. Transform B according to rule 1. If A needs to transformed during the transformation of B, abort transformation; the Kconfig file contains a dependency loop.
2. Make feature `A` child of feature `B`.

Rule 1.2: If A depends on multiple boolean symbols $B_1, \ldots, B_i$:

1. Transform $B_1, \ldots, B_i$ according to rule 1. If A needs to be transformed during any of these transformations, abort transformation; the Kconfig file contains a dependency loop.
2. Find B_j with $B_j = min(d(B_k)), k \in \{1, \ldots, i\}$. d here is the depth function. If B_j isn't unique, choose the B_j with the lowest index.
3. Make feature `A` child of feature `Bj`, which corresponds to the aforementioned boolean symbol B_j (see rule 1).
4. For all B_l with $l \neq j, l \in \{1, \ldots, j\}$, add a cross-tree constraint: `A` implies `Bl`.

An example transformation using this rule is given in figure 6.

4.2.2. Tristate symbols

A tristate configuration symbol can assume three different values: If the tristate symbol is not selected, it assumes the value `f`. A tristate symbol can be selected as a module (which corresponds to the value `m`) or can be selected as an *active* module (this corresponds to the value `t`).

Features in a feature model have a strictly two-state nature: A feature is either selected or not selected. Hence in the case of tristate symbols we cannot simply replicate the transformation we have used for boolean symbols. The transformation rule for tristate symbols must be a one-to-many mapping.

It is important to mention that the transformation rule for tristate symbols will directly affect the transformation rules involving dependencies with tristate symbols. Hence we try a conformist approach: We first investigate the structure of dependencies with tristate symbols and try to find an adequate transformation rule for tristate symbols themselves in a way that the transformation rules for the dependencies are correct.

We have already established the fact that a dependency is an upper bound relationship between two configuration symbols. A dependency between two tristate symbols A and B (A depends on B) has the following value matrix:

Value of symbol A	Value of symbol B	Dependency satisified?
f	f	yes
f	m	yes
f	t	yes
m	f	no
m	m	yes
m	t	yes
t	f	no
t	m	no
t	t	yes

Let us define this dependency formally using second-order logic:

Considering two predicate variables A and B, we define the unary relations F, M and T. The predicate variables A and B are in these relations when the tristate symbols that correspond to them assume the values `f`, `m` and `t` respectively. We additionally define the binary relation D, which holds true for $D(A, B)$ when the tristate symbol corresponding to the predicate variable A depends on the tristate symbol corresponding to the predicate variable B. The following formulae must hence hold true:

$$\forall A, B : D(A, B) \implies (M(A) \implies M(B) \lor T(B)) \qquad \text{(I)}$$

(if A is set to `m`, B has to be either set to `t` or `m`)

$$\forall A, B : D(A, B) \implies (T(A) \implies T(B)) \qquad \text{(II)}$$

(if A is set to `t`, B has to be set to `t` as well)

We can take these formulas as the fundament of our transformation. For a tristate symbol A, let us introduce two features in our target feature model: `A_m`, which corresponds to

```
1  config SYMBOL_A
2    bool "Config symbol A"
3
4  config SYMBOL_B
5    bool "Config symbol B"
6    depends on SYMBOL_A
7
8  config SYMBOL_C
9    bool "Config symbol C"
10   depends on SYMBOL_B
11
12 config SYMBOL_X
13   bool "Config symbol X"
14
15 config SYMBOL_Y
16   bool "Config symbol Y"
17   depends on SYMBOL_X
18   depends on SYMBOL_C
```

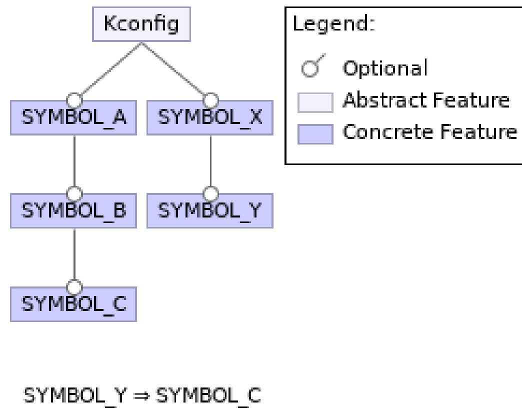


Figure 6.: Example transformation of the given Kconfig file. All configuration symbols were transformed using rule 1. The multiple dependency of `SYMBOL_Y` was handled by rule 1.2; `SYMBOL_Y` is here the child feature `SYMBOL_X` as it has less depth than `SYMBOL_C`, which was recognized correctly according to the heuristic we have defined in the respective sub-rule. All other (singular) dependencies were handled according to rule 1.1 (`SYMBOL_B`, `SYMBOL_C`).

the formula $M(A)$; when `A_m` is selected, symbol A assumes the value `m`, and `A_t`, which corresponds to the formula $T(A)$. It is important that `A_t` and `A_m` are not selected at the same time, as a tristate symbol can either assume the value `m` or `t`. There are two ways to model this exclusivity:

- We introduce an abstract parent feature `A` and add `A_m` and `A_t` as the children of this parent feature, so that if `A` is selected, either `A_m` or `A_t` must be selected (but not both). This is done with a mandatory XOR relation (alt group) between the parent and the children features. If this approach is used, (I) can be modelled as a cross-tree constraint between `A_m` and `B`. It is important that `A` is an abstract feature, as `A` itself does not correspond to a configuration symbol.
- We add a cross-tree constraint between `A_m` and `A_t`, i.e. `A_m` excludes `A_t` and vice versa. If this approach is used, (I) must be modelled as a composite cross-tree constraint, i.e. `A_m` implies `B_t` or `B_m`.

Preferring the approach that minimises the amount of semantics hidden behind cross-tree constraints, we end up with the following rule:

RULE 2: Whilst transforming Kconfig files into feature models, transform a tristate symbol A as follows:

1. First, create a new abstract feature called `A` and add this to the root feature of the feature model.
2. Now add two features as children to `A`: `A_m` and `A_t`. Set the children of `A` as mandatory alternatives (alt group), so that selecting `A` implies `A_m` xor `A_t`.

RULE 2.1: If A depends on another tristate symbol B:

1. Transform B according to rule 2. If A needs to be transformed during the transformation of B, abort transformation; the Kconfig file contains a dependency loop.
2. Add a cross-tree constraint: `A_m` implies `B`.
3. Add a cross-tree constraint: `A_t` implies `B_t`.

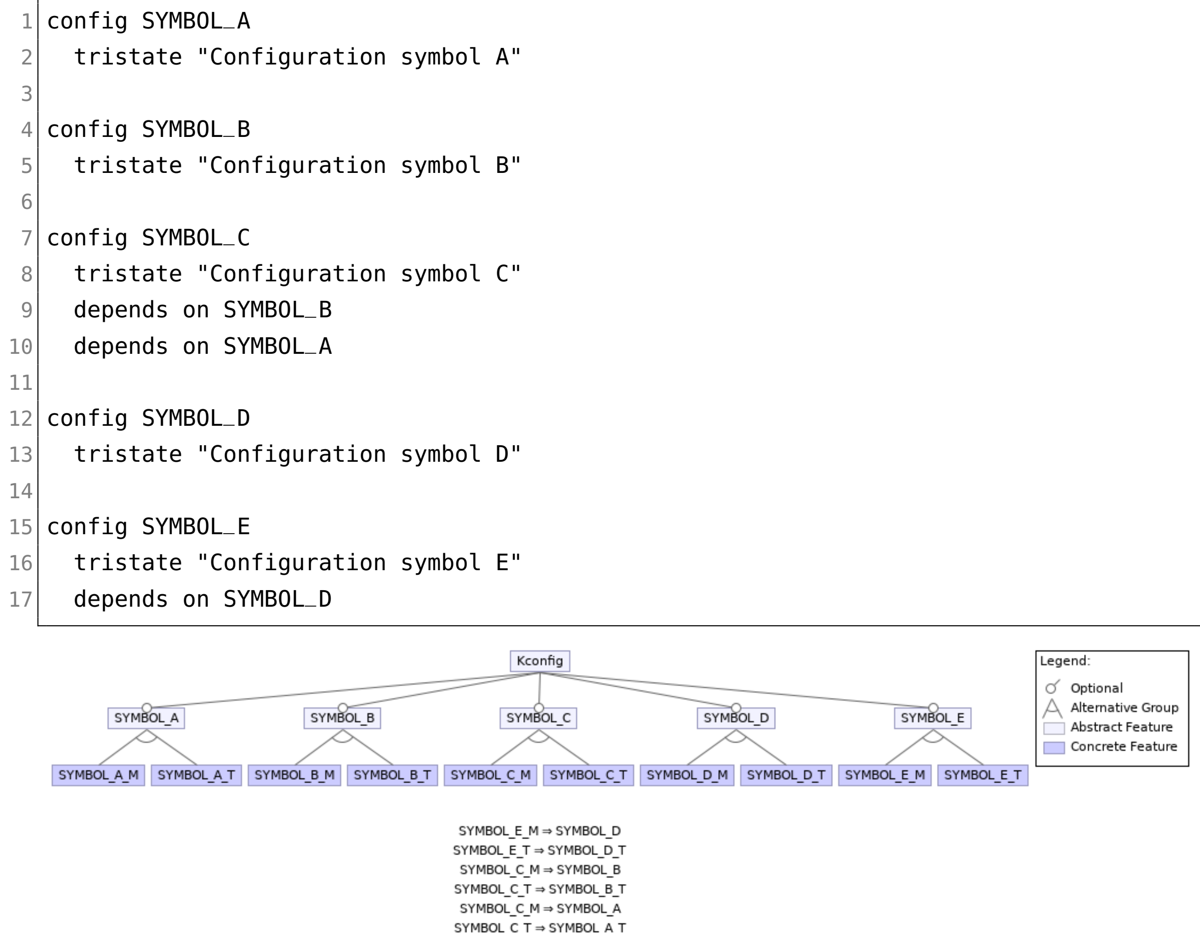
RULE 2.2: If A depends on multiple tristate symbols $B_1, \ldots, B_i$:

1. Process every pair $(A, B_l), l \in \{1, \ldots, i\}$ according to rule 2.1, considering every dependency of its own regard.

An example transformation using this rule is given in figure 7.

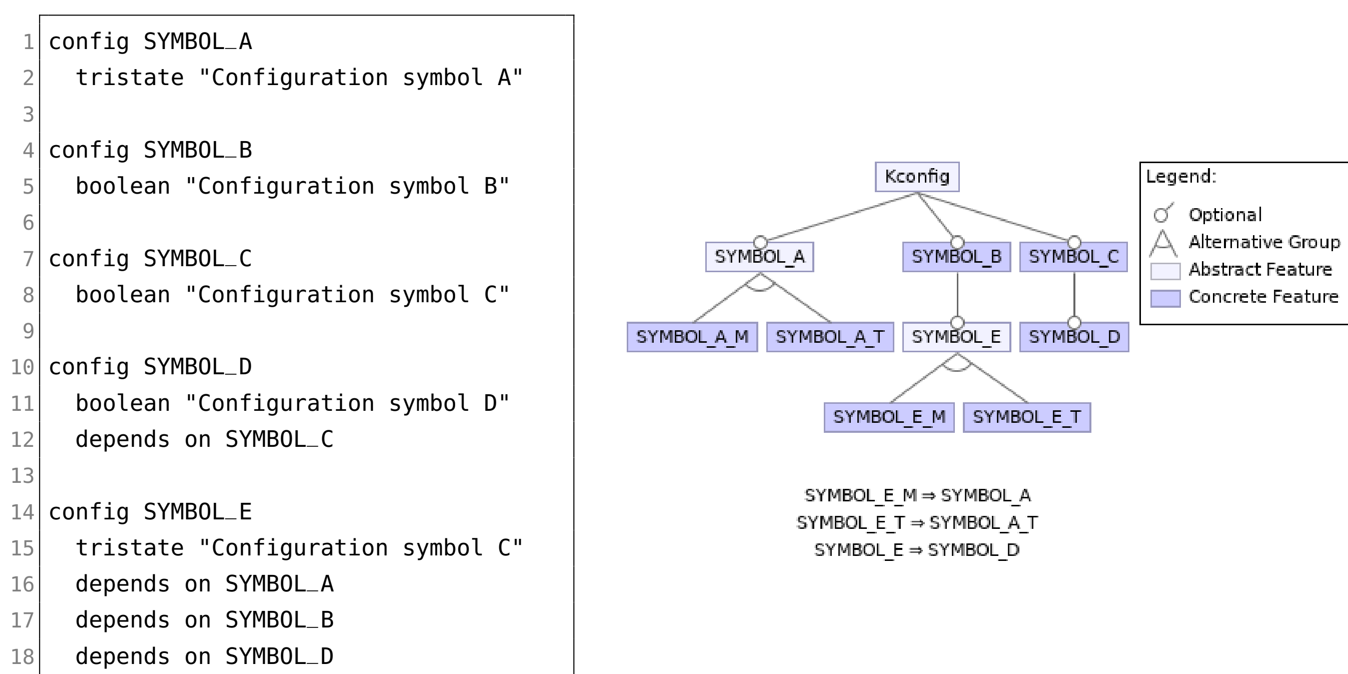
4.2.3. Mixed-type dependencies

Boolean configuration symbols can depend on tristate configuration symbols and vice versa. For this case, we introduce a further rule:



```
1  config SYMBOL_A
2    tristate "Configuration symbol A"
3
4  config SYMBOL_B
5    tristate "Configuration symbol B"
6
7  config SYMBOL_C
8    tristate "Configuration symbol C"
9    depends on SYMBOL_B
10   depends on SYMBOL_A
11
12 config SYMBOL_D
13   tristate "Configuration symbol D"
14
15 config SYMBOL_E
16   tristate "Configuration symbol E"
17   depends on SYMBOL_D
```


Figure 7.: Example transformation of the given Kconfig file. All configuration symbols were transformed using rule 2. The multiple dependency of `SYMBOL_C` and the single dependency of `SYMBOL_E` were handled by rule 2.2 and 2.1, respectively.



```
config SYMBOL_A
  tristate "Configuration symbol A"

config SYMBOL_B
  boolean "Configuration symbol B"

config SYMBOL_C
  boolean "Configuration symbol C"

config SYMBOL_D
  boolean "Configuration symbol D"
  depends on SYMBOL_C

config SYMBOL_E
  tristate "Configuration symbol C"
  depends on SYMBOL_A
  depends on SYMBOL_B
  depends on SYMBOL_D
```


Figure 8.: Example transformation of the given Kconfig file. `SYMBOL_E` was transformed using rule 3. `SYMBOL_E` is the child of `SYMBOL_B` and not `SYMBOL_D` as `SYMBOL_B` has less depth; this is the case as we explicitly mentioned that we should use the heuristic from rule 1.2 whilst processing boolean dependees of tristate dependers.

Rule 1.3: If a boolean symbol A depends on a tristate symbol B:

1. Add a cross-tree constraint: `A` implies `B`.

Rule 2.3: If a tristate symbol A depends on a boolean symbol B:

1. Make feature `A` child of feature `B`.

Rule 3: If a configuration symbol A has a mixed-type multiple dependency:

1. If A is a tristate symbol, for each other tristate symbol B_j which A depends on, process every pair according to rule 2.1. If A also depends on boolean configuration symbols, see rule 2.3 and use the heuristic from rule 1.2.
2. If A is a boolean symbol, see rule 1.2 and 1.3.

An example transformation using this rule is given in figure 8.

In case of boolean symbols depending on tristate symbols, it is sufficient for the dependee to assume the value `m` for the depender to assume the value `t`. This might seem contradictory to the truth table for dependencies between tristate symbols (as `m` and `t` are understood as distinct values) but experimenting with `menuconfig` shows that boolean dependers do not differentiate between `m` and `t`.

The dependency between a boolean depender *A* and a tristate dependee *B* cannot be represented as a child-parent feature relation because alt groups cannot be defined selectively. All children of `A` (`A_m` and `A_t`) are in an alt group and if `B` was then made a child feature of `B`, the transformation would have been semantically incorrect, as `B` would have been in an alt group with `A_m` and `A_t`. Ergo we use a cross-tree constraint to transform such a dependency.

4.2.4. Reverse dependencies

The `select` option in configuration symbols can be used to set a "lower bound" on another symbol, i.e. the dependee.

If *A* selects *B*, the following value matrix is to be considered (assuming *A* and *B* are boolean configuration symbols):

Value of symbol A	Value of symbol B	Dependency satisified?
f	f	yes
f	t	yes
t	f	no
t	t	yes

This value matrix is identical to the value matrix of "*A* depends on *B*". We can hence process reverse dependencies as we process regular dependencies.

RULE 4: If a configuration symbol *A* selects another configuration symbol *B*:

1. Process this relation as "*A* depends on *B*" according to the respective rules.

Process multiple selections in similar manner.

It might not be evident why the `select` keyword exists considering that is (logically) equivalent to an ordinary dependency. The official Kconfig documentation warns: "`select` should be used with care. `select` will force a symbol to a value without visiting the dependencies. By abusing `select` you are able to select a symbol FOO even if FOO depends on BAR that is not set." Kconfig handles `select` a bit differently than `depends`; there is no transitive dependency resolution. This should however not be an issue if `select` is only used with configuration symbols that have no dependencies; in that case `select` is no different than `depends`, hence our rule should hold.

4.2.5. Choice blocks

Choice blocks consist of multiple configuration symbols that are mutually exclusive: If the choice block is selected, only one of the configuration symbols might be set to `t`. In case of tristate choice blocks, if the choice block is set to `m`, an arbitrary amount of the contained configuration symbols might be set to `m`, however setting the choice block to `t` forces one of the contained symbols to be set to `t` and all others to be set to `f`, hence the `m` option is disabled when the choice block is set to `t`.

A reverse transformation of alt feature groups into boolean choice blocks has already been proposed by [34]. We build upon this transformation with certain considerations:

- Although choice blocks can be named, they may not occur as dependees [31]. At this instance they are similar to menu blocks.
- Choices are, by default, mandatory. A mandatory choice block must have a value set if all of its dependencies are satisfied.
- The Kconfig language specification is unclear about mixed choice blocks (e.g. tristate choice blocks containing boolean configuration symbols) [31]. We will assume that boolean and tristate choice blocks only contain configuration symbols of their respective types.

Modelling the mandatory nature of choice blocks may first seem trivial due to the presence of mandatory features in feature models, but this is unfortunately only the case if a choice block depends on a single boolean symbol: Only child-parent relationships can be defined as mandatory. Because we have already established that only one of the dependencies may be represented structurally in case of multiple dependencies (see rule 3), we need to work with cross-tree constraints to ensure that our transformation semantically preserves mandatory choice blocks. Without any consideration of syntax, the following logical formula must be true for a mandatory choice block C that depends on multiple other configuration symbols A_j with $j \in \{1, \ldots, i\}$:

$$A_1 \land \ldots \land A_i \implies C \tag{III}$$

(If all dependencies of C are selected/true, C must be selected/true)

Because $C \implies A_j, j \in \{1, \ldots, i\}$ is already implied through the dependency between C and A_j, (III) but as a bi-implication should also hold true for a mandatory choice block C. The bi-implication might however be redundant if $C \implies A_j$ already occurs for any $j \in \{1, \ldots, i\}$ as a cross-tree constraint. Additionally, it is important to mention that the formula above is not equivalent to multiple individual implications for each dependency ($A_1 \implies C$, $A_2 \implies C$ and so forth): A mandatory choice C should only be selected when *all* of its dependencies are satisfied. If C is a boolean choice block, rule 1.3 applies, so that if any of the A_j in (III) are tristate configuration symbols, the expression A_j for the respective configuration symbol should be replaced by $M(A_j) \lor T(A_j)$, i.e. it is sufficient for the dependee to be set to any value but `f`. This should however not be an issue whilst transforming, as in rule 2 the abstract parent feature `Aj` of `Aj_m` and `Aj_t` exactly corresponds to this expression.

With these considerations, we propose the following rules for the transformation of tristate and boolean choice blocks:

Rule 5: Whilst transforming a Kconfig file into a feature model, process a boolean choice block C as follows:

1. Create an abstract feature `C`.

2. If the choice block depends on another configuration symbol B, first process that dependency as if C is a boolean configuration symbol. If C doesn't depend on anything, make `C` child of the root feature of the feature model.

3. If C is not optional:
   - If C depends on multiple configuration symbols A_j ($j \in \{1, \ldots, i\}$): Add cross-tree constraint: `A1 AND ... AND Ai IMPLIES C`.
   - If C depends on a single configuration symbol A: If A is a tristate symbol, add cross-tree constraint `A IMPLIES C`. If A is a boolean symbol, make `C` a mandatory child feature of A.
   - If C does not have any dependencies, make `C` a mandatory child of the root feature.

4. Process all configuration symbols A_j contained in the choice block C, but ignore their dependencies.

5. Make all features corresponding to the contained configuration symbols of C children of feature `C`.

6. Make the children of `C` mandatory alternatives (alt group).

7. Now process the dependencies of all A_j contained in C. Ensure that features corresponding to A_j remain children of `C`, i.e., do not represent any of these dependencies structurally.

An example transformation using this rule is given in figure 9.

In case of tristate choice blocks, we effectively build upon rule 2: We want tristate choice blocks to have the same structure tristate configuration symbols have: Three features for the three states. A tristate choice block assumes the value `m` when one or more tristate configuration symbols contained in the choice block are set to `m`: Here "one or more" calls for an or group. When the choice block is set to `t`, *exactly one* of the enclosed configuration symbols must be set to `t`: This calls for an alt group.

We can describe this constraint in second-order logic (using the relations T and M we have used for rule 2). For a choice block C with enclosed tristate symbols $A_1, \ldots, A_j$, the following should hold true:

$$T(C) \implies ((T(A_1) \oplus \ldots \oplus T(A_j)) \land \neg(M(A_1) \lor \ldots \lor M(A_j))) \quad \text{(IV)}$$

$$M(C) \implies ((M(A_1) \lor \ldots \lor M(A_j)) \land \neg(T(A_1) \lor \ldots \lor T(A_j))) \quad \text{(V)}$$

We ensure that these expressions hold true by making the features corresponding to $M(A_i)$ for $i \in 1, \ldots, j$ children of `C_m`. These features are in an or group. We do the same for the features corresponding to $T(A_i)$ for $i \in 1, \ldots, j$; these features become children of `C_t` and are in an alt group. The second term in (IV) and (V) are already modelled through

```
config SYMBOL_A
  boolean "Configuration symbol A"

config SYMBOL_B
  tristate "Configuration symbol B"

choice CHOICE_C
  boolean "Choice block C"
  depends on SYMBOL_A
  depends on SYMBOL_B

  config SYMBOL_D
    boolean "Configuration symbol D"
    depends on SYMBOL_X

  config SYMBOL_E
    boolean "Configuration symbol E"
    depends on SYMBOL_Y

endchoice

config SYMBOL_Y
  boolean "Configuration symbol Y"

config SYMBOL_X
  tristate "Configuration symbol X"
```

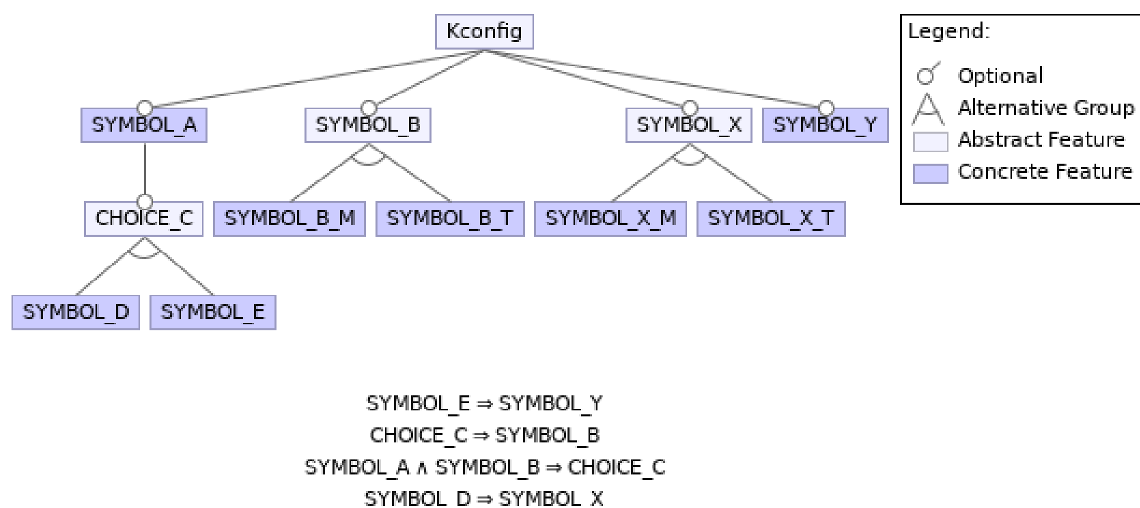


Figure 9.: Example transformation of the given Kconfig file. CHOICE_C was transformed using rule 5. As defined in the rule, the dependency between SYMBOL_E and SYMBOL_Y is not represented structurally, so that SYMBOL_D remained a child feature of CHOICE_C. Because CHOICE_C is not an optional choice, a cross-tree constraint with a conjunction over all dependees of CHOICE_C was added to the feature model.

the mutually exclusive nature of `C_t` and `C_m`. Breaking up the enclosed tristate symbols means that we omit the abstract parent feature `Ai` which we normally create whilst using rule 2. This is however not an issue: `Ai` can be replaced by `Ai_m OR Ai_t` in cross-tree constraints so that all rules also work for tristate symbols enclosed in choice blocks.

Ergo we end up with the following transformation steps:

RULE 6: Tristate choice blocks are to be processed differently; given the tristate choice block C:

1. Process C as if it was a tristate symbol: Create the features `C`, `C_t`, `C_m`. Resolve C's dependencies according to rule 2.

   ⚠ Due to a Kconfig implementation bug, tristate choice blocks are always optional (see figure 11).

2. For each tristate configuration symbol A_j within C, create a feature `Aj_m` and `Aj_t`. Make these features children of `C_m` and `C_t`, respectively.

3. Whilst resolving the dependencies of the configuration symbols within the choice block, follow rule 2, but avoid structural changes; model all dependencies as cross-tree constraints. Because for a given A_j in C there is no abstract parent feature `Aj`, substitute `Aj` with `Aj_m OR Aj_t`:

Type of dependency	Cross-tree constraint(s) to add
A_j depends on a boolean symbol B	`Aj_m OR Aj_t IMPLIES B`
A_j depends on a tristate symbol B	`Aj_m IMPLIES B` and `Aj_t IMPLIES B_t`

4. Make the children of `C_t` an alt group.

5. Make the children of `C_m` an or group.

6. If configuration symbols contained within C appear as dependees of other configuration symbols, process these dependencies as defined in their respective rules, but work around the missing abstract parent feature of tristate symbols contained within a choice block:

Type of dependency	Cross-tree constraint(s) to add
Boolean symbol B depends on A_j	`B IMPLIES Aj_m OR Aj_t`
Tristate symbol B depends on A_j	`B IMPLIES Aj_m OR Aj_t` and `B_t IMPLIES Aj_t`

An example transformation using this rule is given in figure 10.

4.2.6. Menus

A hierarchy between configuration symbols is already implied through the dependencies of these symbols, but the Kconfig language also allows the definition of menus, which group up the enclosed configuration symbols in a submenu. A menu may occur as a depender; in that case, all the configuration symbols contained within the menu transitively inherit

```
 1  config SYMBOL_A
 2    boolean "Configuration symbol A"
 3
 4  config SYMBOL_B
 5    tristate "Configuration symbol B"
 6
 7  choice CHOICE_C
 8    tristate "Choice block C"
 9    depends on SYMBOL_A
10
11    config SYMBOL_D
12      tristate "Configuration symbol D"
13      depends on SYMBOL_B
14
15    config SYMBOL_E
16      tristate "Configuration symbol E"
17
18  endchoice
19
20  config SYMBOL_F
21    tristate "Configuration symbol F"
22    depends on SYMBOL_E
23
24  config SYMBOL_G
25    boolean "Configuration symbol G"
26    depends on SYMBOL_D
```

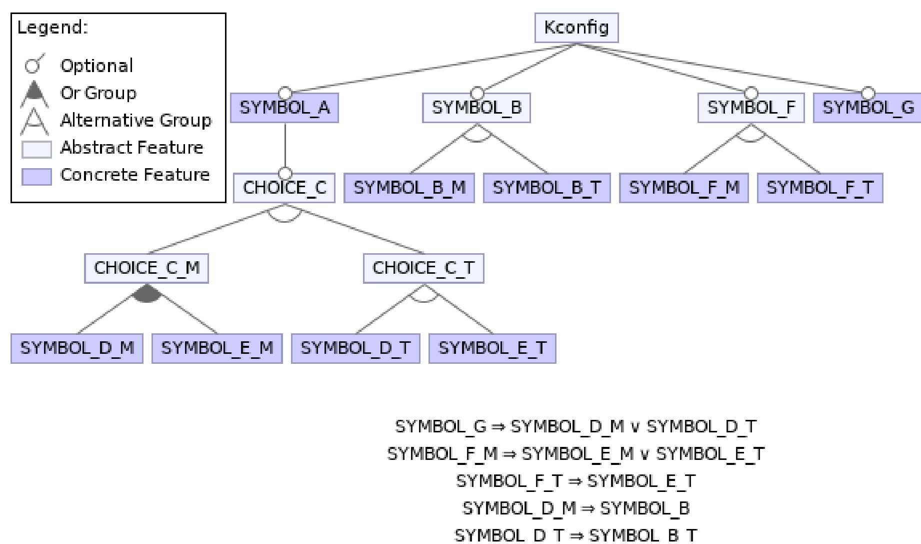


Figure 10.: Example transformation of the given Kconfig file. `CHOICE_C` was transformed using rule 6. `CHOICE_C` itself was processed like a regular tristate configuration symbol, the tristate symbols within `CHOICE_C` were broken apart; references to the missing abstract parent feature have been replaced by an OR expression of the two state features of the respective tristate symbol.

```
config MODULES
  bool
  modules
  default y

config SYMBOL_D
  bool "Configuration symbol D"

choice CHOICE_C
  tristate "Choice block C"
  depends on SYMBOL_D

  config SYMBOL_A
    tristate "Configuration symbol A"

  config SYMBOL_B
    tristate "Configuration symbol B"

endchoice
```

```
[*] Configuration symbol D
<M>    Choice block C
< >      Configuration symbol A
< >      Configuration symbol B
```

```
#
# Automatically generated file; DO NOT EDIT.
# Main menu
#
CONFIG_MODULES=y
CONFIG_SYMBOL_D=y
# CONFIG_SYMBOL_A is not set
# CONFIG_SYMBOL_B is not set
```

Figure 11.: Despite `CHOICE_C` being mandatory, the following `menuconfig` state could be saved; in the resulting `.config` file (bottom listing) is neither `SYMBOL_A` nor `SYMBOL_B` set.

these dependencies. Menus may contain further menus, but they may not occur within choice blocks.

There are also so-called `menuconfig` configuration symbols, which are (boolean) configuration symbols themselves but also imply a menu structure: All configuration symbols that depend on a `menuconfig` configuration symbol are placed in a submenu "hidden" under the respective `menuconfig` symbol (see image below).

We process menu blocks like a group of symbols that all depend on the same boolean symbol, so to say the boolean symbol that corresponds to the menu header. As menu blocks explicitly define a hierarchy (compared to dependencies which do the same but implicitly), we ensure that menus are structurally recognizable in the resulting feature model. For this we ignore the heuristic introduced in rule 1.

This "menu header" feature must be abstract, as it does not correspond to an actual configuration symbol contained in the source Kconfig file. In case of `menuconfig`, the menu header itself is a valid configuration symbol, so that the "header" feature should not be abstract.

Ultimately, we propose the following two rules for transforming menu blocks and `menuconfig` configuration symbols:

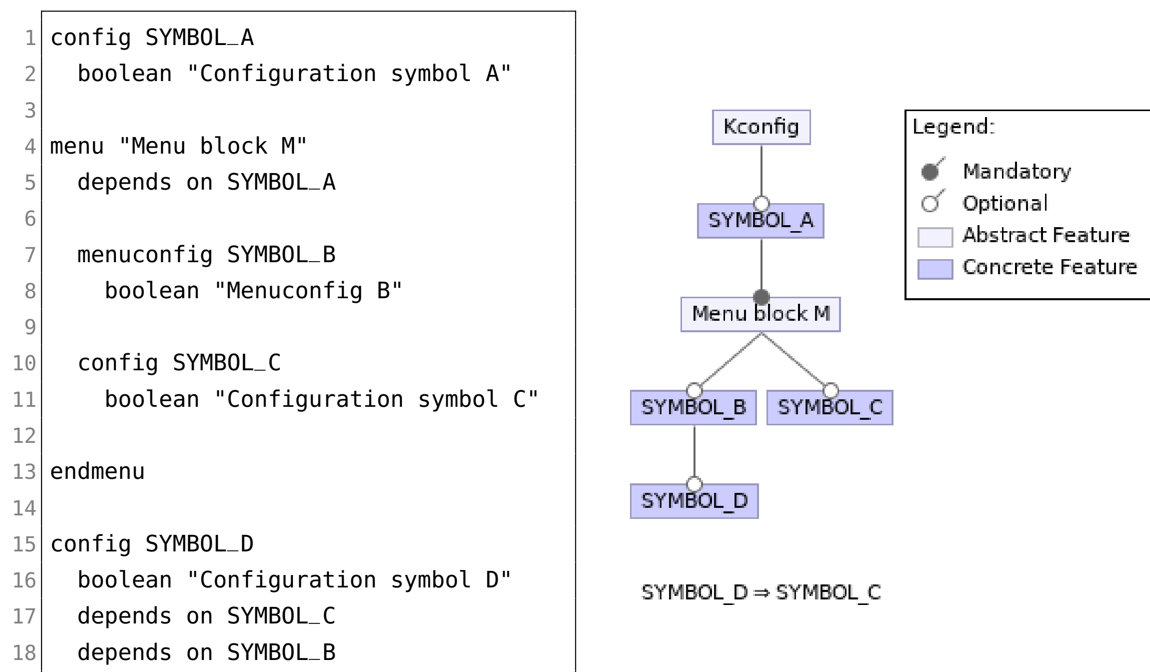
RULE 7: Transform a menu block M as follows:

1. Process the menu block itself like a boolean configuration symbol, see rule 1.
2. Make `M` an abstract and mandatory feature.
3. For all configuration symbols A_j contained within M, process the symbols according to their respective rules, but additionally consider the dependency on M and ensure that the features corresponding to A_j remain *immediate* children of `M` (i.e. ignore the heuristic used for transforming boolean symbols).

RULE 8: For a `menuconfig` configuration symbol M, do the following:

1. Process M like a boolean configuration symbol, see rule 1.
2. For all configuration symbols depend on M, ensure that the features corresponding to these configuration symbols remain *immediate* children of `M` (i.e. ignore the heuristic used for transforming boolean symbols).

An example transformation using the rules 7 and 8 is given in figure 12.



```
1  config SYMBOL_A
2    boolean "Configuration symbol A"
3
4  menu "Menu block M"
5    depends on SYMBOL_A
6
7    menuconfig SYMBOL_B
8      boolean "Menuconfig B"
9
10   config SYMBOL_C
11     boolean "Configuration symbol C"
12
13 endmenu
14
15 config SYMBOL_D
16   boolean "Configuration symbol D"
17   depends on SYMBOL_C
18   depends on SYMBOL_B
```


Figure 12.: Example transformation of the given Kconfig file. Menu block *M* was transformed using rule 7, `SYMBOL_B` was transformed using rule 8. In accordance to rule 8, `SYMBOL_D` is a child of `SYMBOL_B`, although the dependency to `SYMBOL_C` was defined first and both possible parents had equal depth; `SYMBOL_B` was however preferred as a `menuconfig` symbol.

5. Implementation

In this chapter, we aim to give an overview of the implementation of the transformation rules we have defined in the previous chapter in scope of a tool to transform Kconfig files into feature models.

5.1. Implementation details

We have implemented the transformations in Java using the Eclipse Modelling Framework (EMF), ANTLR3 and FeatureIDE. We have named the resulting tool "Kfeature", inspired by the K-prefixed names of utility scripts in the Linux kernel. We use ANTLR3 to parse Kconfig files, for this we use the ANTLR grammar developed by She [33]. Once an abstract syntax tree (AST) is created, we traverse this AST and bring the Kconfig file in an intermediate graph form: Configuration symbols are processed as nodes and the dependencies between configuration symbols are processed as edges. To deal with menu and choice blocks, we use so-called enclosing nodes, so that encapsulated nodes can be recognized during transformation.

Once the Kconfig file is brought to the intermediary graph form, we process this graph and apply the transformations introduced in the previous section. Instead of working with XML tags directly, we use the metamodel developed by Ateş [3], which corresponds to the XML schema used by FeatureIDE to represent feature models: We build the feature model corresponding to the transformed Kconfig file initially as a ECore model (with the automatically generated classes/objects corresponding to the FeatureIDE XML schema metamodel) and then serialize this model into an XML file to ultimately create a feature model that can be viewed with FeatureIDE.

We have chosen the FeatureIDE XML format as our transformation target as FeatureIDE is a gateway tool to feature-oriented programming/design; it allows the exporting of feature models into many different formats. This increases the impact of our contribution greatly. Our decision to use EMF and metamodels for the implementation of the transformation rules was mostly pragmatic: As there was already a tested metamodel for the FeatureIDE XML schema, implementation of an own metamodel or class hierarchy to model the structrure of FeatureIDE XML files would have endangered the internal validity of our implementation.

5.2. Challenges during implementation

The parser was a constant source of problems during implementation. There is an official Kconfig grammar that is also used by the Kconfig tool itself, but this grammar is to be used

with the parser generator Bison, which generates parser code in C/C++. Because Kfeature had to be developed in Java (as FeatureIDE only offers Java libraries), it was not possible to bind the parser created by Bison into Kfeature (Bison *should* also be able to generate Java code, but the official Kconfig grammar contains multiple custom functions in C, which Bison cannot translate). Creation of a correct and complete translation of the official Yacc/Bison grammar was unrealistic, as the author of this thesis had no prior experience with these parser generators and even if such a translation was to be done within the scope of this thesis, there would be no possibility to test the correctness of the resulting ANTLR grammar, as there is no other official grammar that is complete other than the Yacc/Bison grammar contained in the Linux source tree; building upon a self-translated and unverified parser grammar would have limit the scope of the implementation immensely and moreover reduce the validity of evaluation results. Hence the plan to implement an own parser grammar was scrapped early on during the development of Kfeature.

Due to the lack of Java software projects that work with Kconfig files, it was not possible to simply adapt the parsing infrastructure of an already existing software tool. Because the absence of a parser was blocking the further development of Kfeature, we have made the decision to use She's ANTLR3 grammar, despite it having its shortcomings.

Because FeatureIDE feature models are FODA-like, implementation of the transformation rules was rather straight-forward: The main hinderance whilst implementing the transformation rules was the lack of EMF documentation, which meant we had to experiment with the EMF API to reach the wished results.

Kfeature has a few limitations:

- `source` blocks are not supported due missing parser rules.
- When a mandatory boolean choice block only has one dependency, the feature corresponding to the choice block is not made a mandatory feature, it is instead processed as if it has multiple dependencies (see rule 5, step 3).
- The official Kconfig tool does not differentiate between spaces and tabs for indentation, but Kfeature only supports tabs due to shortcomings of the used ANTLR3 grammar.

6. Evaluation

In this chapter, we discuss the evaluation of transformation rules we have introduced in chapter 4. In the first section, we introduce the methods of evaluation we have used. In the second section, we discuss the metric of human-readability. In the third section, we argue for the adequacy of the evaluation methods we have introduced in the first section. Finally, we report the results of our evaluation and mention few risks to validity that we have detected during the conception of this thesis.

6.1. Methods of evaluation

It is important to define when we consider a transformation "correct". We made clear in chapter 4 that we want our transformation rules to preserve the semantics of the underlying Kconfig file, i.e. the constraints defined in a Kconfig file should also be present in the corresponding feature model. To evaluate how well our transformation rules accomplish this, we can check for logical equivalency of a Kconfig file and its transformed feature model counterpart:

> A Kconfig file and a feature model are equivalent when every invalid configuration of the feature model corresponds to an invalid configuration of the Kconfig file. Additionally every valid configuration of the feature module must correspond to a valid configuration of Kconfig file.

This process can be automated using the FeatureIDE API: FeatureIDE can generate all solutions (i.e. configurations that fulfil the constraints of a feature model) for a given feature model [23]. Each solution must then be fed to a tool that checks if the given configuration is valid for the underlying Kconfig file. The remaining configurations of the feature model are hence non-solutions, these should correspond to invalid Kconfig configurations. The set of all possible configurations of a feature model corresponds to the power set of features that are found in this feature model (it is important these configurations are *possible*, but not necessarily valid).

If automated validation is not possible, one can manually check for equivalency by attempting to reconstruct a given solution/non-solution for a feature model in the Kconfig menu interface (`menuconfig`). Every configuration that can be reconstructed within the Kconfig menu interface is valid (as every such configuration can be saved and used whilst building the underlying software project). In this case, a Kconfig file and a feature model should be equivalent if every solution of the feature model can be reconstructed in `menuconfig`. Additionally it should be impossible to reconstruct a non-solution of the feature model in `menuconfig`.

6.2. Human readability of the created feature models

Another goal that we have mentioned in the concept chapter of this thesis was human readability. We want the explicit menu structure of the Kconfig file we are transforming to be visible in the resulting feature model, i.e. menu and choice blocks should be recognizable. No evaluation of this property is necessary, as it is ensured directly by definition of the respective rules (see rules 1, 5, 6, 7 and 8 in chapter 4). We can check if this is the case in the feature models created by Kfeature to see if the aforementioned rules were implemented correctly in Kfeature. For this, we introduce the following metric:

A transformed feature model fulfils the *human-readability metric* if the following requirements are met:

1. All configuration symbols (or moreover the features they are transformed into) that are contained within a menu block are immediate children of the abstract feature that corresponds to the aforementioned menu block in the resulting feature model.
2. If a configuration symbol depends on a `menuconfig` symbol, the feature that corresponds to the configuration symbol should be a child feature of the feature that corresponds to the `menuconfig` symbol. If multiple `menuconfig` dependees occur, this requirement applies for the `menuconfig` symbol with the least depth.
3. The tristate structure with the abstract parent feature (as defined in rule 2) should remain intact, with the exception of tristate symbols in choice blocks (see rule 7).
4. In case of multiple dependencies, ensure that the heuristic defined in rule 1.2 was applied correctly.

6.3. Converting feature model configurations to Kconfig configurations

In multiple transformation rules, we have marked certain features as abstract. These were features that did not have a corresponding configuration symbol in the underlying Kconfig file, ex. choice blocks themselves cannot be set to `t` or `f`, because they are not configuration symbols in the conventional sense. Configurations created by `menuconfig` reinforce this notion, see figure 13. Even when a choice block is selected in `menuconfig`, the resulting `.config` file does not contain an assignment for the variable that corresponds to the choice block, as no such variable exists in the first place. “Selection” of such menu entries is fully cosmetic.

Because abstract features do not correspond to specific configuration symbols, they must be ignored whilst converting feature model configurations into respective Kconfig

configurations. Additionally they may cause incorrect feature model configurations to correspond valid Kconfig configurations, see figure 14.

In case of tristate configuration symbols, the configuration symbol *C* should be set to `m` if the corresponding feature `C_m` is selected, and respectively set to `t` if `C_t` is selected. A configuration where both of these features are selected is necessarily incorrect (see rule 2) but moreover it cannot be converted into a respective Kconfig configuration, as a tristate configuration symbol cannot be set to both `t` and `m` within the same configuration file. In this case we assume that the validator would reject the configuration due to invalid syntax and say that the validator has correctly recognized the invalidity of the underlying feature model configuration.

6.4. Results

Five Kconfig files were transformed and validated automatically according to the method described in the first section of this chapter. We have used the `klocalizer` tool from the kmax tool suite [14] to check if a given configuration is valid or not. `klocalizer` returns non-zero for a configuration that is invalid (a configuration that does not satisfy the constraints defined in the underlying Kconfig file), and zero for a configuration that is valid. Results of the automatic verification can be seen in figure 15. "Used rules" refers to rules that were utilised by Kfeature while transforming the given Kconfig file. "Validity rate" is the ratio of correctly transformed configurations to the total number of configurations (when abstract features are omitted). "Human-readability metric" is checked if the conditions given in 6.2 are fulfilled by the resulting feature model.

The klocalizer tool however does not support tristate configuration symbols (or at least not in full scope, see issue #230 in [14]). This made the automated validation of Kconfig files with tristate configuration symbols impossible. In this case, we transformed four Kconfig files with tristate configuration symbols using Kfeature and sampled 60 configurations from each generated feature model and checked the equivalency to their `menuconfig` counterparts. Sampling was done using FeatureIDE. To ensure that both solutions and non-solutions are represented in the set of sampled configurations, we have ensured that at least half of the sampled configurations are solutions (for cases where there were less than 30 solutions, we have sampled more non-solutions to reach 60 configurations in total). Results of the manual vertification of the aforementioned four Kconfig files can be seen in figure 16. Sample validity rate refers to the percentage of sampled configurations that were respectively solutions and non-solutions for both the feature model and the underlying Kconfig file. Sampling details for the individual Kconfig files can be found in the appendix.

The respective Kconfig files and the corresponding feature models can be found in the appendix. Raw data containing the configurations that were sampled can be found on Zenodo [40].

```
config MODULES
  bool
  modules
  default y

config D
  bool "Configuration symbol D"

choice C
  bool "Choice block C"
  depends on D

  config A
    bool "Configuration symbol A"

  config B
    bool "Configuration symbol B"

endchoice
```

```
#
# Automatically generated file; DO NOT EDIT.
# Main menu
#
CONFIG_MODULES=y
CONFIG_D=y
CONFIG_A=y
# CONFIG_B is not set
```

Figure 13.: For the given Kconfig file above, the shown `menuconfig` state (image in the middle) is represented with the given configuration file. There is no `CONFIG_C` entry in the configuration file (bottom listing).

```
menu "Menu block M"

  config SYMBOL_A
    boolean "Configuration symbol A"

endmenu

config SYMBOL_B
  boolean "Configuration symbol B"
```

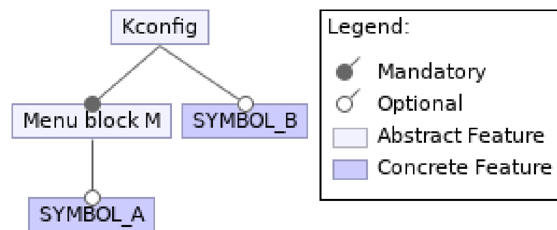


Figure 14.: The configuration `SYMBOL_A=y` is valid for the given Kconfig file, but the configuration [`SYMBOL_A`] is not a solution for the resulting feature model, due to the abstract feature `Menu block M`.

Name	Used rules	Number of solutions	Validity rate	Human-readability metric
Kconfig1	1	6	16/16	✓
Kconfig2	1, 5, 7	11	64/64	✓
Kconfig3	1, 7, 8	64	128/128	✓
Kconfig4	1, 4, 5, 7	9	64/64	✓
Kconfig5	1, 8	15	128/128	✓

Figure 15.: Results of the automatic verification for the chosen five Kconfig files.

Name	Used rules	Sample validity rate	Human-readability metric
Kconfig6	1, 2, 3	100%	✓
Kconfig7	1, 3, 4, 5, 6	100%	✓
Kconfig8	3, 7, 8	100%	✓
Kconfig9	all rules	100%	✓

Figure 16.: Results of the manual verification of the chosen four Kconfig files.

6.5. Interpretation

With 100% validity rate for both manually and automatically verified Kconfig files, we can say that our transformations map most Kconfig files onto semantically equivalent feature models, given that the Kconfig file only contains constructs which we have transformation rules for. Additionally, the human-readability metric is satisfied by all the generated feature models, this is a strong argument for the correct implementation of the concerning transformation rules in Kfeature.

6.6. Threats to validity

In this section, we will shortly discuss the possible threats to internal and external validity.

6.6.1. External validity

In case of our thesis, external validity refers to the question: Is the correct transformation of these nine Kconfig files sufficient to argument for the correct transformation of all Kconfig files that exclusively consist of Kconfig structures that we have transformation rules for, i.e. to what degree can we scale up the results of our evaluation?

The Kconfig files we have used for evaluation were tailored specifically to contain many different combinations of constructs (ex. choice blocks in menu blocks, nested menu blocks, multiple dependencies), however the number of configuration symbols had to be limited due to the fact that automated verification of feature models can take very long (due to the sheer number of possible configurations to process). Manual verification of such feature models is possible, however the sample size of 60 becomes meaninglessly small for feature models with 20+ features, and larger sample sizes make manual verification error-prone. Hence it is possible that the 100% validity rate falsely implies the *total* correctness of the proposed transformation rules. We would nevertheless argue that 100% validity for a selection of smaller Kconfig files with many combinations of constructs implies some validity for the transformation of larger Kconfig files, as larger Kconfig files tend to consist of separate parts that can be their own Kconfig files (this is why the Linux kernel utilises the `source` keyword very often). Beyond this, the size of a Kconfig file does not always mean a challenge for transformation, ex. the transformation of a Kconfig file with 1000 boolean configuration symbols without any dependencies is trivial, wherein the transformation of a Kconfig file with 5 configuration symbols in choice blocks and mixed-type dependencies is not.

6.6.2. Internal validity

Internal validity in this instance refers to the correctness of the evaluation methods we have chosen to utilise for the verification of the correctness of our transformation rules.

Manual verification is always error-prone, it is possible that we have deemed a configuration invalid assuming that it cannot be built with `menuconfig`, but it is possible that we have overseen a certain configuration symbol or somehow failed to recognize that

the configuration can be actually built. We have attempted to minimize the possibility of such an occurrence by prioritising rule coverage over the net number of configuration symbols: The Kconfig files that we analysed are small but complex, so that many different transformation rules must be utilised for a full transformation.

The tools that we have used for verification, FeatureIDE and `kmax`, are mature software projects that have been already been utilised in scope of other research papers. It is nevertheless possible that e.g. `klocalizer` has returned zero for an invalid configuration due to an implementation bug.

7. Conclusion

The goal of this thesis was to develop and present a method for the semantics-preserving transformation of Kconfig files in feature models.

Before making any considerations about the transformation rules themselves, we have first set a scope for our transformations; we have chosen *what* we want to transform. We have set transformation goals: total semantic equivalency and preservation of the explicit menu structure.

The results of the evaluation of the transformation rules we have developed offer a strong argument for the accomplishment of our first goal. Additionally, we have documented the considerations we had whilst developing the transformation rules, so that any future work that aims to develop a method for transforming Kconfig files to another target format can use our approach to Kconfig as a starting point for their own work.

The Kconfig framework has rather rudimentary documentation and there is no formal definition of the Kconfig language. Beyond introducing a transformation, this thesis can additionally be seen as an exploration of the Kconfig framework, with all the undefined behaviour and/or peculiarities we have discovered whilst developing the transformation rules. These occurrences are documented in this thesis.

7.1. Benefits

We have stated in the introduction chapter that we aim to close the gap between Kconfig files and feature model analysis tools through the implementation of a transformation between Kconfig files and feature models. We have accomplished this. Feature models that are created by Kfeature can be loaded into FeatureIDE and FeatureIDE can even deliver ad-hoc insights on Kconfig files. Figure 17 shows an example of a redundant dependency being detected by FeatureIDE.

This is only one example how the generated feature models can be processed further. Our contribution lays the groundwork of studies and surveys that can be done on software projects that utilise the Kconfig framework with the use of feature model analysis tools.

7.2. Future Work

In extent of the evaluation chapter of this thesis, we have shown the logical equivalence of five Kconfig files to their generated feature models. We have shown logical equivalence over model exhaustion; effectively, we have checked if both the Kconfig file and the feature model deliver the same truth value (solution/not a solution) for every model, models being possible (but not necessarily valid) configurations of the generated feature model. Another

```
config SYMBOL_A
  bool "Configuration symbol A"

menu "Menu block M"
depends on SYMBOL_A

  config SYMBOL_B
    bool "Configuration symbol B"
    depends on SYMBOL_A

endmenu
```

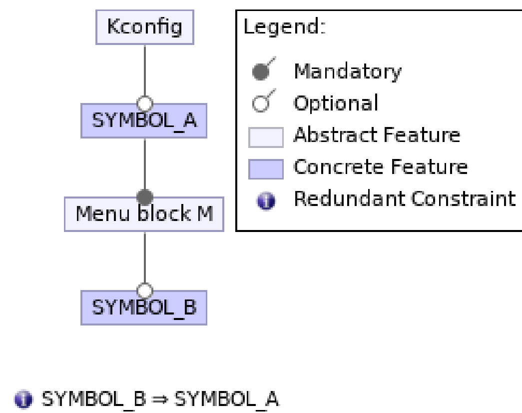


Figure 17.: FeatureIDE marks the cross-tree constraint SYMBOL_B IMPLIES SYMBOL_A as redundant. This was detected correctly: The respective dependency in the given Kconfig file is redundant.

and most probably more robust approach would have been transforming the Kconfig file into expressions in propositional logic (e.g. with `undertaker`) and doing the same with the feature model. Then the generated expressions could have been analysed for logical equivalency, through the use of a SAT solver. This might be the fundament of a future work aiming to further evaluate the transformation rules we have presented in this thesis.

Implementation of a reverse transformation (transformation of feature models into Kconfig files) would allow researchers and developers of projects that utilise Kconfig to apply the modifications done on the feature models on the corresponding Kconfig files. This would allow the automatic application of feature model optimisations on Kconfig files.

A comparative study of what insights feature model tools can deliver compared to the already existing Kconfig analysis tools such as `undertaker`, `fmdiff` and LVAT would allow us the measure the impact of our work in comparison to other work in this area of interest.

Bibliography

[1] *APACHE NUTTX*. original-date: 2019-12-14T23:27:55Z. Mar. 2023. URL: https://github.com/apache/nuttx (visited on 03/28/2023).

[2] Sven Apel and Christian Kästner. "An Overview of Feature-Oriented Software Development." en. In: *The Journal of Object Technology* 8.5 (2009), p. 49. ISSN: 1660-1769. DOI: 10.5381/jot.2009.8.5.c5. URL: http://www.jot.fm/contents/issue_2009_07/column5.html (visited on 12/01/2022).

[3] Atilla Ateş. "Konsistenzerhaltung von Feature-Modellen durch externe Sichten". de. In: (2022). Medium: PDF Publisher: Karlsruher Institut für Technologie (KIT). DOI: 10.5445/IR/1000143212. URL: https://publikationen.bibliothek.kit.edu/1000143212 (visited on 12/22/2022).

[4] Chris Bennett et al. "The Aesthetics of Graph Visualization". en. In: *Computational Aesthetics in Graphics* Visualization (2007). Artwork Size: 8 pages ISBN: 9783905673432 Publisher: The Eurographics Association, 8 pages. ISSN: 1816-0859. DOI: 10.2312/COMPAESTH/COMPAESTH07/057-064. URL: http://diglib.eg.org/handle/10.2312/COMPAESTH.COMPAESTH07.057-064 (visited on 03/29/2023).

[5] Thorsten Berger et al. "A Study of Variability Models and Languages in the Systems Software Domain". In: *IEEE Transactions on Software Engineering* 39.12 (Dec. 2013), pp. 1611–1640. ISSN: 0098-5589, 1939-3520. DOI: 10.1109/TSE.2013.34. URL: http://ieeexplore.ieee.org/document/6572787/ (visited on 11/29/2022).

[6] Thorsten Berger et al. "Variability modeling in the real: a perspective from the operating systems domain". en. In: *Proceedings of the IEEE/ACM international conference on Automated software engineering - ASE '10*. Antwerp, Belgium: ACM Press, 2010, p. 73. ISBN: 978-1-4503-0116-9. DOI: 10.1145/1858996.1859010. URL: http://portal.acm.org/citation.cfm?doid=1858996.1859010 (visited on 11/29/2022).

[7] Danilo Beuche, Holger Papajewski, and Wolfgang Schröder-Preikschat. "Variability management with feature models". en. In: *Science of Computer Programming* 53.3 (Dec. 2004), pp. 333–352. ISSN: 01676423. DOI: 10.1016/j.scico.2003.04.005. URL: https://linkinghub.elsevier.com/retrieve/pii/S0167642304000954 (visited on 03/28/2023).

[8] Jin Cao. *Exploring the Linux kernel: The secrets of Kconfig/kbuild*. en. Oct. 2018. URL: https://opensource.com/article/18/10/kbuild-and-kconfig (visited on 03/28/2023).

[9] Christian Dietrich et al. "A robust approach for variability extraction from the Linux build system". en. In: *Proceedings of the 16th International Software Product Line Conference on - SPLC '12 -volume 1*. Salvador, Brazil: ACM Press, 2012, p. 21. ISBN: 978-1-4503-1094-9. DOI: 10.1145/2362536.2362544. URL: http://dl.acm.org/citation.cfm?doid=2362536.2362544 (visited on 12/05/2022).

[10] Nicolas Dintzner, Arie van Deursen, and Martin Pinzger. "Analysing the Linux kernel feature model changes using FMDiff". en. In: *Software & Systems Modeling* 16.1 (Feb. 2017), pp. 55–76. ISSN: 1619-1366, 1619-1374. DOI: 10.1007/s10270-015-0472-2. URL: http://link.springer.com/10.1007/s10270-015-0472-2 (visited on 11/29/2022).

[11] Mohammed El Dammagh and Olga De Troyer. "Feature Modeling Tools: Evaluation and Lessons Learned". In: *Advances in Conceptual Modeling. Recent Developments and New Directions*. Ed. by Olga De Troyer et al. Vol. 6999. Series Title: Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 120–129. ISBN: 978-3-642-24573-2 978-3-642-24574-9. DOI: 10.1007/978-3-642-24574-9_17. URL: http://link.springer.com/10.1007/978-3-642-24574-9_17 (visited on 03/28/2023).

[12] David Fernandez-Amoros et al. "A Kconfig Translation to Logic with One-Way Validation System". en. In: *Proceedings of the 23rd International Systems and Software Product Line Conference - Volume A*. Paris France: ACM, Sept. 2019, pp. 303–308. ISBN: 978-1-4503-7138-4. DOI: 10.1145/3336294.3336313. URL: https://dl.acm.org/doi/10.1145/3336294.3336313 (visited on 11/29/2022).

[13] Matthias Galster et al. "Variability in software architecture: current practice and challenges". en. In: *ACM SIGSOFT Software Engineering Notes* 36.5 (Sept. 2011), pp. 30–32. ISSN: 0163-5948. DOI: 10.1145/2020976.2020978. URL: https://dl.acm.org/doi/10.1145/2020976.2020978 (visited on 03/28/2023).

[14] Paul Gazzillo. *The kmax tool suite*. original-date: 2017-06-30T16:25:48Z. Mar. 2023. URL: https://github.com/paulgazz/kmax (visited on 03/28/2023).

[15] Stefan Hengelein. "Analyzing the Internal Consistency of the Linux KConfig Model". en. MA thesis. University of Erlangen, Dept. of Computer Science, July 2015. URL: https://www4.cs.fau.de/Ausarbeitung/MA-I4-2015-04-Hengelein.pdf.

[16] *init-kconfig - easy way to embrace Linux's kconfig [LWN.net]*. URL: https://lwn.net/Articles/767691/ (visited on 03/28/2023).

[17] *ipconfig.c « ipv4 « net - kernel/git/torvalds/linux.git - Linux kernel source tree*. URL: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/net/ipv4/ipconfig.c (visited on 03/29/2023).

[18] Kyo C. Kang et al. *Feature-Oriented Domain Analysis (FODA) Feasibility Study:* tech. rep. Fort Belvoir, VA: Defense Technical Information Center, Nov. 1990. DOI: 10.21236/ADA235785. URL: http://www.dtic.mil/docs/citations/ADA235785 (visited on 12/02/2022).

[19] Christian Kästner and Sven Apel. “Feature-Oriented Software Development”. en. In: *Generative and Transformational Techniques in Software Engineering IV*. Ed. by Ralf Lämmel, João Saraiva, and Joost Visser. Vol. 7680. Series Title: Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 346–382. ISBN: 978-3-642-35991-0 978-3-642-35992-7. DOI: 10.1007/978-3-642-35992-7_10. URL: http://link.springer.com/10.1007/978-3-642-35992-7_10 (visited on 03/28/2023).

[20] *Kconfig « ipv4 « net - kernel/git/torvalds/linux.git - Linux kernel source tree*. URL: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/net/ipv4/Kconfig (visited on 03/29/2023).

[21] *Kconfig Language — The Linux Kernel documentation*. URL: https://www.kernel.org/doc/html/latest/kbuild/kconfig-language.html (visited on 03/28/2023).

[22] *Kconfig make config — The Linux Kernel documentation*. URL: https://www.kernel.org/doc/html/latest/kbuild/kconfig.html#menuconfig (visited on 03/28/2023).

[23] Sebastian Krieter et al. “FeatureIDE: Empowering Third-Party Developers”. en. In: *Proceedings of the 21st International Systems and Software Product Line Conference - Volume B*. Sevilla Spain: ACM, Sept. 2017, pp. 42–45. ISBN: 978-1-4503-5119-5. DOI: 10.1145/3109729.3109751. URL: https://dl.acm.org/doi/10.1145/3109729.3109751 (visited on 02/20/2023).

[24] Kun Chen et al. “An approach to constructing feature models based on requirements clustering”. In: *13th IEEE International Conference on Requirements Engineering (RE’05)*. Paris, France: IEEE, 2005, pp. 31–40. ISBN: 978-0-7695-2425-2. DOI: 10.1109/RE.2005.9. URL: http://ieeexplore.ieee.org/document/1531025/ (visited on 12/02/2022).

[25] Thomas Leich et al. “Tool support for feature-oriented software development: featureIDE: an Eclipse-based approach”. en. In: *Proceedings of the 2005 OOPSLA workshop on Eclipse technology eXchange - eclipse ’05*. San Diego, California: ACM Press, 2005, pp. 55–59. DOI: 10.1145/1117696.1117708. URL: http://portal.acm.org/citation.cfm?doid=1117696.1117708 (visited on 03/27/2023).

[26] Jörg Liebig et al. “An analysis of the variability in forty preprocessor-based software product lines”. en. In: *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*. Cape Town South Africa: ACM, May 2010, pp. 105–114. ISBN: 978-1-60558-719-6. DOI: 10.1145/1806799.1806819. URL: https://dl.acm.org/doi/10.1145/1806799.1806819 (visited on 03/28/2023).

[27] Rafael Lotufo et al. “Evolution of the Linux Kernel Variability Model”. In: *Software Product Lines: Going Beyond*. Ed. by David Hutchison et al. Vol. 6287. Series Title: Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 136–150. ISBN: 978-3-642-15578-9 978-3-642-15579-6. DOI: 10.1007/978-3-642-15579-6_10. URL: http://link.springer.com/10.1007/978-3-642-15579-6_10 (visited on 11/29/2022).

[28] Jeho Oh et al. “Finding broken Linux configuration specifications by statically analyzing the Kconfig language”. en. In: *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. Athens Greece: ACM, Aug. 2021, pp. 893–905. ISBN: 978-1-4503-8562-6. DOI: `10.1145/3468264.3468578`. URL: `https://dl.acm.org/doi/10.1145/3468264.3468578` (visited on 03/28/2023).

[29] Jeho Oh et al. “Uniform sampling from kconfig feature models”. In: *The University of Texas at Austin, Department of Computer Science, Tech. Rep. TR-19* 2 (2019).

[30] *parser.y « kconfig « scripts - kernel/git/torvalds/linux.git - Linux kernel source tree.* URL: `https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/scripts/kconfig/parser.y` (visited on 03/29/2023).

[31] Sascha El-Sharkawy, Adam Krafczyk, and Klaus Schmid. “Analysing the Kconfig semantics and its analysis tools”. en. In: *Proceedings of the 2015 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*. Pittsburgh PA USA: ACM, Oct. 2015, pp. 45–54. ISBN: 978-1-4503-3687-1. DOI: `10.1145/2814204.2814222`. URL: `https://dl.acm.org/doi/10.1145/2814204.2814222` (visited on 11/29/2022).

[32] S. She et al. “The Variability Model of The Linux Kernel”. In: 2010. URL: `https://www.semanticscholar.org/paper/The-Variability-Model-of-The-Linux-Kernel-She-Lotufo/944f35fccdcfc5d189375812d3e7e5ae3519e603` (visited on 03/29/2023).

[33] Steven She. *kconfig-g: ANTLR grammar for parsing Linux Kconfig files*. July 2009. URL: `https://code.google.com/archive/p/kconfig-g/` (visited on 03/30/2023).

[34] Julio Sincero and Wolfgang Schröder-Preikschat. “The Linux Kernel Configurator as a Feature Modeling Tool”. In: *Software Product Lines, 12th International Conference, SPLC 2008, Limerick, Ireland, September 8-12, 2008, Proceedings. Second Volume (Workshops)*. Ed. by Steffen Thiel and Klaus Pohl. Lero Int. Science Centre, University of Limerick, Ireland, 2008, pp. 257–260.

[35] Julio Sincero et al. “Efficient extraction and analysis of preprocessor-based variability”. en. In: *Proceedings of the ninth international conference on Generative programming and component engineering - GPCE ’10*. Eindhoven, The Netherlands: ACM Press, 2010, p. 33. ISBN: 978-1-4503-0154-1. DOI: `10.1145/1868294.1868300`. URL: `http://portal.acm.org/citation.cfm?doid=1868294.1868300` (visited on 12/05/2022).

[36] Julio Sincero et al. “Is The Linux Kernel a Software Product Line?” In: *Proceedings of the International Workshop on Open Source Software and Product Lines (SPLC-OSSPL 2007)*. Ed. by Frank van der Linden and Björn Lundell. Kyoto, Japan, 2007. URL: `http://fame-dbms.org/publications/SPLC-OSSPL2007-Sincero.pdf`.

[37] Reinhard Tartler et al. “Feature consistency in compile-time-configurable system software: facing the linux 10,000 feature problem”. en. In: *Proceedings of the sixth conference on Computer systems - EuroSys ’11*. Salzburg, Austria: ACM Press, 2011, p. 47. ISBN: 978-1-4503-0634-8. DOI: `10.1145/1966445.1966451`. URL: `http://portal.acm.org/citation.cfm?doid=1966445.1966451` (visited on 12/01/2022).

[38] Thomas Thum et al. “Abstract Features in Feature Modeling”. en. In: *2011 15th International Software Product Line Conference*. Munich, Germany: IEEE, Aug. 2011, pp. 191–200. ISBN: 978-1-4577-1029-2. DOI: 10.1109/SPLC.2011.53. URL: http://ieeexplore.ieee.org/document/6030061/ (visited on 03/20/2023).

[39] Thomas Thüm et al. “FeatureIDE: An extensible framework for feature-oriented software development”. en. In: *Science of Computer Programming* 79 (Jan. 2014), pp. 70–85. ISSN: 01676423. DOI: 10.1016/j.scico.2012.06.002. URL: https://linkinghub.elsevier.com/retrieve/pii/S0167642312001128 (visited on 12/09/2022).

[40] Kaan Berk Yaman. *The Kconfig Variability Framework as a Feature Model: Sampled Configurations for Manual Evaluation*. Type: dataset. Mar. 2023. DOI: 10.5281/zenodo.7787304. URL: https://zenodo.org/record/7787304 (visited on 03/31/2023).

[41] *zephyrproject-rtos/zephyr*. original-date: 2016-05-26T17:54:19Z. Mar. 2023. URL: https://github.com/zephyrproject-rtos/zephyr (visited on 03/28/2023).

A. Appendix

```
config SYMBOL_A
  bool "Config symbol A"

config SYMBOL_B
  bool "Config symbol B"
  depends on SYMBOL_A

config SYMBOL_C
  bool "Config symbol C"
  depends on SYMBOL_B

config SYMBOL_X
  bool "Config symbol X"

config SYMBOL_Y
  bool "Config symbol Y"
  depends on SYMBOL_X
  depends on SYMBOL_C
```

Figure 18.: Contents of Kconfig1, which was verified automatically using `kmax`. Kconfig1 contains a multiple dependency and a dependency chain.

```
config SYMBOL_A
  bool "Config symbol A"

menu "Generic menu"

config SYMBOL_X
  bool "Config symbol X"

choice CHOICE_B
  bool "Choice block B"
  depends on SYMBOL_A
  depends on SYMBOL_X

  config SYMBOL_C
    bool "Config symbol C"

  config SYMBOL_D
    bool "Config symbol D"

endchoice

endmenu

choice CHOICE_E
  bool "Choice block E"
  optional

  config SYMBOL_F
    bool "Config symbol F"

  config SYMBOL_G
    bool "Config symbol G"
    depends on SYMBOL_C

endchoice
```

Figure 19.: Contents of Kconfig2, which was verified automatically using `kmax`. Kconfig2 contains an optional and a mandatory boolean choice block, one these being enclosed in a menu block.

```
 1 menu "Menu block M"
 2
 3   config SYMBOL_X
 4     bool "Configuration symbol X"
 5
 6   config SYMBOL_Y
 7     bool "Configuration symbol Y"
 8     depends on SYMBOL_W
 9
10   config SYMBOL_Z
11     bool "Configuration symbol Z"
12
13 endmenu
14
15 menuconfig SYMBOL_C
16   bool "menuconfig symbol C"
17
18 config SYMBOL_B
19   bool "Configuration symbol B"
20
21 config SYMBOL_D
22   bool "Configuration symbol D"
23   depends on SYMBOL_W
24   depends on SYMBOL_C
25
26 config SYMBOL_W
27   bool "Configuration symbol W"
```

Figure 20.: Contents of Kconfig3, which was verified automatically using `kmax`. Kconfig3 contains a menu block and a `menuconfig` symbol.

```
config SYMBOL_A
  bool "Configuration symbol A"

config SYMBOL_X
  bool "Configuration symbol X"
  select SYMBOL_A

menu "Menu block M"

  choice CHOICE_C
    bool "Choice block C"
    depends on SYMBOL_A

    config SYMBOL_B
      bool "Configuration symbol B"

    config SYMBOL_D
      bool "Configuration symbol D"

  endchoice

  choice CHOICE_E
    bool "Choice block D"
    depends on SYMBOL_X
    optional

    config SYMBOL_F
      bool "Configuration symbol F"

    config SYMBOL_G
      bool "Configuration symbol G"

  endchoice

endmenu
```

Figure 21.: Contents of Kconfig4, which was verified automatically using `kmax`. Kconfig4 contains a `select` dependency, two choice blocks enclosed in a menu block.

```
 1 config SYMBOL_A
 2   bool "Configuration symbol A"
 3
 4 config SYMBOL_B
 5   bool "Configuration symbol B"
 6   depends on SYMBOL_A
 7
 8 menuconfig SYMBOL_C
 9   bool "Menuconfig symbol C"
10   depends on SYMBOL_B
11
12 config SYMBOL_D
13   bool "Configuration symbol D"
14   depends on SYMBOL_C
15   depends on SYMBOL_F
16
17 config SYMBOL_F
18   bool "Configuration symbol F"
19
20 menuconfig SYMBOL_X
21   bool "Menuconfig symbol X"
22   depends on SYMBOL_C
23
24 config SYMBOL_Y
25   bool "Menuconfig symbol Y"
26   depends on SYMBOL_X
```

Figure 22.: Contents of Kconfig5, which was verified automatically using `kmax`. Kconfig5 contains a longer dependency chain, multiple dependencies on a single configuration symbol and a `menuconfig` symbol.

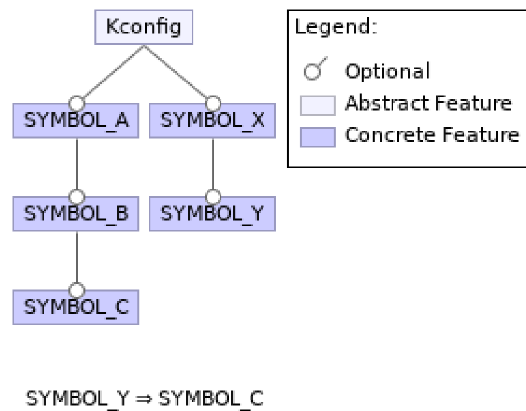


Figure 23.: This feature model was generated by Kfeature for Kconfig1.

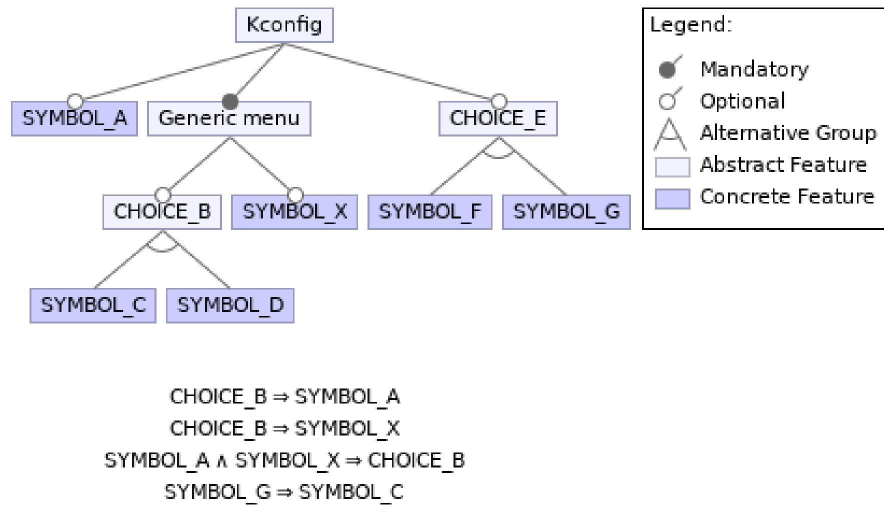


Figure 24.: This feature model was generated by Kfeature for Kconfig2.

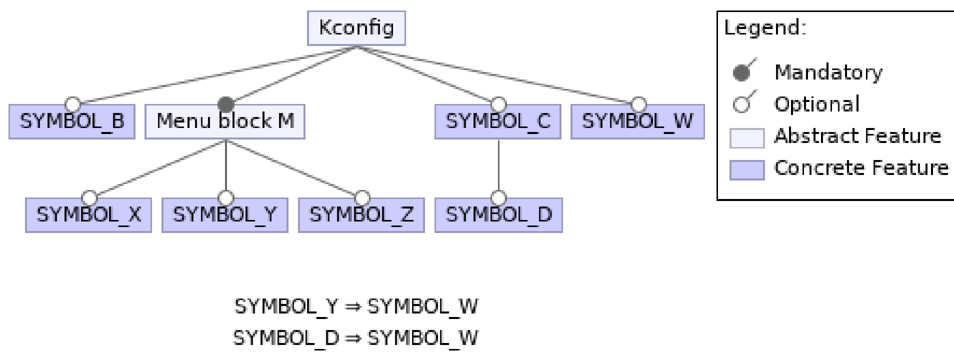


Figure 25.: This feature model was generated by Kfeature for Kconfig3.

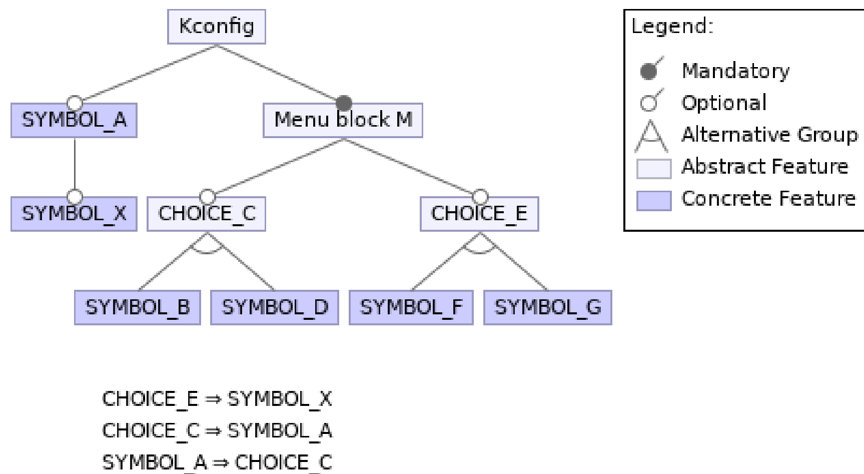


Figure 26.: This feature model was generated by Kfeature for Kconfig4.

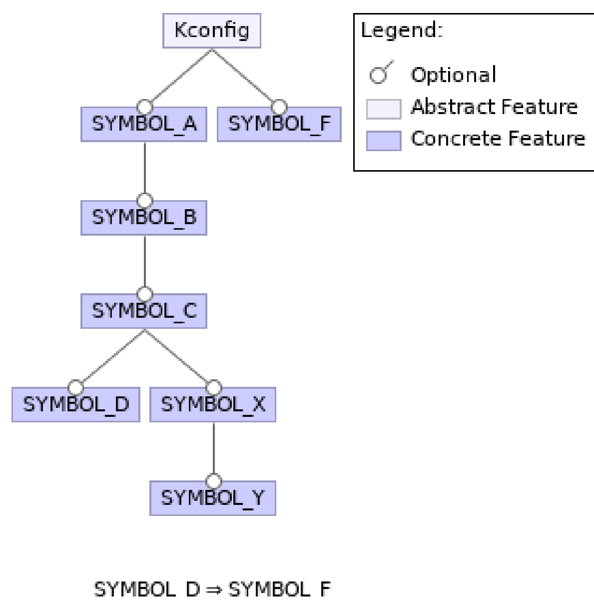
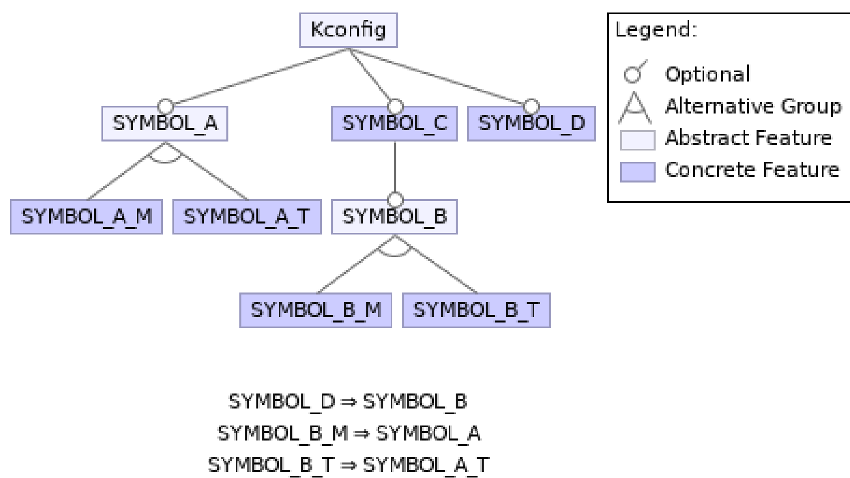


Figure 27.: This feature model was generated by Kfeature for Kconfig5.

```
1  config SYMBOL_A
2    tristate "Configuration symbol A"
3
4  config SYMBOL_B
5    tristate "Configuration symbol B"
6    depends on SYMBOL_A
7    depends on SYMBOL_C
8
9  config SYMBOL_C
10   bool "Configuration symbol C"
11
12 config SYMBOL_D
13   bool "Configuration symbol D"
14   depends on SYMBOL_B
```

Figure 28.: Contents of Kconfig6, which was verified manually through sampling. Kconfig6 contains a mixed-type multiple dependency, tristate and boolean configuration symbols.

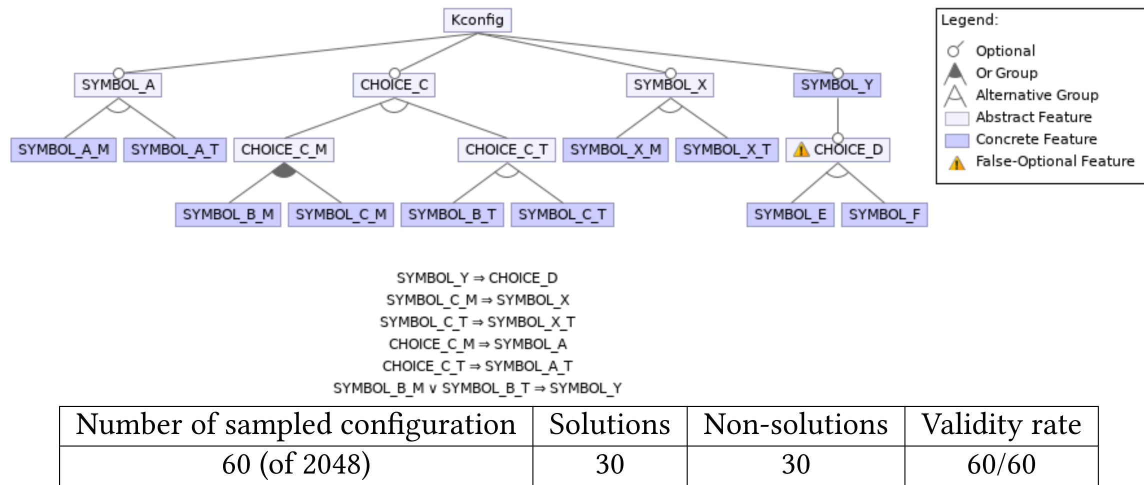


Number of sampled configuration	Solutions	Non-solutions	Validity rate
64 (of 64)	12	52	64/64

Figure 29.: This feature model was generated by Kfeature for Kconfig6. The given table contains the results of the verification of the sampled configurations. We have decided to widen the sample pool for this Kconfig file so that all possible configurations were considered whilst evaluation.

```
config SYMBOL_A
  tristate "Configuration symbol A"

choice CHOICE_C
  tristate "Choice block C"
  depends on SYMBOL_A
  optional

  config SYMBOL_B
    tristate "Configuration symbol B"
    select SYMBOL_Y

  config SYMBOL_C
    tristate "Configuration symbol C"
    depends on SYMBOL_X

endchoice

config SYMBOL_Y
  bool "Configuration symbol Y"

config SYMBOL_X
  tristate "Configuration symbol X"

choice CHOICE_D
  bool "Choice block C"
  depends on SYMBOL_Y

  config SYMBOL_E
    bool "Configuration symbol E"

  config SYMBOL_F
    bool "Configuration symbol F"

endchoice
```

Figure 30.: Contents of Kconfig7, which was verified manually through sampling. Kconfig7 contains a tristate choice block and a boolean choice block.

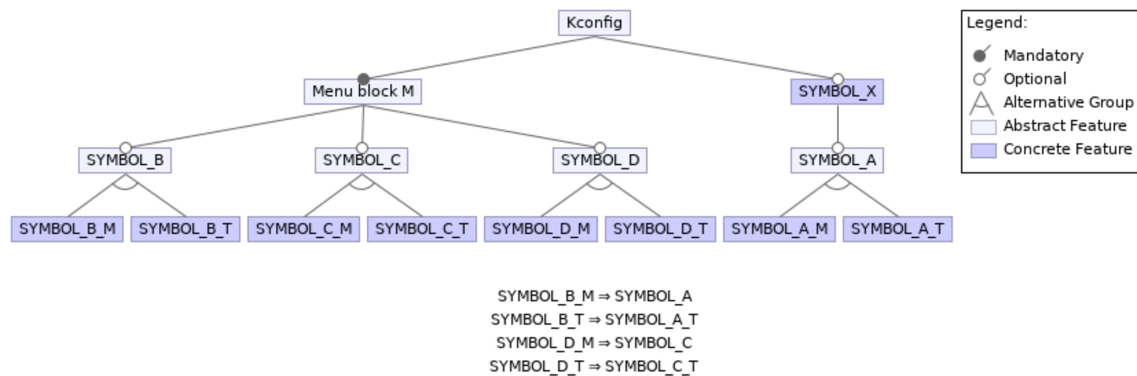


Number of sampled configuration	Solutions	Non-solutions	Validity rate
60 (of 2048)	30	30	60/60

Figure 31.: This feature model was generated by Kfeature for Kconfig7. The given table contains the results of the verification of the sampled configurations.

```
config SYMBOL_A
  tristate "Configuration symbol A"
  depends on SYMBOL_X

menu "Menu block M"

  config SYMBOL_B
    tristate "Configuration symbol B"
    depends on SYMBOL_A

  config SYMBOL_C
    tristate "Configuration symbol C"

  config SYMBOL_D
    tristate "Configuration symbol D"
    depends on SYMBOL_C

endmenu

menuconfig SYMBOL_X
  bool "Menuconfig symbol X"
```

Figure 32.: Contents of Kconfig8, which was verified manually through sampling. Kconfig8 contains a menu block, multiple tristate configuration symbols and a mixed-type dependency.

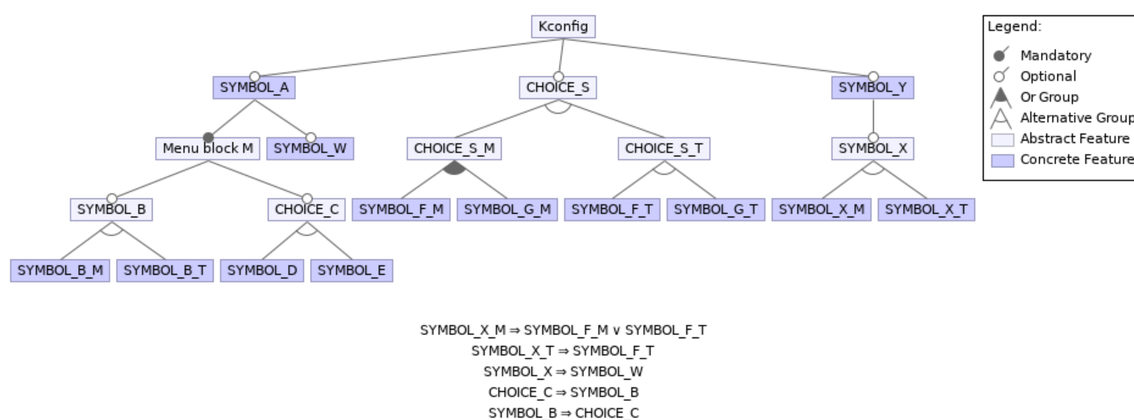


Number of sampled configuration	Solutions	Non-solutions	Validity rate
60 (of 512)	30	30	60/60

Figure 33.: This feature model was generated by Kfeature for Kconfig8. The given table contains the results of the verification of the sampled configurations.

```
config SYMBOL_A
  bool "Configuration symbol A"

menu "Menu block M"
  depends on SYMBOL_A

  config SYMBOL_B
    tristate "Configuration symbol B"

  choice CHOICE_C
    bool "Choice block C"
    depends on SYMBOL_B

    config SYMBOL_D
      bool "Configuration symbol D"

    config SYMBOL_E
      bool "Configuration symbol E"

  endchoice

endmenu

choice CHOICE_S
  tristate "Choice block S"

  config SYMBOL_F
    tristate "Configuration symbol F"

  config SYMBOL_G
    tristate "Configuration symbol G"

endchoice

config SYMBOL_W
  bool "Configuration symbol W"
  select SYMBOL_A

menuconfig SYMBOL_Y
  bool "Configuration symbol Y"

config SYMBOL_X
  tristate "Configuration symbol X"
  depends on SYMBOL_W
  depends on SYMBOL_Y
  depends on SYMBOL_F
```

Figure 34.: Contents of Kconfig9, which was verified manually through sampling. Kconfig9 contains a menu block, two choice blocks, a `menuconfig` configuration symbol and multiple mixed-type dependencies.



Number of sampled configuration	Solutions	Non-solutions	Validity rate
60 (of 8192)	30	30	60/60

Figure 35.: This feature model was generated by Kfeature for Kconfig9. The given table contains the results of the verification of the sampled configurations.