Hildesheimer Informatik-Berichte

Sascha El-Sharkawy, Adam Krafczyk, Nazish Asad, and Klaus Schmid

Analysing the KConfig Semantics and Related Analysis Tools

October 22, 2015

Report No. 1/2015, SSE 1/15/E

Software Systems Engineering • Institut für Informatik
Universität Hildesheim •Universitätsplatz 1 • D-31134 Hildesheim

Abstract

The Linux Kernel is often used as real world use case to demonstrate novel Software Product Line Engineering techniques. The large open source repository facilitates the analysis of the variability model, the instantiation process, the instantiable artefacts, and the evolution of all of them.

This report focusses on the analysis of undocumented KConfig functionalities. These functions have to be considered while applying any variability management technique to the Linux Kernel. Hence, this report will contribute to a better understanding how variability is handled in KConfig files.

Further, we analyse existing work, which also analysed KConfig. Based on the weak documentation of KConfig, these works contain errors. These errors threat the validity of many existing analysis of the Linux Kernel.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

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

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

Chapter 2 introduces KConfig, the textual variability language of Linux. We present the concepts of KConfig together with an example how they can be modelled. Further, we present our systematic analysis to find undocumented corner cases of the KConfig semantics. In Chapter 3, we show how well these corner cases are supported by analysis tools used for scientific work. In Section 4, we conclude and show future work.

Chapter 2

KConfig

KConfig is a textual language, which was developed to manage the variability of the Linux Kernel. Although it never became a standalone project, it was also used in several other projects [BSL+12]. Meanwhile KConfig has been analysed in multiple studies. This work often relies on the language description of KConfig [KCo14]. However, this language description is rather informal and ambiguous, which might be the reason that existing research papers do not always cover its real behaviour.

This chapter is intended to provide a better and clear understanding of the behaviour of KConfig. In the following section, we briefly describe the concepts of KConfig based on its language description. Section 2.2 is used to detect systematically potential misunderstandings related to the KConfig language specification. These misunderstandings are further elaborated in Section 2.4. In Chapter 3, we analyse and discuss how existing tools and papers handle our findings of Section 2.4.

2.1 Concepts

In this section, we briefly describe the most important variability management concepts of KConfig based on its language specification [KCo14]. We do not include concepts which do not affect KConfig's variability management logic, like `help`, `comments`, or `mainmenu`.

Config options

Config options (and also menuconfigs[1]) are the most used elements of the language and can be seen as variables. These config options can be of type `tristate`, `bool`, `string`, `hex`, or `int`. All but the first are known from typical programming languages (with the exception that `bool` ranges from `y` to `n`). Hence, we only describe `tristate` further.

Tristate config options encode the following alternatives:

n – The related feature will not be part of the resulting system.

y – The related feature will be a permanent part of the resulting system.

m – The related feature will be compiled as a module, which means that it can be flexibly loaded or unloaded at runtime.

[1] A menuconfig is similar to the simple config option, but it also gives a hint to configuration front-ends, that all sub options (cf. Hierarchies) should be displayed in a separate sub menu.

Constraints can restrict tristate config options in such a way that they can only be disabled or configured as modules.

There exist a special boolean config option `MODULES`[2], which modifies the semantics of the whole tristate support globally. If `MODULES` is set to `n` (false), all tristate config options become boolean config options, meaning that `m` is treated in all tristate configs as `y`.

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

- A prompt displays a name to the user and makes the config option visible to him. A config option without a prompt can not be configured by the user.
- The default attribute specify default values of a config option. These defaults can be changed by the user if the config option is visible. Furthermore, a config option can contain any number of (conditional) defaults.
- A range specifies a upper and a lower bound for config options of type int or hex.
- KConfig specifies 2 different kind of constraints to restrict the possible values of a config option. Also these constraints can be conditional, i.e. whether the constraint exist or not:
  - `depends on` is used to describe whether a config option can be configured or is disabled. These kind of constraints can also be used to restrict `tristate` config options to be selected only as `n` or `m`. The Kernel developer name this kind of constraints as the restriction of the upper limit of a config option. Contrary to the other attributes, this attribute can not be made conditional.
  - `select` is used to specify a lower bound. The current value of the surrounding config option will be used as lower bound for the selected config option. If a config option is selected multiple times, it is set to the largest selection. It is also possible to select a config option without fulfilling its `depends on` constraints.
- `option env="<environment variable>"` can be used to load the value an environment variable into KConfig as a default value for a config option. Current Kernel versions use this functionality to load the Kernel version and the target architecture from its makefiles.
- We omitted further functionalities which could be used to support the user during the configuration process, but which do not affect the configuration logic of KConfig. For instance `defconfig_list` or `allnoconfig_y`. It is also possible to use undefined config options inside of conditions or constraints, e.g. because they are part of another architecture. Such config options will be interpreted as `n` (false).

[2]While older versions of KConfig need at least a config option which is named as `MODULES`, the current version of KConfig needs that at at most one arbitrary config option is attributed with `option modules` for controlling the tristate semantics. The Linux Kernel still uses the (attributed) `MODULES` config option for doing this.

Choices

A choice groups several config options and can only be of type `bool` or `tristate`. In case of a boolean choice, exactly one config option must be selected. A tristate choice allows the selection of multiple elements as modules. Thus, a choice can be treated as a XOR-decomposition (boolean choice) or as an OR-decomposition (tristate choice). A choice can be attributed with the same attributes as a config option. A choice can be `optional`, which allows to set the choice to `n` (false) and none of the contained config options may be selected.

Hierarchies

KConfig offers two possibilities to structure config options in hierarchies:

- Any number of config options (and other elements like choices) can be surrounded by the keywords `menu` and `endmenu` to place them in a sub menu. These sub menu can also contain other menus. Hence, this concept allows to model arbitrary hierarchical structures.

  The visibility of these menus can be restricted with the attribute `visible if`. This attribute is only applicable for menus and restrict the whole menu including all nested elements. However, these elements still exists. Hence, they can be selected or modified by constraints or default values, but not by the user.
- Constraints and if-Statements offer an alternative to model hierarchies. If the visibility of a config option B depends on another config option A and B is directly written below A in a KConfig file, than B will also be displayed in a hierarchical structure below A inside configuration front-ends.

File inclusions

KConfig allows to split a huge variability model into separate files. The source statement includes the contents of another KConfig file into the current file. This concept facilitates arbitrary nested KConfig files.

if-Statements

It is possible to surround the elements from above with `if` and `endif` to make their existence depending from a condition. For instance, this is used for defining the same config option twice with different attributes or for including different KConfig files depending on the target architecture.

Constraint logic

KConfig uses a tristate logic instead of a boolean logic for creating and evaluating expressions. This kind of logic is hard to translate into a pure boolean logic, which is the objective of many tools from Chapter 3. The most important distinctions to the usual boolean algebra are:

- The tristate logic of KConfig has three constants: `n` (=0), `m` (=1), and `y` (=2).
- Comparisons (`=` and `!=`) can be applied on all type of config options.
- `Bool` and `tristate` config options are evaluated to their current values. All other config options (`string`, `hex`, and `int`) are treated as `n`.
- `!<expression>` returns the result of `2 - expression`. This means that `!m` has the same value as `m`.

- `<expression 1> && <expression 2>` returns the result of `min(<expression 1>, <expression 2>)`.
- `<expression 1> || <expression 2>` returns the result of `max(<expression 1>, <expression 2>)`.

2.2 Systematic Analysis of KConfig's Capabilities

In Table 2.1 on the following page, we analyse how different concepts of KConfig do interact if they are combined. Interactions which are not obvious and may lead to misunderstandings are discussed in more detail in Section 2.3. Critical observations which must be considered during translating the KConfig model to another logical model like boolean formula are discussed in Section 2.4.

In Table 2.1, we show how the different concepts of Section 2.1 interact if they are combined. There, we tried to include the elements of the columns into the elements of the rows. For instance ("Config options" × "Choices") shows that choices cannot be embedded into config options, while the other way round is a well defined functionality of KConfig.

The meaning of "Hierarchies" and "Constraints" inside the table need further explanation to be understand correctly. While "Constraints" is used to check whether `depends on` or `select` statements can be nested inside other elements to restrict the visibility or to set a specific value of other elements, we also analysed whether hierarchies can be created using `depends on` or `if` statements. The later one is expressed with "Hierarchies → via Constraints". For instance, ("Hierarchies → menu" × "Hierarchies → via Constraints") denotes that it is not possible to structure config options defined outside of the menu structure via `depends on` or `if` statements into the menu structure. But ("Hierarchies → menu" × "depends") show that in general `depends on` statements can be used directly inside a menu without additional config options between them.

For a complete understanding it is also important to know how the different types of constraints do interact. In Section 2.1 we already mentioned that `select` statements are able to select config options without fulfilling their `depends on` constraints. `if` statements can be ignored in the same way. Thus, `if` and `depends on` constraints have a lower precedence than `select` statements and are only evaluated if no `select` statement is selecting the same config option. `if` and `depends on` statements can be combined arbitrary to model a dependency between different config options. Multiple `depends on` and `if` statements are combined via a logically AND relation. Thus, `if` and `depends on` statements have the same precedence. Attributes like `default`s and `prompt`s become only active if all `depends on` and `if` statements got fulfilled. Invisible config options with a `default` value will be set automatically to this default value. Thus, attributes have the lowest precedence and default values of a invisible config option can be treated as an assignment.

The precedence of the different constraint types can be summarized as follows:

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

contains ⟶ ↑		Config options				Choices		Hierarchies		Constraints		Attributes				If
		bool	tristate	string	numerical	bool	tristate	menu	via Constraints	depends	select	prompt	default	range	visible if	
Config options	**bool**	✗				✗		✗	✓	switch visibility	Sec. 2.4.1	✓	Sec. 2.4.2	—	✗	✗
	tristate								✓	Sec. 2.3.2	Sec. 2.4.1		Sec. 2.4.2	—		
	string								Sec. 2.3.1	switch visibility	—		✓	—		
	numerical								Sec. 2.3.1	switch visibility	—		Sec. 2.3.3	Sec. 2.3.4		
Choices	**bool**	✓	Sec. 2.4.3	Sec. 2.3.5		Sec. 2.3.6		✗	Sec. 2.4.4 & Sec. 2.4.5	Sec. 2.4.6	Sec. 2.3.7	Sec. 2.4.7	✓	Sec. 2.3.7	✗	Sec. 2.4.8
	tristate	Sec. 2.4.3	✓													
Hierarchies	**menu**	✓				✓		✓	✗	✓	✗	✓	✗	✗	✓	✓
	Constraints								✓		✓		✓	✓	✗	Sec. 2.3.8
Constraints	**depends**	✓		—		—		—	✓	✗		✗				✗
	select							✗	Sec. 2.3.9							✓
Attributes	**prompt**	✗				✗		✗		✗		Sec. 2.4.9				✓
	default															
	range															
	visible if															
If		✓				✓		✓	Sec. 2.3.8	✗		✗				✓

Table 2.1: Systematic Analysis of Undocumented Functionalities (✗ = not supported by the grammar; — = can be modelled, but has no impact; ✓ = well defined by KConfig).

2.3 Non-Critical Observations

In this section, we describe observations, which are not critical for a formal transformation. These are capabilities of modelling inconsistent models, graphical representations or unimportant inconsistencies of the KConfig language specification. We also show capabilities of KConfig which are mentioned in its language specification, but are not obvious for non KConfig experts. In Section 2.4, we describe more critical observations, which must be considered during a formal transformation.

2.3.1 Hierarchies inside String/Numerical Config Options

It is possible to use non `bool`/`tristate` config options inside `depends on` constraints, but this will lead in permanently visible/invisible config options. The reason behind this lies inside the constraint logic of KConfig (cf. Section 2.1 on page 15), which treats all non boolean/tristate elements as `n` constants:

- A config option will become permanently invisible if it `depends on` a `string`, `int`, or `hex` config option (e.g. `depends on STR_OPTION`).
- A config option will become permanently visible if it `depends` **`not`** `on` a `string`, `int`, or `hex` config option (e.g. `depends on !STR_OPTION`).

Nonetheless, this kind of constraints can still be used to model hierarchies. The depending config options will still indented below the non `bool`/`tristate` config option, but the selection of the non `bool`/`tristate` config option will not affect the visibility of the nested config options.

```
1  config STRING_VAR1
2      string "1st String Variable"
3
4      config SUB_VAR_1
5          bool "1st Boolean Sub Variable"
6          depends on !STRING_VAR1
7
8   config STRING_VAR2
9      string "2nd String Variable"
10
11     config SUB_VAR_2
12         bool "2nd Boolean Sub Variable"
13         depends on STRING_VAR2
```

Figure 2.1: KConfig code for a hierarchy inside a string config option.

Figure 2.2 on the following page shows that “1st Boolean Sub Variable” is intended below “1st String Variable”. But this config option is permanently visible independent of the value of “1st String Variable”.

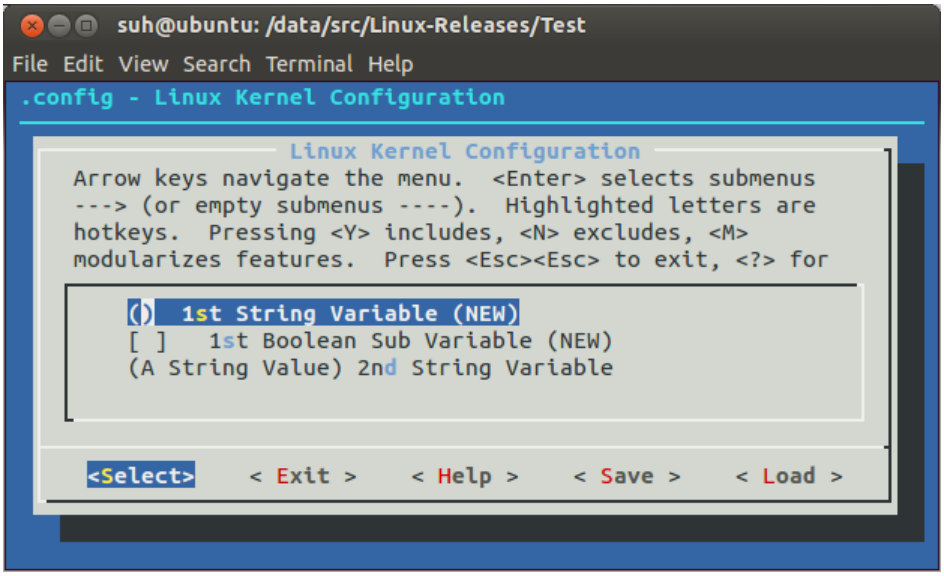

Figure 2.2: Menuconfig execution of Figure 2.1. Selection of "1st/2nd String Variable" has no affect to the visibility of "1st/2nd Boolean Sub Variable".

Comparisons in form of `string/numerical` config option `=` or `!= <value>` can be used for modelling evaluable constraints. Such constraints will not be treated as constant `n` and will be evaluated like constraints on config options of type `bool` or `tristate`.

2.3.2 Depends for Tristate Config Options

`depends on` constraints can be used to specify conditional visibility of a symbol, e.g. a config option, rather this kind of constraints restrict the upper bound of a symbol. However, only the range of `tristate` config option can be restricted, since they are the only one symbols with more than two states. For instance, the constant `m` can be used to avoid the permanent selection (`y`), but this will also completely deselect the config option if module support is globally disabled.

```
1 config MODULES
2     bool "En-/Disable Tristate Support"
3     option modules
4
5 config TRISTATE_VAR
6     tristate "A Tristate Variable"
7     depends on m
```

Figure 2.3: KConfig code for a tristate config option with only two states (`y` is no longer available).

Figure 2.4 on the next page demonstrates how menuconfig handles `depends on` constraints for `tristate` config options (cf. Figure 2.3). The constraint in line 7 avoids that "A Tristate Variable" is set to `y`.

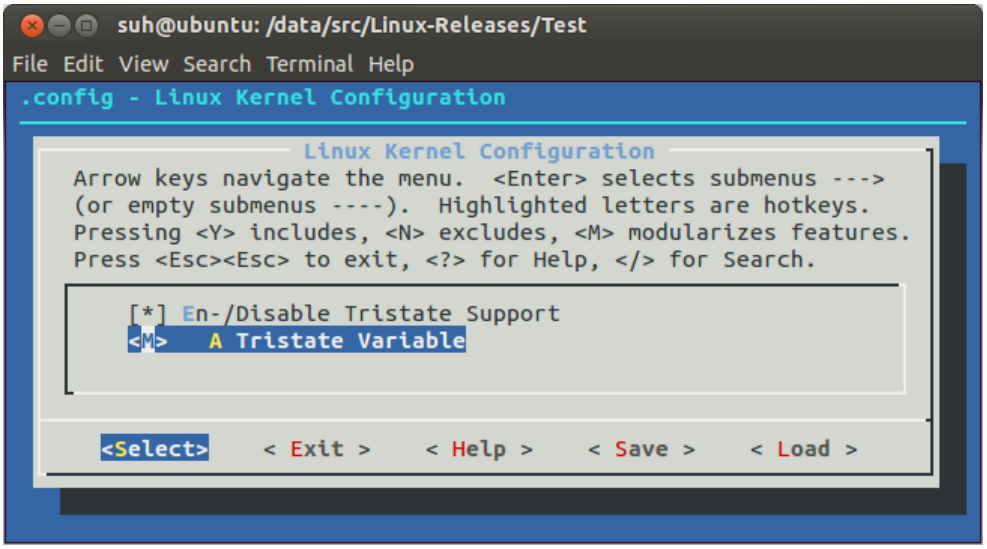

Figure 2.4: Menuconfig execution of Figure 2.3. If "En-/Disable Tristate Support" is selected, "A Tristate Variable" can only be selected as `n` or `m` (otherwise it is permanently selected as `n`).

2.3.3 Numbers with String Defaults

Since KConfig is not a real type-safe language, it is possible to specify a `default` value which is not suitable for the type of the config option. An example is shown in Figure 2.5.

```
config INT_VAR
    int "Integer Variable"
    default "Hello World"

config HEX_VAR
    hex "Hex Variable"
    default "Hello World"
```

Figure 2.5: KConfig code of numerical config options with a `string` as `default` value.

Figure 2.6 shows how the model from Figure 2.5 is saved if the user does not change the default values. The saved values do not match to the type of the config options. Further, the `string` values are not surrounded by quotes as it is usually the case (cf. Figure 2.10 on page 22).

```
#
# Automatically generated file; DO NOT EDIT.
# Linux Kernel Configuration
#
CONFIG_INT_VAR=Hello World
CONFIG_HEX_VAR=Hello World
```

Figure 2.6: Saved `.config` file of the example from Figure 2.5.

Note: Since the KConfig logic does not check such constructs, a good analysis tool could find such constructs and display a warning.

2.3.4 Range of Numerical Config Options

`int` and `hex` config options can also be attributed with an attribute `range <lower bound> <upper bound>`. This will limit the user to enter values which are larger than or equal to the *<lower bound>* and smaller than or equal to the *<upper bound>*. The *<lower bound>* is also used as default value, if the variable is visible to the user.

```
1  config NUM_LOWER_BOUND
2      int "Lower Bound"
3      default 2
4  config NUM_UPPER_BOUND
5      int "Upper Bound"
6      default 16
7
8  config NUM_VAR
9      int "A Numerical Variable"
10     range NUM_LOWER_BOUND NUM_UPPER_BOUND
```

Figure 2.7: KConfig code of a range specification.

Figure 2.8 demonstrates how menuconfig handles `range` definitions (cf. Figure 2.7). The value of "Lower Bound" is also used as the default value for "A Numerical Variable". The "(new)" behind its name indicates that there was no user input.

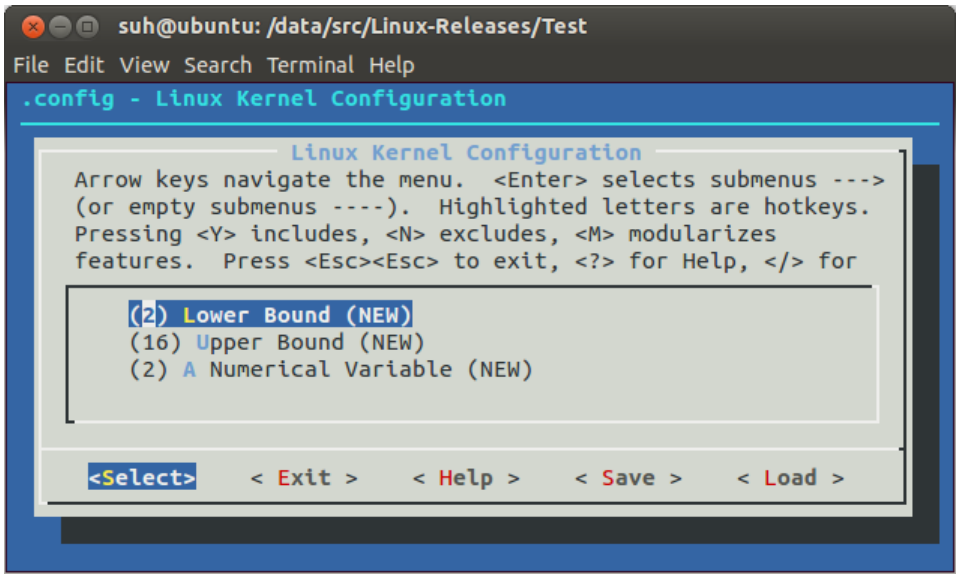

Figure 2.8: Menuconfig execution of Figure 2.7. Although "A Numerical Variable" has not an explicit specification of a `default` value, it is set to 2 through its `range` definition.

It is also possible to specify an illegal range where the *<lower bound>* has an higher value than the *<upper bound>*. Also in this case, the *<lower bound>* is used as a default value, but the user is no longer able to change the value.

Note: Since the KConfig logic does not check such conditions, a good analysis tool could find such situations and display a warning.

2.3.5 Choices with nested Strings/Numerical Config Options

KConfig's language specification [KCo14] specifies only choices of type `bool` or `tristate`, which allows the selection of config options of the same type. Nonetheless, the grammar does not avoid the creation of nested `string` or numerical config options. But after the user has changed the value of the choice, he is no longer able to reselect the `string` value. He is also not able to change the value of the `string` config option.

```
1  choice
2      bool "A Choice"
3      default STR_VAL
4
5      config STR_VAL
6          string "A String Value"
7          default "Hello World"
8
9      config BOOL_VAL
10         bool "A Boolean Value"
11 endchoice
```

Figure 2.9: KConfig code of a corrupt choice holding a string config option.

Figure 2.9 illustrates how a nested `string` option yield in a corrupt model. The nested **string** config option is selected by default. This `string` value will also be saved to the `.config` file, if the user does not change the value of the choice (cf. Figure 2.10).

```
1 #
2 # Automatically generated file; DO NOT EDIT.
3 # Linux Kernel Configuration
4 #
5 CONFIG_STR_VAL="Hello World"
6 # CONFIG_BOOL_VAL is not set
```

Figure 2.10: Saved `.config` file of the example from Figure 2.9.

Note: Since the KConfig logic does not check such illegal constructs, a good analysis tool could find such constructs and display a warning.

2.3.6 Choices Nested in Other Choices

Description

It is not possible to nest choices directly in other choices. The grammar is not supporting nesting choices directly in other choices. Only by using an `if`, it is possible to make a choice dependant of the selection of a nested config option and nest it together with the config option inside the surrounding choice.

Real world example from Linux

- **Kernel version:** 3.19
- **File:** `drivers/usb/gadget/legacy/Kconfig`
- **Line(s):** 446 – 464, nested in
- **File:** `drivers/usb/gadget/Kconfig`
- **Line(s):** 199 – 428
- **Location inside the menu of menuconfig:**

```
Prompt: EHCI Debug Device mode
 -> Device Drivers
  -> USB support (USB_SUPPORT [=y])
   -> USB Gadget Support (USB_GADGET [=m])
    -> USB Gadget Drivers (<choice> [=m])
     -> EHCI Debug Device Gadget (USB_G_DBGP [=m])
Depends on: <choice> && USB_G_DBGP [=m]
Selected by: <choice> && USB_G_DBGP [=m] && m
```

Minimal example

```
config MODULES
    def_bool y
    option modules

choice
    tristate "Top Level Choice"

    config VAL_1
        tristate "Value 1"

    config VAL_2
        tristate "Value 2"

    if VAL_2
        choice
            tristate "Nested Choice"

            config SUB_VAL_1
                tristate "Nested Value 1"

            config SUB_VAL_2
                tristate "Nested Value 2"
        endchoice
    endif
endchoice
```

Figure 2.11: KConfig code for creating choice nested in another choice.

Behaviour of KConfig

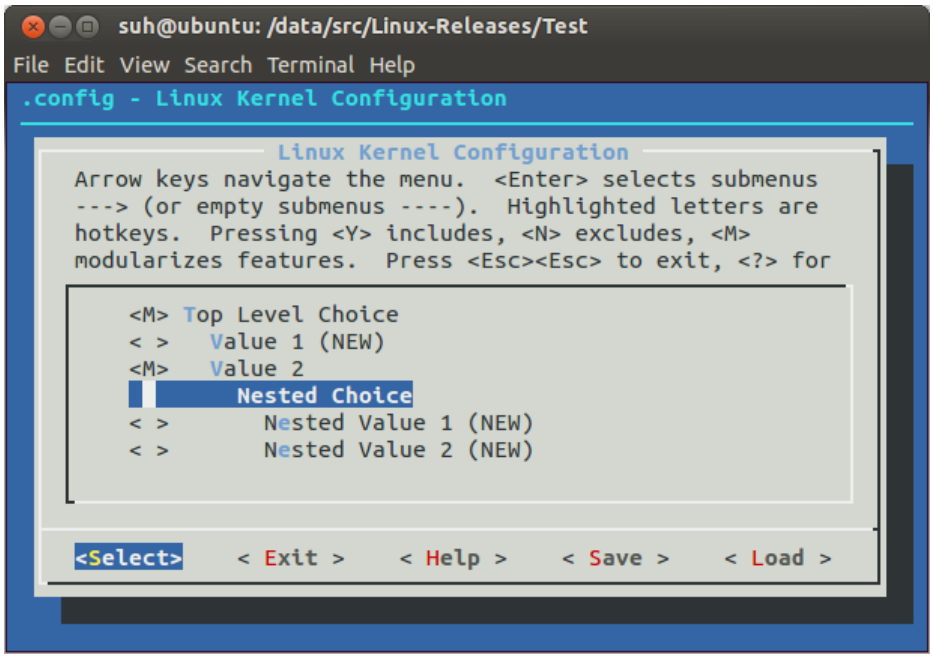

Figure 2.12: Menuconfig execution of Figure 2.11.

Figure 2.12 shows how menuconfig displays the example from Figure 2.11. "Nested Choice" is intended below "Value 2". Such constructs do only affect the presentation inside the configuration front-ends and do not differ logically from a choice which is dependant of another choice but not modelled inside a hierarchy.

2.3.7 Prohibited Attributes for Choices

Although KConfig [KCo14] specifies that all attributes can be used inside a choice group, the application of some attributes is prohibited by the grammar. These are attributes which usage would not make sense inside a choice group, like `select` or `range`.

2.3.8 If Used in Constraint Hierarchies

`if` statements can be used to make the existence of a config option conditional. This kind of constraints can also be used to model constraint hierarchies like `depends on` statements. However, the combination of both kinds of statements can break the constraint hierarchy. Figure 2.13 on the following page gives an example, where the combination of both statements breaks such a constraint hierarchy. Although the existence of `NESTED_VAR_1`, `NESTED_VAR_2`, and `NESTED_VAR_3` are dependant of `VAR_2` (see `depends on` constraints) and they are written below this config option, only `NESTED_VAR_1` will be indented below `VAR_2`.

```
1  config VAR_1
2      bool "Variable 1"
3
4  config VAR_2
5      bool "Variable 2"
6
7      config NESTED_VAR_1
8          bool "Nested Variable 1"
9          depends on VAR_2
10
11     if VAR_1
12         config NESTED_VAR_2
13             bool "Nested Variable 2"
14             depends on VAR_2
15     endif
16
17     config NESTED_VAR_3
18         bool "Nested Variable 3"
19         depends on VAR_2
```

Figure 2.13: KConfig code for breaking a constraint hierarchy.

The condition of the `if` statement in line 11 must use `VAR_2` in order that the hierarchy becomes restored. This is shown in Figure 2.14. The `depends on` constraint from line 14 has been removed as this is now part of the `if` condition in line 11.

```
1  config VAR_1
2      bool "Variable 1"
3
4  config VAR_2
5      bool "Variable 2"
6
7      config NESTED_VAR_1
8          bool "Nested Variable 1"
9          depends on VAR_2
10
11     if VAR_1 && VAR_2
12         config NESTED_VAR_2
13             bool "Nested Variable 2"
14     endif
15
16     config NESTED_VAR_3
17         bool "Nested Variable 3"
18         depends on VAR_2
```

Figure 2.14: Fixed hierarchy of example from Figure 2.13.

Usually this behaviour should not be a problem as this only affects the presentation inside the configuration front-ends. However, inside a `choice` (cf. Section 2.4.5) both kinds of constraints cause a different logical behaviour, which has to be considered.

2.3.9 Selected Config Option of a Constraint Hierarchy

KConfig [KCo14] specifies that `select` statements will select a variable independently whether existing `depends on` constraints get violated. Hence, it is possible to select a config option of a constraint hierarchy, while its parent is still deselected. In this case, the selected config option will not be displayed as long the hierarchy constraints are violated, but the correct value will be part of the `.config` file. Menuconfig is also displaying a warning. Also the the Kernel developers [KCo14] warn that `select` statements should be used with care and should not select config options having a `prompt`.

2.4 Critical Observations

In this section, we describe critical observations which must be considered during formal transformations of KConfig models to other languages, e.g. boolean formula. The observed behaviour of KConfig does not follow clearly from its language specification.

2.4.1 Selection of Nested Config Options

Description

A `select` constraint should take the value from the containing `bool`/`tristate` config option and should set this value as minimal value for the selected config option independent of existing `depends on` constraints. However, this does not apply for config options which are nested inside a choice.

Real world example from Linux

- **Kernel version:** 3.19
- **File:** `arch/x86/Kconfig.cpu`
- **Line(s):** 2 – 279
- **File:** `arch/x86/Kconfig`
- **Line(s):** 529 – 538
- **Location inside the menu of menuconfig:**

```
Prompt: 486
 -> Processor type and features
  -> Processor family (<choice> [=y])
Depends on: <choice> && X86_32 [=y]
Selected by: X86_RDC321X [=y] && X86_32 [=y] &&
X86_EXTENDED_PLATFORM [=y]
```

Minimal example

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

Figure 2.15: KConfig code for an inoperable selection of a nested config option inside a choice.

Behaviour of KConfig

Figure 2.16 shows that the `select` statement of Figure 2.15 on the previous page is inoperable. Even if "A Boolean Variable" is set to `y`, the `select` statement in line 3 will have no effect (cf. Figure 2.17). However, this behaviour avoids the multiple selection of nested config options of the same choice. Thus, we assume that this behaviour was indeed indented by the developers.

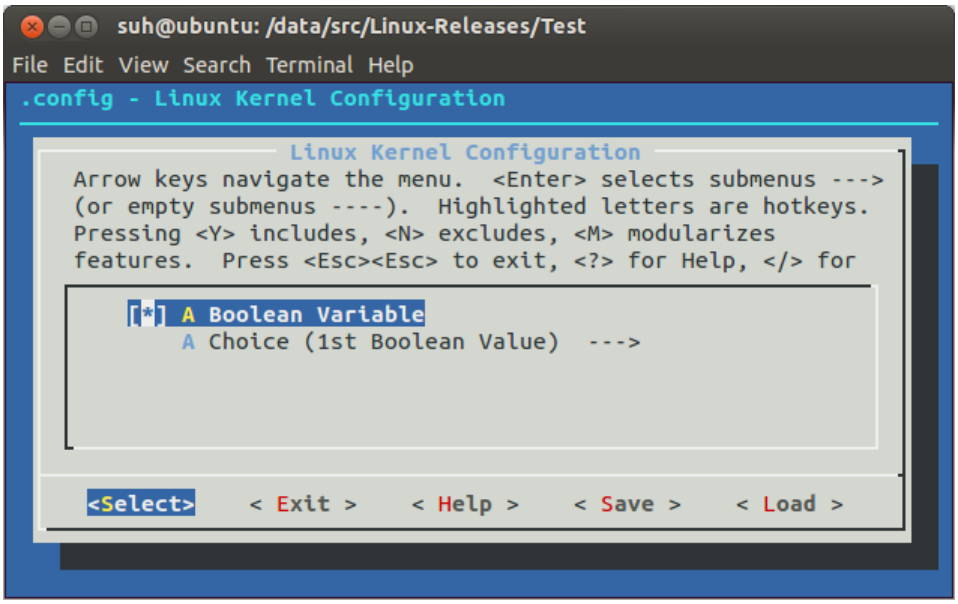

Figure 2.16: Menuconfig execution of Figure 2.15

Resulting .config file

```
1 #
2 # Automatically generated file; DO NOT EDIT.
3 # Linux Kernel Configuration
4 #
5 CONFIG_BOOL_VAR=y
6 CONFIG_BOOL_VAL1=y
7 # CONFIG_BOOL_VAL2 is not set
```

Figure 2.17: Saved `.config` file of the example from Figure 2.16 ("A Boolean Variable was selected").

2.4.2 Default Value `m`

Description

It is possible to specify `m` as `default` value for `bool` as well as for `tristate` config options, even if the the third state is not activated (cf. Section 2.1 on page 14).

Real world example from Linux

- **Kernel version:** 3.19
- **File:** `net/netfilter/Kconfig`
- **Line(s):** 63 – 74
- **Location inside the menu of menuconfig:**

```
Prompt: Connection tracking security mark support
 -> Networking support (NET [=y])
  -> Networking options
   -> Network packet filtering framework (Netfilter) (NETFILTER
[=y])
    -> Core Netfilter Configuration
Depends on: NET [=y] && INET [=y] && NETFILTER [=y] &&
NF_CONNTRACK [=m] && NETWORK_SECMARK [=y]
```

Minimal example

```
1  config MODULES
2      bool "Enable Tristate Support"
3      option modules
4
5  config VAR1
6      bool "Boolean Variable"
7      default m
8
9  config VAR2
10     tristate "Tristate Variable"
11     default m
```

Figure 2.18: KConfig code for specifying `m` as `default` value for a config option of type `bool`.

Behaviour of KConfig

The `default` expression is evaluated to `y` for both kinds of config options as long the third state is deactivated. If the third state is enabled, `tristate` config options will be set to `m`. `bool` config options will still be set to `y`.

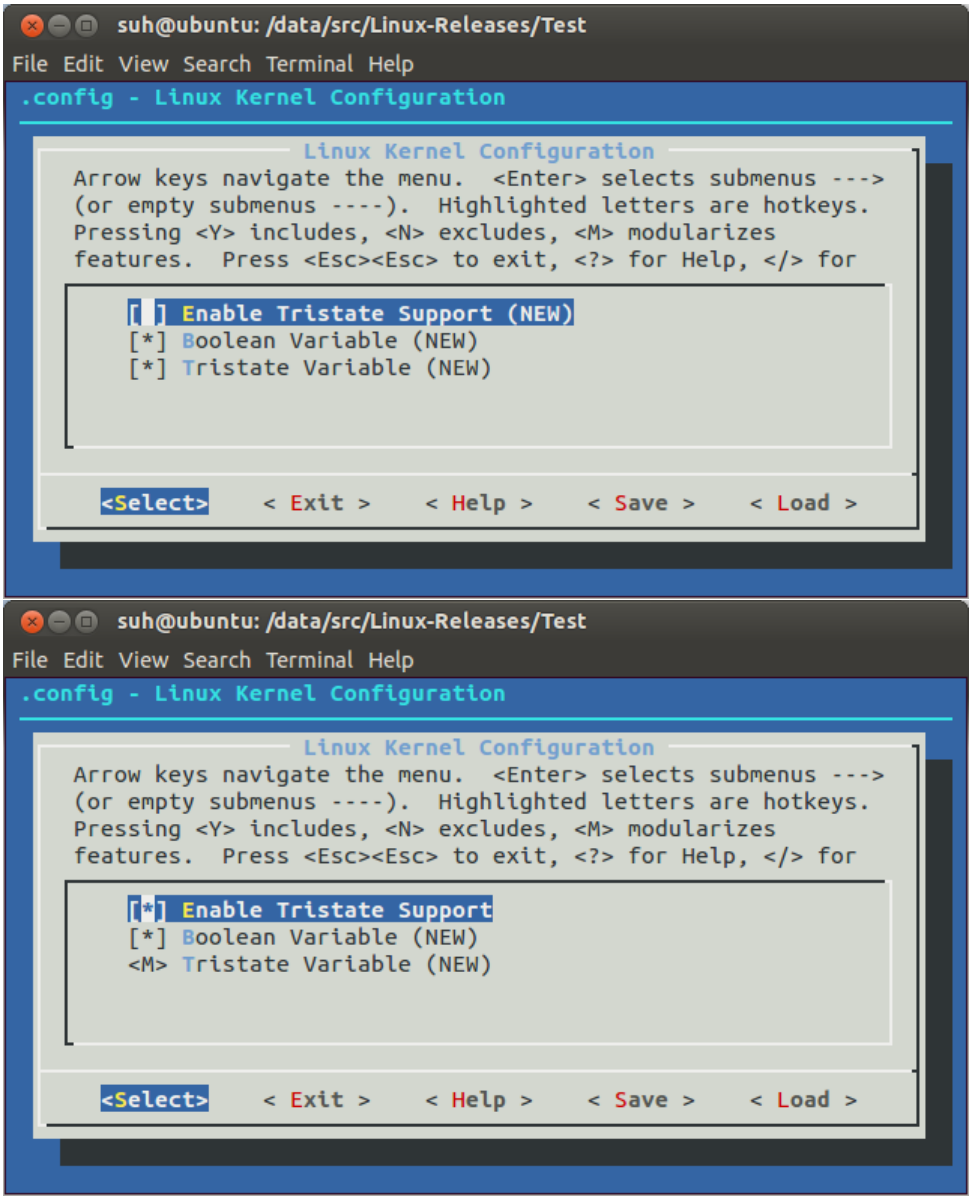

Figure 2.19: Menuconfig execution of Figure 2.18

Consequence

The real value of a "`default m`" expression depends from the type of the config option and whether the thrid state was activated. This is illustrated in Figure 2.19, which shows how the model from Figure 2.18 on the previous page is handled by menuconfig. The upper image shows how the `default` values are handled by `tristate`/`bool` config options without any user interaction. The saved configuration is shown in Figure 2.20 on the following page. After the user has activated the third state (2nd image), the `default` value of the `tristate` config option has been changed automatically.

Resulting .config file

```
#
# Automatically generated file; DO NOT EDIT.
# Linux Kernel Configuration
#
# CONFIG_MODULES is not set
CONFIG_VAR1=y
CONFIG_VAR2=y
```

Figure 2.20: Saved `.config` file of the example from Figure 2.19 (no user input).

2.4.3 Tristate Choice with Boolean Config Options

Description

Choices can be either of type `bool` or `tristate`. However, a `tristate` choice must not necessarily contain only `bool` config options. It is also possible to nest `tristate` config options in a choice of type `bool` and vice versa.

Real world example from Linux

No example found in Kernel 3.19.

Minimal example

```
config MODULES
    bool "Enable Tristate Support"
    option modules

choice
    tristate "Tristate Choice"

    config TRISTATE_VAL_1
        tristate "Tristate Value 1"
    config TRISTATE_VAL_2
        tristate "Tristate Value 2"
    config BOOL_VAL_1
        bool "Boolean Value"
endchoice
```

Figure 2.21: KConfig code for creating a tristate choice with nested boolean config options.

Behaviour of KConfig

Nested `bool` config options will only be selectable if a the choice is set to `y` (cf. Figure 2.22, upper image). In this case, the selected config option will aslo be set to `y`, independently whether the config option is of type `bool` or `tristate` (cf. Figure 2.23 on page 34). If the choice is set to `m`, only the tristate config options will be selectable as modules and all `bool` config options become permanently invisible (cf. Figure 2.22, 2nd image).

A choice of type `bool` containing `tristate` config options, facilitates the selection of exact one config option. The selected config option will be set to `y` independent of its type (same result as in Figure 2.23 on the following page).

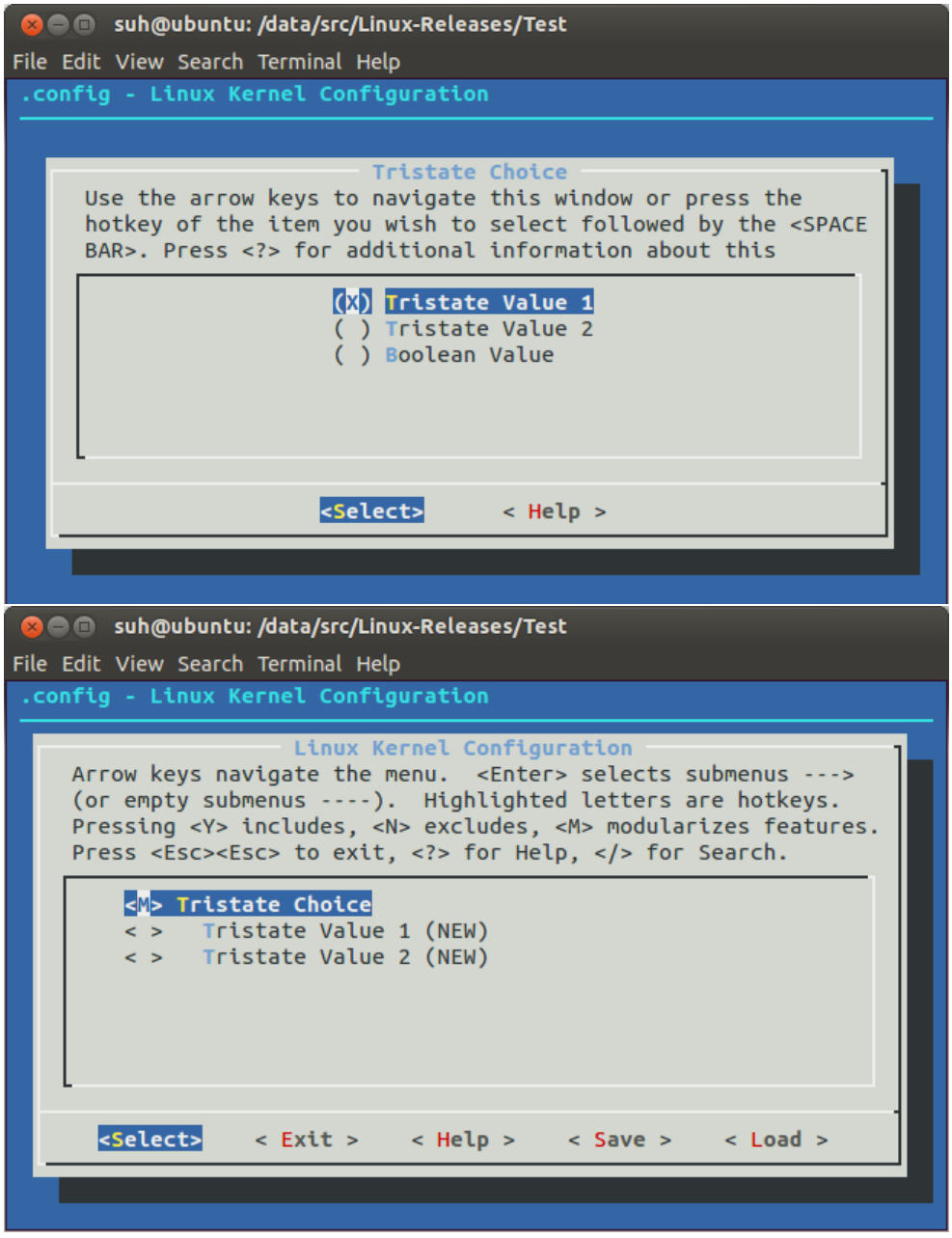

Figure 2.22: Menuconfig execution of Figure 2.21. Selection of “Boolean Value” is only possible, if the choice is set to `y`.

Resulting .config file

```
#
# Automatically generated file; DO NOT EDIT.
# Linux Kernel Configuration
#
CONFIG_MODULES=y
CONFIG_TRISTATE_VAL_1=y
# CONFIG_TRISTATE_VAL_2 is not set
# CONFIG_BOOL_VAL_1 is not set
```

Figure 2.23: Saved `.config` file of the example from Figure 2.22. The selection form the upper image was saved.

2.4.4 Structured Choices

Description

Config options of a choice can hold arbitrary `depends on` constraints. This `depends on` constraints may also be used to model dependencies between `config` entries of the same choice. In this case, the depended config options may only be selected if the affiliated config option was selected before. This is used for a more detailed configuration of selected config options of a choice (cf. Hierarchies in Section 2.1 on page 15).

Real world example from Linux

- **Kernel version:** 3.19
- **File:** `drivers/usb/gadget/Kconfig`
- **Line(s):** 199 – 428
- **Location inside the menu of menuconfig:**
  `Prompt: USB Gadget Drivers`
  `->Device Drivers`
  `→ USB support (USB\SUPPORT [=y])`
  `→ USB Gadget Support (USB\GADGET [=m])`

Minimal example

```
1  choice
2      bool "A structured Choice"
3
4      config VAL_1
5          bool "Value 1"
6      config VAL_2
7          bool "Value 2"
8
9      config SUB_VAL
10         bool "A sub value"
11         depends on VAL_2
12 endchoice
```

Figure 2.24: KConfig code for creating a structured choice ("A sub value" can only be selected together with "Value 2").

Behaviour of KConfig

Figure 2.25 demonstrates how KConfig handles the minimal example from Figure 2.24. As documented, exactly one of the first two config options has to be selected. Through the `depends on` constraint in line 11, "A sub value" may also be selected if "Value 2" was selected before. This scenario allows the selection of two config options for a boolean choice.

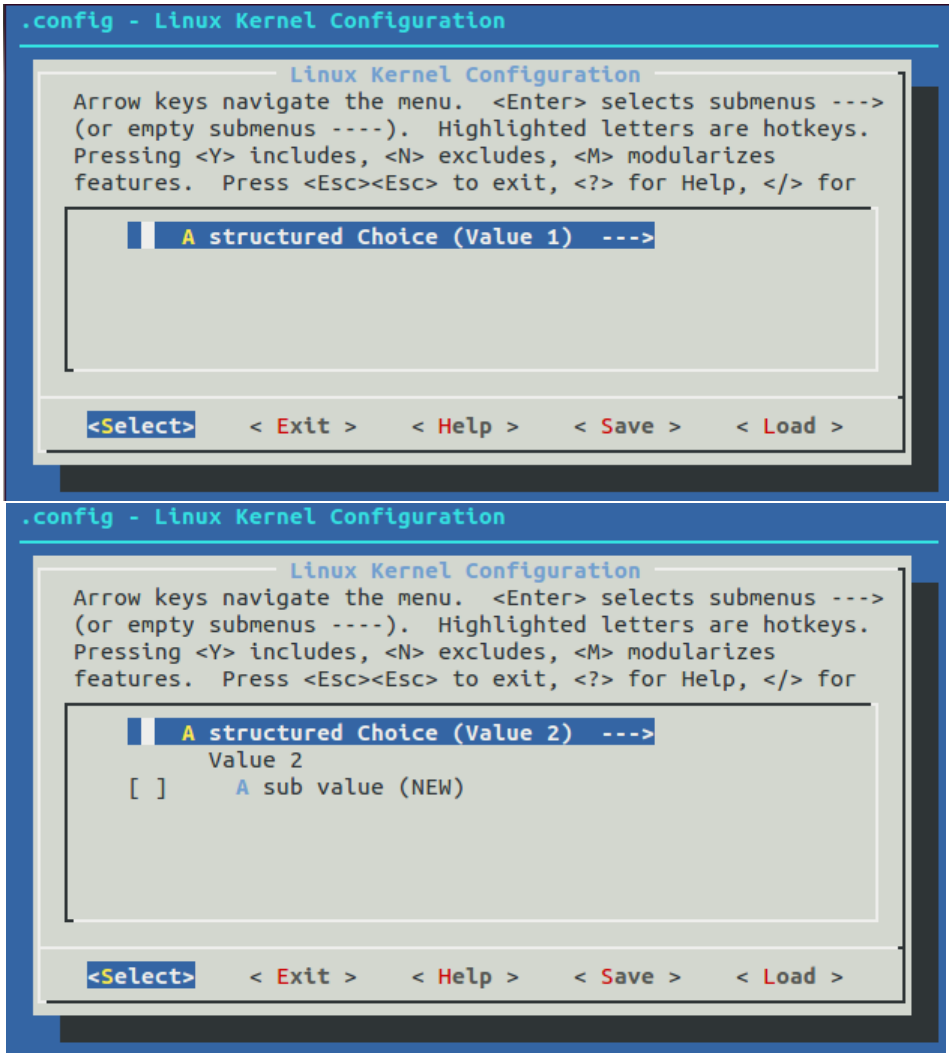

Figure 2.25: Menuconfig execution of Figure 2.24. Selection of “Value 2” facilitates the selection of “A sub value” (2nd screen shot).

Consequence

Exactly one of VAL_1 or VAL_2 will be selected inside the resulting `.config` file. The `.config` may also contain the selection of SUB_VALUE if “A sub value” was selected together with “Value 2” during the configuration process.

Resulting .config file

```
#
# Automatically generated file; DO NOT EDIT.
# Linux Kernel Configuration
#
# CONFIG_VAL_1 is not set
CONFIG_VAL_2=y
CONFIG_SUB_VAL=y
```

Figure 2.26: Saved `.config` file of the example from Figure 2.25 ("A sub value" was selected).

2.4.5 Recursive Dependency inside a Choice

Description

In Section 2.4.4, we already described how `depends on` constraints can be used inside a choice to create hierarchical dependencies inside a choice, which will also enable the multiple selection of nested config options inside the same choice. For such hierarchies it is important, that all nested config options are written directly below the containing config option. If a not nested config option is written between the containing and the nested config option, the whole hierarchy will be broken.

Minimal example

```
1  choice
2      bool "A Structured Choice"
3
4      config VAL_1
5          bool "Value 1"
6
7      config SUB_VAL_1
8          bool "Visible Sub Value 1"
9          depends on VAL_1
10
11     config SUB_VAL_2
12         bool "Visible Sub Value 2"
13         depends on VAL_1
14
15     config VAL_2
16         bool "Value 2"
17
18     config DEAD_SUB_VAL
19         bool "Dead Sub Value"
20         depends on VAL_1
21
22 endchoice
```

Figure 2.27: KConfig code for creating a structured choice with a dead config option ("Dead Sub Value" will never be selectable).

Behaviour of KConfig

Figure 2.28 demonstrates how KConfig handles the minimal example from Figure 2.27 on the previous page. Because there was a top level config option inserted in line 15 between the nested config options of "Value 1", "Dead Sub Value" is not longer part of the hierarchy. Thus, "Dead Sub Value" will not be selectable as it is also part of a choice of type `bool`. Moreover, KConfig is detecting a "recursive dependency" (cf. Figure 2.29 on the following page) and will remove the complete hierarchy from the choice as shown in the upper image from Figure 2.28.

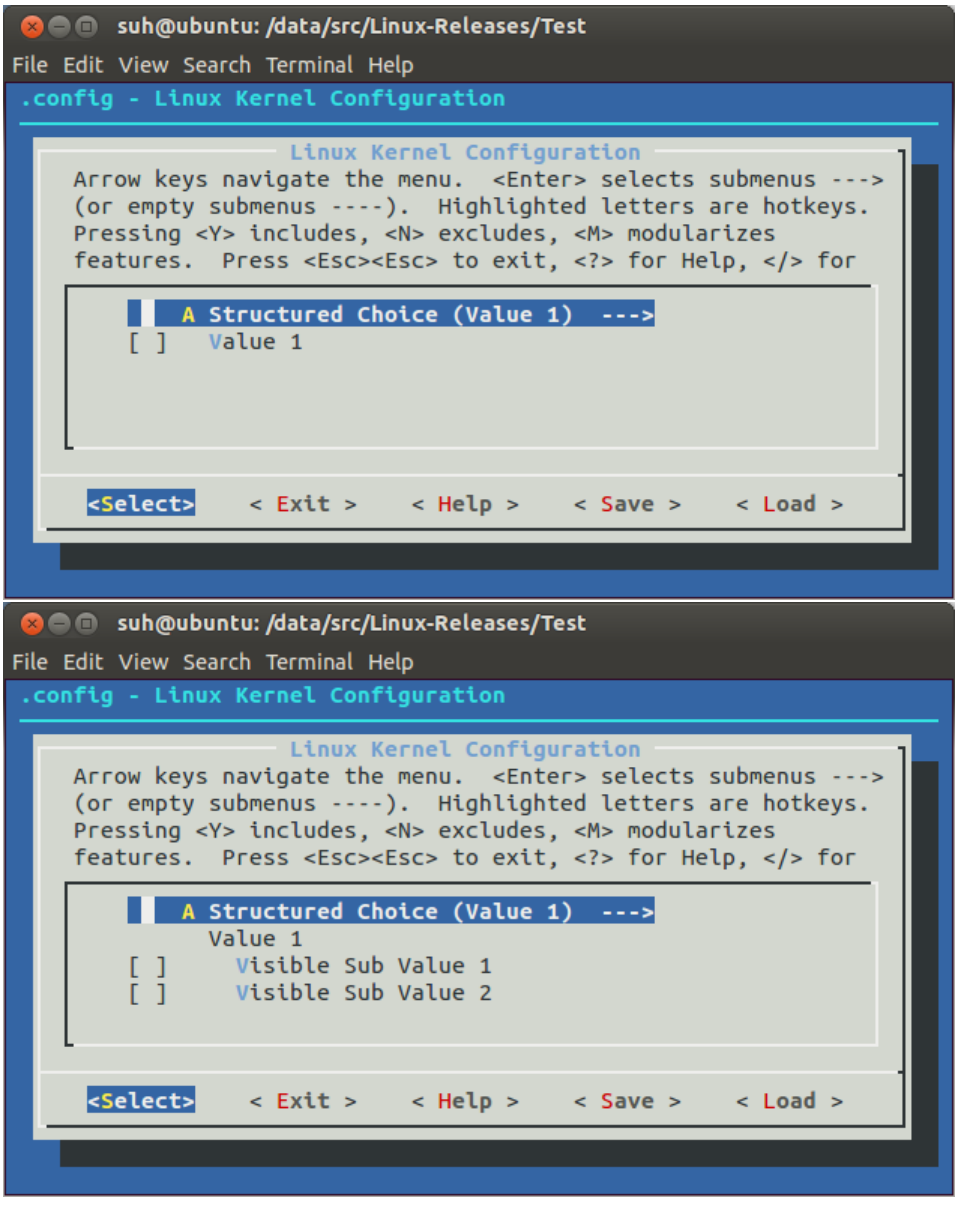

Figure 2.28: Menuconfig execution of Figure 2.27 (first screen shot). Only the removal of "Value 2" from the KConfig model in Figure 2.27 (lines 15 – 16) facilitates the selection of nested elements below "Value 1" (2nd screen shot).

The hierarchy will only be displayed correctly after removing the dead config option from the KConfig model (Figure 2.27 on page 38, lines 18 – 20). The resulting menu is shown in the lower image of Figure 2.28 on the previous page.

```
1 suh@ubuntu:/data/src/Linux-Releases/Test$ make menuconfig
2 make[1]: Entering directory '/data/src/Linux-Releases/Test'
3 make[1]: Leaving directory '/data/src/Linux-Releases/Test'
4 make[1]: Entering directory '/data/src/Linux-Releases/Test'
5 scripts/kconfig/mconf Kconfig
6 Kconfig:4:error: recursive dependency detected!
7 Kconfig:4:  choice <choice> contains symbol DEAD_SUB_VAL
8 Kconfig:21: symbol DEAD_SUB_VAL depends on VAL_1
9 Kconfig:7:  symbol VAL_1 is part of choice <choice>
```

Figure 2.29: Log file for compiling the model of Figure 2.27

2.4.6 Empty Choices

Description

Choices as well as their included config options can be made dependant of other config options. These `depends on` constraints need not necessarily to be same. This allows the creation of empty boolean choices. Although the KConfig language specification specifies the selection of exactly one variable, KConfig is able to handle such situations and will not force the selection of a non existing config option.

Real world example from Linux

- **Kernel version:** 2.6.33.3
- **File:** `drivers/mmc/host/Kconfig`
- **Line(s):** 194 – 218
- **Location inside the menu of menuconfig:**
  `Prompt: Atmel SD/MMC Driver`
  `-> Device Drivers`
  `-> MMC/SD/SDIO card support (MMC [=y])`

Minimal example

```
1  choice
2      bool "An Empty Choice"
3
4      config VAL_1
5          bool "Value 1"
6          depends on NOT_EXISTING
7      config VAL_2
8          bool "Value 2"
9          depends on NOT_EXISTING
10 endchoice
```

Figure 2.30: KConfig code for creating an empty choice (no config item can be choosen).

Behaviour of KConfig

The choice "An Empty Choice" is still part of the configuration menu, but the configuration front-ends do not allow the selection of one of its nested config options. Figure 2.31 demonstrates how the empty choice is displayed in menuconfig. The absence of brackets behind its name indicates that there is no further selection of nested config options possible.

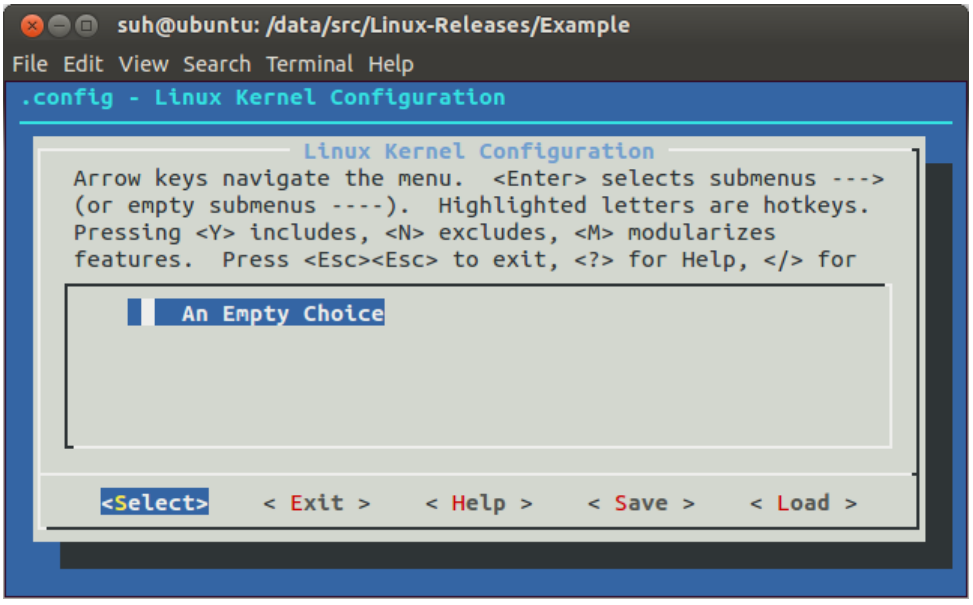

Figure 2.31: Menuconfig execution of Figure 2.30 (no value can be selected).

Consequence

The configuration of the model presented in Figure 2.30 will always produce an empty `.config` file, since the empty choice does now allow the selection of a nested config item. The complete `.config` file is presented in Figure 2.32.

Resulting .config file

```
#
# Automatically generated file; DO NOT EDIT.
# Linux Kernel Configuration
#
```

Figure 2.32: Saved `.config` file of the example from Figure 2.31 (no value could be selected).

2.4.7 Choices Without a Prompt

Description

The `prompt` attribute is also for choices only optional or can be conditional. A choice which is not visible to the user, will also not part of the configuration even if the choice should be present (not `optional` and all `depends on` constraints are fulfilled) and a `default` value was specified.

Real world example from Linux

- **Kernel version:** 3.19
- **File:** `arch/x86/Kconfig`
- **Line(s):** 1204 – 1236
- **Location inside the menu of menuconfig:**
  ```
  Prompt: Memory split
   -> Processor type and features
  Depends on: X86_32 [=y]
  Selected by: X86_32 [=y] && EXPERT [=y] && m
  ```

Minimal example

```
choice
    bool
    default BOOL_VAL1

    config BOOL_VAL1
        bool "1nd Boolean Value"

    config BOOL_VAL2
        bool "2nd Boolean Value"
endchoice
```

Figure 2.33: KConfig code of an invisible choice.

Behaviour of KConfig

A (mandatory) choice with no prompt will not be displayed inside the configuration front-ends. The `default` value has no impact as it will be ignored.

Consequence

Figure 2.34 and Figure 2.35 demonstrate how menuconfig handle the default value of the mandatory choice from Figure 2.33 on the preceding page. For Figure 2.35, we added a prompt to Figure 2.33 in line 2; everything else remained unchanged. Both times, we did not changed anything inside menuconfig before we saved the configuration.

Resulting .config file

```
1 #
2 # Automatically generated file; DO NOT EDIT.
3 # Linux Kernel Configuration
4 #
```

Figure 2.34: Saved `.config` file of the example from Figure 2.33 (no value could be selected).

In Figure 2.34 was nothing saved. In Figure 2.35, the default value was used to save the configuration of the nested config options.

```
1 #
2 # Automatically generated file; DO NOT EDIT.
3 # Linux Kernel Configuration
4 #
5 CONFIG_BOOL_VAL1=y
6 # CONFIG_BOOL_VAL2 is not set
```

Figure 2.35: Saved `.config` file of the example from Figure 2.33 (no value was selected). The model from Figure 2.33 was extended with a prompt in line 2.

2.4.8 Recursive Dependency inside a Choice via an `if`

Description

In Section 2.4.5, we already described how `depends on` constraints can create a recursive dependency inside a choice, which will remove the complete nested hierarchy from the KConfig model. Further, we showed in Section 2.3.8 that an `if` statement can also break a hierarchy, even if the constraints are modelled correctly.

Minimal example

```
config EXT_VAR
    bool "Not nested Variable"

choice
    bool "Choice"
    config VAR_1
        bool "Variable 1"

    config VAR_2
        bool "Variable 2"

        config NESTED_VAR_1
            bool "Nested Variable 1"
            depends on VAR_2
        if EXT_VAR
            config NESTED_VAR_2
                bool "Dead Sub Value"
                depends on VAR_2
        endif
        config NESTED_VAR_3
            bool "Nested Variable 3"
            depends on VAR_2
endchoice
```

Figure 2.36: KConfig code for creating a structured choice with a dead config option via an `if` ("Dead Sub Value" will never be selectable).

Behaviour of KConfig

Figure 2.37 on the next page demonstrates how KConfig handles the minimal example from Figure 2.36. Although `NESTED_VAR_2` is dependant of `VAR_2` and written correctly below this config option, the whole hierarchy is removed, since `VAR_2` is not also used inside the `if` statement. Further, menuconfig also displays a warning while creating the menu similar to the warning already shown in Figure 2.29 on page 40.

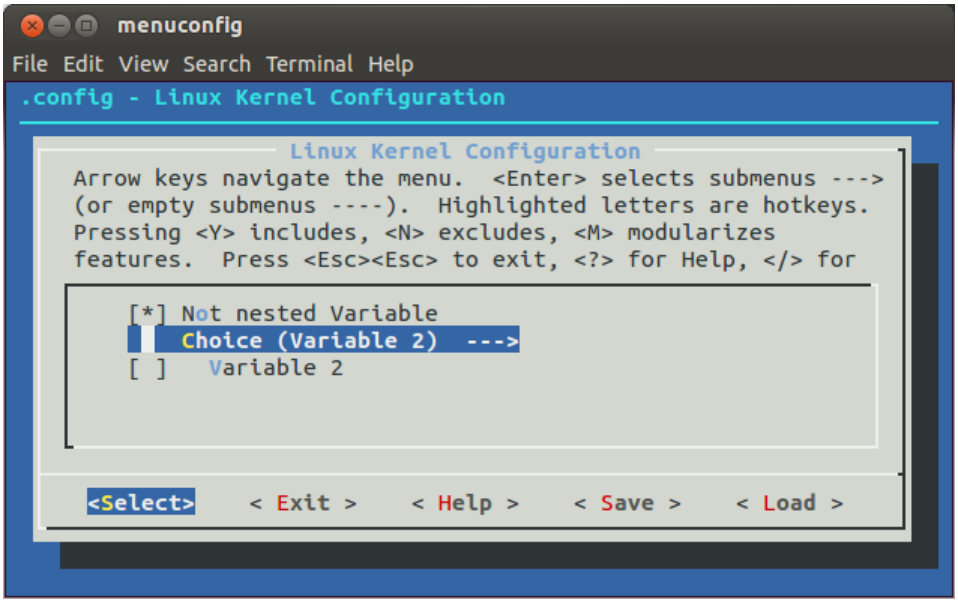

Figure 2.37: Menuconfig execution of Figure 2.36. The complete hierarchy below "Variable 2" is missing.

The hierarchy will only be displayed correctly after involving `VAR_2` inside the `if` statement in line 15.

2.4.9 Multiple Attributes inside a Config Option

Description

Config options may nest multiple attributes of the same type, e.g. multiple `default` value definitions. Inside the KConfig model of Linux, multiple conditional attributes are used to specify different values under different conditions. In cases were multiple interacting attributes of the same type become active, only the first one is used.

Real world example from Linux

- **Kernel version:** 3.19
- **File:** `arch/x86/Kconfig`
- **Line(s):** 1334 – 1343
- **Location inside the menu of menuconfig:**

```
Prompt: Maximum NUMA Nodes (as a power of 2)
 -> Processor type and features
Depends on: NEED_MULTIPLE_NODES [=y]
```

Minimal example

```
1 config VAR
2     string "A String Variable"
3     default "Value 1"
4     default "Value 2"
```

Figure 2.38: KConfig code of an invisible choice.

Behaviour of KConfig

In Figure 2.38 multiple default values are specified for the same config option. Menuconfig will take the first default value (line 3) during the configuration as shown in Figure 2.39.

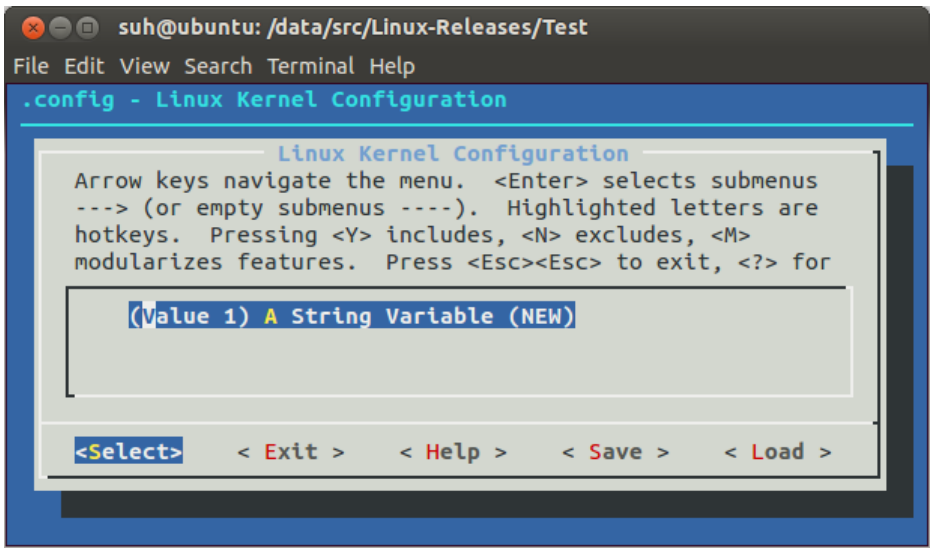

Figure 2.39: Menuconfig execution of Figure 2.38 (the first `default` was set as default value).

Consequence

If the default value was not changed, e.g. because the config option was also not visible to the user, the first default value will automatically written into the `.config` file (cf. Figure 2.40).

Resulting .config file

```
1 #
2 # Automatically generated file; DO NOT EDIT.
3 # Linux Kernel Configuration
4 #
5 CONFIG_VAR="Value 1"
```

Figure 2.40: Saved `.config` file of the example from Figure 2.39.

Chapter 3

Tool Analysis

In this section, we show how the existing tools and approaches handle the findings from Chapter 2. In Section 3.1, we give a brief introduction into the analysed tools. In Section 3.2 to Section 3.4, we analyse how the tools handle some important concepts of Section 2.1, which differ fundamentally to boolean logic. In sections 3.5 to 3.15, we show how the tools handle the observations of Chapter 2. Finally, in Section 3.16 we summarize the findings of our analysis.

3.1 Analysed Tools

We analysed three tools: Undertaker, KConfig Reader, and LVAT. Further we analysed KConfig Reader in two different versions. All but LVAT are capable of translating KConfig into three logical models. LVAT offers mechanisms for printing statistical information about a given KConfig model like, hierarchies, permanent visible variables, visibility conditions, or different kinds of dependency analyses (OR, XOR, Mutex groups). Here, we analysed LVAT's capabilities of translating a KConfig models into DIMACS–format. In Section 3.1.1, we introduce Undertaker and its tools responsible for the generation of the logical models. Section 3.1.2 introduces both versions KConfig Reader, which use parts of Undertaker for an alternative translation. Section 3.1.3 explains how LVAT stores the variability information of KConfig models in DIMACS–format.

We will also briefly describe the generated output files of the analysis tools based on the KConfig model from Figure 3.1 on the next page. This model contains a menu with 3 nested config options and a choice with two nested config options. Inside the menu, a constraint is used to model a hierarchy. Through the constraint in line 10, `STR_VAR` will be indented below `MODULES`. `TRI_VAR` is a config option of type tristate, which can only be selected to `m`, if `MODULES` was set to `y` before, because it contains the `option modules` attribute (cf. line 5). Selecting `VAL2` inside the choice will force the permanent selection (`y`) of `TRI_VAR`, because of the modeled select constraint in line 24.

```
menu "A Menu"
    config MODULES
        bool "Enable third state"
        default y
        option modules

    config STR_VAR
        string
        prompt "A string variable"
        depends on MODULES

    config TRI_VAR
        tristate "A tristate variable"
endmenu

choice
    bool "Like an enumeration"

    config VAL1
        bool "Value 1"

    config VAL2
        bool "Value 2"
        select TRI_VAR
endchoice
```

Figure 3.1: Small KConfig example for explaining the translated logical models.

3.1.1 Undertaker

Undertaker [Und] is a tool set developed to check the structure of preprocessor directives of the Linux Kernel against its configuration model to find code blocks, which are permanently (de-)selected in all configurations. This analysis is done in two steps. First, all KConfig files are translated into logical models. In the second step, the preprocessor directives are also translated into logical models and passed directly together with the logical translations of KConfig to a (SAT) solver to find code, which is not configurable.

Undertaker embeds two alternative tool chains for translating Kconfig into boolean formula:

- **dumpconf** is a modified version of menuconfig. This tool translates the KConfig files first into an intermediate format. This intermediate information is saved as RSF files (*.rsf), which are Tab Separated Value files (TSV) with a fixed set of used elements to describe the information of the KConfig files in a more structured way. The RSF files mainly contain the same logical information as the KConfig files, but dumpconf already adds some extra information for downstream analysis tools, e.g. `depends on` constraints between nested config options and the containing choices.

```
1  Choice          CHOICE_1        required        boolean
2  ChoiceItem      VAL1            CHOICE_1
3  ChoiceItem      VAL2            CHOICE_1
4  Default         MODULES         "y"             "y"
5  Depends         STR_VAR         "MODULES"
6  Depends         VAL1            "CHOICE_1"
7  Depends         VAL2            "CHOICE_1"
8  HasPrompts      MODULES         1
9  HasPrompts      STR_VAR         1
10 HasPrompts      TRI_VAR         1
11 HasPrompts      VAL1            1
12 HasPrompts      VAL2            1
13 Item            MODULES         boolean
14 ItemSelects     VAL2            "TRI_VAR"       "CHOICE_1"
15 Item            STR_VAR         string
16 Item            TRI_VAR         tristate
17 Item            VAL1            boolean
18 Item            VAL2            boolean
```

Figure 3.2: dumpconf's translation of Figure 3.1 on the preceding page.

An example for such a RSF file is given in Figure 3.2. Each row can be interpreted as a tuple:

- `(Item, <name>, <type>)` declares a config option (cf. lines 13 and 15 – 18).
- `(HasPrompts, <name>, <No.>)` indicates how many (conditional) prompts are associated to the config option (cf. lines 8 – 12).
- `(Choice, CHOICE_<No.>, required|optional, boolean|tristate)` declares a choice (cf. line 1).
- `(ChoiceItem, <name of an Item>, <name of a Choice>)` denotes that a config option is nested inside a choice (cf. lines 2 and 3).
- `(Default, <name of an Item or Choice>, <default value>, <expression>)` defines a (conditional) default value (cf. line 4).
- `(Depends, <name of an Item or Choice>, <expression>)` defines a dependency between a config option or choice and other elements of the KConfig model (cf. lines 5 – 7).
- `(Has Prompts, <name of an Item>, <No.>)` shows how many (conditional) prompts are defined for the related config option (cf. lines 8 – 12).
- `(ItemSelects, <name of holding Item>, "<name of target Item>", <expression>)` defines a (conditional) `select` statement between two config options (cf. line 14).
- `if`s are translated into `Depends` elements.

Even expressions based on Strings and Numbers are translated. But string values are not surrounded by quotes, thus, it is not possible to clearly determine start and ending of string values inside complex constraints.

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

An example for such a Model file is given in Figure 3.3. The first lines are used to declare the config options and choices of the KConfig model (cf. lines 1 – 7). These lines may contain a quoted expression, which must be fulfilled if the element was selected. `Tristate` elements are translated into two elements (`CONFIG_<NAME>` and `CONFIG_<NAME>_MODULE`), with the following semantics:

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

The statement in line 8 lists elements which are permanently set to `y` (`true`). Constraints based on comparisons of String or Numerical config options are not considered during the translation.

```
1  CONFIG_CHOICE_1 "((CONFIG_VAL1 && !CONFIG_VAL2) ||
       (!CONFIG_VAL1 && CONFIG_VAL2))"
2  CONFIG_MODULES
3  CONFIG_STR_VAR "CONFIG_MODULES"
4  CONFIG_TRI_VAR "!CONFIG_TRI_VAR_MODULE"
5  CONFIG_TRI_VAR_MODULE "!CONFIG_TRI_VAR && CONFIG_MODULES"
6  CONFIG_VAL1 "CONFIG_CHOICE_1"
7  CONFIG_VAL2 "CONFIG_CHOICE_1"
8  UNDERTAKER_SET ALWAYS_ON "CONFIG_CHOICE_1"
9  UNDERTAKER_SET SCHEMA_VERSION 1.1
10 CONFIG_X86 ""
11 CONFIG_n
12 CONFIG_y
13 CONFIG_m
```

Figure 3.3: rsf2model's translation of Figure 3.1 on page 48.

- **Satyr** is an alternative to **dumpconf** + **rsf2model**, which was introduced in Undertaker 1.5 [Und]. This tool translates the KConfig files directly into DIMACS–format [Sat93].

  An example for the output produced by Satyr is given in Figure 3.4.

  - The first lines are comments, presenting a mapping of the KConfig elements to DIMACS variables, which are only numbers.
  - The first line below the comments declares how many boolean variables and how many disjunction constraints are used inside the DIMACS file (cf. line 27: `p cnf <number of variables> <number of constraints>`).

– The rest of the file contains the disjunction constraints.
– Constraints using such non–Boolean–/Tristate–elements are translated into separate variables in the form of `CONFIG___FREE__(NE | EQ)<Number>`. These variables are used correctly to translate constraints, but Satyr does not model a dependency between this assignment variable and the related String/Numerical element.
– Tristate elements are translated in the same way as in Model–files.

```
c File Format Version: 2.0
c Generated by satyr
c Type info:
c c sym <symbolname> <typeid>
c with <typeid> being an integer out of:
c enum {S_BOOLEAN=1, S_TRISTATE=2, S_INT=3, S_HEX=4, S_STRING=5, S_OTHER=6}
c variable names:
c c var <variablename> <cnfvar>
c meta_value ALWAYS_ON CONFIG_CHOICE_0
c sym CHOICE_0 1
c sym MODULES 1
c sym STR_VAR 5
c sym TRI_VAR 2
c sym UNAME_RELEASE 5
c sym VAL1 1
c sym VAL2 1
c sym ____MODULES_MAGIC_INTERNAL_VAR____ 1
c var CONFIG_CHOICE_0 1
c var CONFIG_MODULES 9
c var CONFIG_STR_VAR 15
c var CONFIG_TRI_VAR 17
c var CONFIG_TRI_VAR_MODULE 18
c var CONFIG_UNAME_RELEASE 14
c var CONFIG_VAL1 2
c var CONFIG_VAL2 3
c var CONFIG_____MODULES_MAGIC_INTERNAL_VAR____ 8
p cnf 21 43
1 0
-4 2 0
-4 -3 0
4 -2 3 0
-5 -2 0
-5 3 0
5 2 -3 0
6 -4 0
6 -5 0
-6 4 5 0
7 1 0
7 -6 0
-7 -1 6 0
```

Figure 3.4: Satyr's translation (excerpt) of Figure 3.1 on page 48.

For the analysis we used Undertaker in version 1.6.1. We analysed all three tools in independent from each other, since they can be executed independently of the whole

Undertaker tool set, which facilitates the usage in other projects outside of the context of Undertaker.

3.1.2 KConfig Reader + KConfig Reader (XML)

KConfig Reader [kcob] was designed for converting the KConfig files into boolean formulas for reasoning. In this analysis we tested two versions KConfig Reader:

- In the following, we use **KConfig Reader** to refer to the latest version of KConfig Reader, which produces RSF files as intermediate format. This version uses a patched version of Undertaker's **dumpconf** for creating RSF files, which must be downloaded from a separate repository[1]. These files are than used to create boolean formulas. These boolean formulas are stored in Model files (*.model). However, KConfig Reader's Model files are different to Undertaker's Model files. KConfig Reader optionally offers the possibility to create CNF formula, which is also saved in DIMACS–format [Sat93].

  For the analysis, we used the latest version of the patched dumpconf tool, which was from 30.06.2014. KConfig Reader was a version from 12.09.2014. We analysed the correctness of generated RSF, Model, and DIMACS files.
- In the following, we use **KConfig Reader (XML)** to refer explicitly to a younger version of KConfig Reader. This version relies on a reimplemented version of the patched dumpconf tool, which is already part of KConfig Reader (XML). Contrary to the older implementation, this version uses XML as an intermediate format instead of the common RSF structure. However, the XML files are still named as *.rsf files. The structure of Model and DIMACS files has not been changed.

  For the analysis, we used KConfig Reader (XML) from 03.06.2015. We analysed the correctness of generated Model and DIMACS files. We did not checked the correctness of the generated XML files, as they are only an intermediate format, used by KConfig Reader (XML) only.

Both tools are able to create all three model types with one execution step. Hence, we analysed the produced files without identifying potential sub components:

- **RSF**–files produced by older versions of KConfig Reader contain additional information compared to the RSF–files produced by **dumpconf**. An example is given in Figure 3.5 on the next page:
  - It seems to keep the ordering of the config options and choices inside the KConfig files. Each time, the elements of the translated RSF files had the same ordering as inside the source KConfig files.
  - It does not only count how many `prompt`s are related to an item `(HasPrompts, <name>, <No.>)`, it also shows under which situation a `prompt` becomes active `(Prompt, <name>, <expression>)`.
  - These RSF files contain further information, which was not part of this analysis, like the comments (cf. lines 12, 16, 22, 29, and 30), and `range`s for numer-

[1] https://github.com/ckaestne/undertaker

ical config options (`Range, <name>, "[<lower bound> <upper bound>]", <expression>`).

```
1  Item          MODULES      boolean
2  Prompt        MODULES      "y"
3  HasPrompts    MODULES      1
4  Default       MODULES      "'y'"          "y"
5  Item          STR_VAR      string
6  Depends       STR_VAR      "MODULES"
7  Prompt        STR_VAR      "MODULES"
8  HasPrompts    STR_VAR      1
9  Item          TRI_VAR      tristate
10 Prompt        TRI_VAR      "y"
11 HasPrompts    TRI_VAR      1
12 #startchoice
13 Choice        CHOICE_1     required       boolean
14 Prompt        CHOICE_1     "y"
15 HasPrompts    CHOICE_1     1
16 #choice value
17 ChoiceItem    VAL1         CHOICE_1
18 Item          VAL1         boolean
19 Depends       VAL1         "CHOICE_1"
20 Prompt        VAL1         "CHOICE_1"
21 HasPrompts    VAL1         1
22 #choice value
23 ChoiceItem    VAL2         CHOICE_1
24 Item          VAL2         boolean
25 Depends       VAL2         "CHOICE_1"
26 Prompt        VAL2         "CHOICE_1"
27 HasPrompts    VAL2         1
28 ItemSelects   VAL2         "TRI_VAR"      "CHOICE_1"
29 #choice value
30 #endchoice
```

Figure 3.5: KConfig Reader's RSF translation of Figure 3.1 on page 48.

- **Model** files produced by KConfig Reader and KConfig Reader (XML) can not be compared with Model files produced by Undertaker. An example is given in Figure 3.6 on the next page. These Model files contain Boolean formula, which are not necessarily in CNF form:
  - Each variable is surrounded by `!def(<variable>)` or `def(<variable>)` denoting whether the variable is deselected (= `false`) or selected (= `true`).
  - Tristate config options / choices are translated into two variables: `<variable>` and `<variable>_MODULE`.
    - `<variable> = false` and `<variable>_MODULE = false` denote that the related element inside the KConfig model was permanently deselected (`n`).

    - * `<variable> = true` and `<variable>_MODULE = false` denote that the related element inside the KConfig model was permanently selected (`y`).
    - * `<variable> = false` and `<variable>_MODULE = true` denote that the related element inside the KConfig model was configured as a module (`m`).
    - * `<variable> = true` and `<variable>_MODULE = true` is not allowed.
  - String and Numerical elements are translated in several variables in form of `def(<NAME>=<value>)`. Kconfig Reader is creating such a construct for each value comparison found in KConfig constraints and creates constraints to avoid multiple value selections for the same config option. Further variables are created as follows:
    - * `def(<NAME>=)` — Config option `<NAME>` was assigned to an emtpy String.
    - * `def(<NAME>=n)` — Config option `<NAME>` is not part of the configuration, i.e. a dependency was not fulfilled.
    - * `def(<NAME>=nonempty)` — A value was assigned, which was not covered by one of the `def(<NAME>=<value>)` variables, i.e. a value which was not used inside a constraint.

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

```
#item CHOICE_1
def(CHOICE_1)
#item MODULES
#item STR_VAR
(!def(MODULES)|!def(STR_VAR=n))
(def(STR_VAR=)|def(STR_VAR=nonempty)|def(STR_VAR=n))
(def(MODULES)|(!def(STR_VAR=)&!def(STR_VAR=nonempty)))
(def(MODULES)|(!def(STR_VAR=)&!def(STR_VAR=nonempty)))
(!def(STR_VAR=)|!def(STR_VAR=nonempty))
(!def(STR_VAR=)|!def(STR_VAR=n))
(!def(STR_VAR=nonempty)|!def(STR_VAR=n))
#item TRI_VAR
(!def(TRI_VAR_MODULE)|def(MODULES))
(!def(TRI_VAR)|!def(TRI_VAR_MODULE))
(!def(VAL2)|!def(CHOICE_1)|def(TRI_VAR)|def(TRI_VAR_MODULE))
(!def(VAL2)|!def(CHOICE_1)|def(TRI_VAR))
#item VAL1
#item VAL2
#choice CHOICE_1
(def(VAL2)|def(VAL1)|!def(CHOICE_1))
(!def(VAL2)|def(CHOICE_1))
(!def(VAL1)|def(CHOICE_1))
(!def(VAL2)|!def(VAL1))
```

Figure 3.6: KConfig Reader's Model translation of Figure 3.1 on page 48.

- KConfig Reader and KConfig Reader (XML) are also able to translate the KConfig model into **CNF** formula and save this as a DIMACS–file. An example is given in Figure 3.7:
  - Also this translation contains a variable mapping at the beginning of the file, like the translation of Satyr.
    - Tristate elements are translated into two variables: `<variable>` and `<variable>_MODULE`. These variables have the same meaning as inside the Model translation of KConfig Reader and KConfig Reader (XML).
    - Non Boolean/Tristate elements are translated as in Model–files.
  - The first line below the comments declares how many boolean variables and how many disjunction constraints are used inside the DIMACS file (cf. line 5: `p cnf <number of variables> <number of constraints>`).
  - The rest of the file contains the disjunction constraints.

```
1  c 1 TRI_VAR_MODULE
2  c 5 CHOICE_1
3  c 2 MODULES
4  c 3 TRI_VAR
5  c 9 VAL1
6  c 7 STR_VAR=
7  c 4 VAL2
8  c 8 STR_VAR=nonempty
9  c 6 STR_VAR=n
10 p cnf 9 18
11 -1 2 0
12 -3 -1 0
13 -4 -5 3 1 0
14 -4 -5 3 0
15 -2 -6 0
16 7 8 6 0
17 -8 2 0
18 -7 2 0
19 -8 2 0
20 -7 2 0
21 -7 -8 0
22 -7 -6 0
23 -8 -6 0
24 5 0
25 4 9 -5 0
26 -4 5 0
27 -9 5 0
28 -4 -9 0
```

Figure 3.7: KConfig Reader's CNF translation of Figure 3.1 on page 48.

3.1.3 LVAT

The Linux Variability Analysis Tools (LVAT) translates KConfig models in **CNF** formula and saves this as a DIMACS–file, but LVAT is not able to translate the KConfig files

directly into DIMACS format. First the models must be translated into an intermediate format, which should simplify parsing. The developers offer two tools for generating the intermediate format [Kcoa]:

exconf translates KConfig into a text format. This extractor is marked as compatible with KConfig models prior to Linux v3.0. An example is given in Listing 3.8, where we translated the KConfig model from Listing 3.1, the referenced KConfig model in drivers was empty.

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

In our analysis, we used protoconf for translating KConfig into the needed intermediate format. Further, we used vm2bool for generating the DIMACS–files. This tool was also developed by the developers of LVAT to simplify the generation of DIMACS–files.

```
menu "A Menu" {
 config MODULES boolean {
  prompt "Enable third state" if []
  default [y] if []
  config STR_VAR string {
   prompt "A string variable" if [MODULES]
   depends on [MODULES]
   inherited [MODULES]
  }
 }
 config TRI_VAR tristate {
  prompt "A tristate variable" if []
 }
}
choice boolean {
 prompt "Like an enumeration" if []
 config VAL1 boolean {
  prompt "Value 1" if [<choice>]
  inherited [<choice>]
 }
 config VAL2 boolean {
  prompt "Value 2" if [<choice>]
  select TRI_VAR if [<choice>]
  inherited [<choice>]
 }
}
if [STR_VAR="Hello World"] {
}
```

Figure 3.8: Intermediate format of LVAT (translated only the menu from Listing 3.1).

KConfig elements are translated into two variables: `<variable>` and `<variable>_m`:

- For Boolean elements only two states are valid:
  - `<variable>` and `<variable>_m` both being false means that the element is permanently deselected (`n`).
  - `<variable>` and `<variable>_m` both being true means that the element is permanently selected (`y`).

  For instance, this behaviour is achieved for the variable `MODULES` via lines 25 and 26 in Figure 3.9 on the next page.
- For Tristate elements three states are valid:
  - `<variable>` and `<variable>_m` both being false means that the element is permanently deselected (`n`).
  - `<variable>` and `<variable>_m` both being true means that the element is permanently selected (`y`).
  - `<variable>` being true and `<variable>_m` being false means that the element is selected as a module (`m`). This state is only valid if the third state of Tristate elements is enabled, as described in Section 2.1 on page 14.

  For example, this behaviour is achieved for the variable `TRI_VAR` via lines 44 and 45 in Figure 3.9 on the next page.
- Non Boolean/Tristate elements are translated exactly like Booleans. Literal comparisons are not considered, thus almost all translations of constraints with non Boolean/Tristate elements are wrong.

LVAT also uses auxiliary variables, which are marked with a `$` in the variable mapping at the beginning of the DIMACS file [Pro].

```
1  c 1$ _X5
2  c 2$ _X1_m
3  c 3$ _X5_m
4  c 4 TRI_VAR_m
5  c 5$ _X7_m
6  c 6$ _X2
7  c 7$ _X1
8  c 8$ _X6
9  c 9 MODULES_m
10 c 10 MODULES
11 c 11$ _X3
12 c 12$ _X4_m
13 c 13$ _X7
14 c 14$ _X3_m
15 c 15$ _X4
16 c 16$ _X6_m
17 c 17 TRI_VAR
18 c 18$ _X2_m
19 p cnf 18 34
20 -7 0
21 -2 0
22 -7 10 9 0
23 -2 10 9 0
24 -7 -2 -10 9 0
25 -10 9 0
26 10 -9 0
27 -6 0
28 -18 0
29 -6 10 0
30 -6 9 0
31 -11 0
32 -14 0
33 -11 -10 7 0
34 -11 -7 10 0
35 -11 -9 2 0
36 -11 -2 9 0
37 -15 0
38 -12 0
39 -1 0
40 -3 0
41 -1 17 4 0
42 -3 17 4 0
43 -1 -3 -17 4 0
44 17 -4 0
45 -17 4 10 0
46 -8 0
47 -16 0
48 -8 -17 1 0
49 -8 -1 17 0
50 -8 -4 3 0
51 -8 -3 4 0
52 -13 0
53 -5 0
```

Figure 3.9: LVAT's CNF translation of Figure 3.1 on page 48 (translated on lines 2 – 5 and 12 – 13).

3.1.4 Tool Summary

Table 3.1 on the following page summarizes the capabilities of the analysed translators. It has to be considered that rsf2model's Model–files are completely different to KConfig Reader's Model–files.

Translator		Output	Strings & Numbers	Note	Last Commit	Repository
Undertaker	**dumpconf**	RSF	Partially supported	Modified version of menuconfig, reorders config options	01.06.2015	`https://vamos.informatik.uni-erlangen.de/trac/undertaker`
	rsf2model	Model	Not supported	Uses RSF–files as input		
	Satyr	DIMACS	Partially supported			
KConfig Reader		RSF, Model, DIMACS	✓	Testet version from 12.09.2014, which was the latest version using modified version of dumpconf	12.09.2014 (latest version using the old dumpconf implementation)	`https://github.com/ckaestne/kconfigreader`, `https://github.com/ckaestne/undertaker`
KConfig Reader (XML)		XML, Model, DIMACS	✓	Testet version from 03.06.2015, which was the latest version	03.06.2015	`https://github.com/ckaestne/kconfigreader`
Linux Variability Analysis Tools (LVAT)		DIMACS	Not supported		01.04.2013 (LVAT), 30.05.2015 (vm2bool)	`https://code.google.com/p/linux-variability-analysis-tools/`, `https://bitbucket.org/tberger/vm2bool`

Table 3.1: Summary of tool capabilities.

3.2 Handling Attribute `option modules`

In Section 2.1 on page 14, we explained that the third state of `tristate` config options can be controlled by a config option with the name `MODLUES`. However, the current version of KConfig facilitates renaming this config option. If the responsible config option is not named as `MODULES`, it must be attributed with `option modules` to control the third state of all `tristate` config options.

Figure 3.10 presents our KConfig model for testing the correct translation of the `option modules` attribute. This example still contains a config option with the name `MODLUES`, but the third state will be controlled by the attributed variable `UNNAMED_VAR`.

```
1 config MODULES
2     bool "A simple variable"
3
4 config UNNAMED_VAR
5     bool "Enable third state"
6     option modules
7
8 config TRISTATE_VAR
9     tristate "A tristate variable"
```

Figure 3.10: KConfig example for testing the translation the `option modules` attribute.

3.2.1 Undertaker

dumpconf

```
1 HasPrompts    MODULES       1
2 HasPrompts    TRISTATE_VAR  1
3 HasPrompts    UNNAMED_VAR   1
4 Item          MODULES       boolean
5 Item          TRISTATE_VAR  tristate
6 Item          UNNAMED_VAR   boolean
```

Figure 3.11: dumpconf's translation of Figure 3.10 (error).

Figure 3.11 shows how dumpconf translates the example from Figure 3.10. The translation does not contain any information which of the variables is used to control the third state of all `tristate` config options.

rsf2model

Line 3 in Figure 3.12 on the following page shows that the third state of `TRISTATE_VAR` is dependant of `MODULES`. This is not correct, since `UNNAMED_VAR` was used to control the third state.

```
1 CONFIG_MODULES
2 CONFIG_TRISTATE_VAR "!CONFIG_TRISTATE_VAR_MODULE"
3 CONFIG_TRISTATE_VAR_MODULE "!CONFIG_TRISTATE_VAR && CONFIG_MODULES"
4 CONFIG_UNNAMED_VAR
5 UNDERTAKER_SET SCHEMA_VERSION 1.1
6 CONFIG_X86 ""
7 CONFIG_n
8 CONFIG_y
9 CONFIG_m
```

Figure 3.12: rsf2model's translation of Figure 3.10 on page 60 (error in line 3).

Satyr

```
1  c File Format Version: 2.0
2  c Generated by satyr
3  c Type info:
4  c c sym <symbolname> <typeid>
5  c with <typeid> being an integer out of:
6  c enum {S_BOOLEAN=1, S_TRISTATE=2, S_INT=3, S_HEX=4, S_STRING=5, S_OTHER=6}
7  c variable names:
8  c c var <variablename> <cnfvar>
9  c sym MODULES 1
10 c sym TRISTATE_VAR 2
11 c sym UNAME_RELEASE 5
12 c sym UNNAMED_VAR 1
13 c sym ____MODULES_MAGIC_INTERNAL_VAR____ 1
14 c var CONFIG_MODULES 7
15 c var CONFIG_TRISTATE_VAR 8
16 c var CONFIG_TRISTATE_VAR_MODULE 9
17 c var CONFIG_UNAME_RELEASE 6
18 c var CONFIG_UNNAMED_VAR 2
19 c var CONFIG_____MODULES_MAGIC_INTERNAL_VAR____ 1
20 p cnf 10 14
21 3 1 0
22 3 -2 0
23 -3 -1 2 0
24 4 2 0
25 4 -1 0
26 -4 -2 1 0
27 -5 3 0
28 -5 4 0
29 5 -3 -4 0
30 5 0
31 -10 8 0
32 -10 9 0
33 10 -8 -9 0
34 -10 0
```

Figure 3.13: Satyr's CNF translation of Figure 3.10 on the preceding page (error).

Figure 3.13 shows that Satyr does not use the `option modules` attribute for creating the CNF formula. There is no connection between `TRISTATE_VAR` and `UNNAMED_VAR`. This allows to set `TRISTATE_VAR` to module (`TRISTATE_VAR_MODULE = true`) while setting `UNNAMED_VAR` to `false`.

3.2.2 KConfig Reader

RSF

```
1 Item          MODULES         boolean
2 Prompt        MODULES         "y"
3 HasPrompts    MODULES         1
4 Item          UNNAMED_VAR     boolean
5 Prompt        UNNAMED_VAR     "y"
6 HasPrompts    UNNAMED_VAR     1
7 Item          TRISTATE_VAR    tristate
8 Prompt        TRISTATE_VAR    "y"
9 HasPrompts    TRISTATE_VAR    1
```

Figure 3.14: KConfig Reader's RSF translation of Figure 3.10 on page 60 (error).

Figure 3.14 shows that also KConfig Reader does not take the `option modules` attribute into the RSF format.

Model

```
1 #item MODULES
2 #item TRISTATE_VAR
3 (!def(TRISTATE_VAR_MODULE)|def(MODULES))
4 (!def(TRISTATE_VAR)|!def(TRISTATE_VAR_MODULE))
5 #item UNNAMED_VAR
```

Figure 3.15: KConfig Reader's Model translation of Figure 3.10 on page 60 (error in line 3).

In Figure 3.15, `TRISTATE_VAR_MODULE` is dependant of `MODULES`. `UNNAMED_VAR` is not used inside the complete translated model.

CNF

```
1 c 1 TRISTATE_VAR_MODULE
2 c 2 MODULES
3 c 3 TRISTATE_VAR
4 p cnf 3 2
5 -1 2 0
6 -3 -1 0
```

Figure 3.16: KConfig Reader's CNF translation of Figure 3.10 on page 60 (error in line 5).

Figure 3.16 shows exactly the same formula as Figure 3.15 on the previous page, and thus, the same error.

3.2.3 KConfig Reader (XML)

Model & CNF

KConfig Reader (XML)'s Model and CNF translations of Figure 3.10 on page 60 are identical to KConfig Reader's translations (cf. Section 3.2.2) and therefore incorrect.

3.2.4 LVAT

In line 62 of Figure 3.17 on the next page, the third state of `TRISTATE_VAR` is allowed if `MODULES` is set to true. There is no reference to `UNNAMED_VAR`, thus the translation is not correct.

```
c 1$ _X9
c 2$ _X5
c 3$ _X1_m
c 4$ _X5_m
c 5 UNNAMED_VAR
c 6$ _X7_m
c 7$ _X2
c 8$ _X1
c 9 TRISTATE_VAR
c 10$ _X6
c 11 MODULES_m
c 12 MODULES
c 13$ _X3
c 14$ _X4_m
c 15 TRISTATE_VAR_m
c 16$ _X7
c 17 UNNAMED_VAR_m
c 18$ _X3_m
c 19$ _X4
c 20$ _X8_m
c 21$ _X6_m
c 22$ _X9_m
c 23$ _X8
c 24$ _X2_m
p cnf 24 45
-8 0
-3 0
-8 12 11 0
-3 12 11 0
-8 -3 -12 11 0
-12 11 0
12 -11 0
-7 0
-24 0
-7 -12 8 0
-7 -8 12 0
-7 -11 3 0
-7 -3 11 0
-13 0
-18 0
-19 0
-14 0
-19 5 17 0
-14 5 17 0
-19 -14 -5 17 0
-5 17 0
5 -17 0
-2 0
-4 0
-2 -5 19 0
-2 -19 5 0
-2 -17 14 0
-2 -14 17 0
-10 0
-21 0
-16 0
-6 0
-16 9 15 0
-6 9 15 0
-16 -6 -9 15 0
9 -15 0
-9 15 12 0
-23 0
-20 0
-23 -9 16 0
-23 -16 9 0
-23 -15 6 0
-23 -6 15 0
-1 0
-22 0
```

Figure 3.17: LVAT's CNF translation of Figure 3.10 on page 60 (error in line 62).

3.3 Constraint Precedence

In Section 2.1 on page 14, we explained that `select` constraints have a higher precedence than `if` or `depends on` constraints. That means that a config option can be selected even if some of its `depends on` constraints are not fulfilled.

We used the model of Figure 3.18 to analyse whether such contrary constraints are translated correctly. Figure 3.18 shows a model with 3 config options: `VAR1`, `VAR2`, and `CONST_FALSE`. `VAR2` is dependant from `CONST_FALSE`. However, if `VAR1` is selected it will also select `VAR2` and set it to `true`. Thus, `VAR1` and `VAR2` are always equal. `CONST_FALSE` will still not be part of the configuration file `.config`.

```
1  config CONST_FALSE
2      def_bool n
3
4  config VAR1
5      bool "Selecting variable"
6      select VAR2
7
8  config VAR2
9      bool "Selected variable"
10     depends on CONST_FALSE
```

Figure 3.18: KConfig example for testing the translation the constraint precedence.

3.3.1 Undertaker

dumpconf

```
1  Default      CONST_FALSE   "n"           "y"
2  Depends      VAR2          "CONST_FALSE"
3  HasPrompts   CONST_FALSE   0
4  HasPrompts   VAR1          1
5  HasPrompts   VAR2          1
6  Item         CONST_FALSE   boolean
7  ItemSelects  VAR1          "VAR2"        "y"
8  Item         VAR1          boolean
9  Item         VAR2          boolean
```

Figure 3.19: dumpconf's translation of Figure 3.18 (correct).

Figure 3.19 shows exactly the same information as encoded in Figure 3.18. Thus, we categorized the translation as correct.

rsf2model

Line 3 in Figure 3.20 on the next page means that if `VAR2` was selected also `CONST_FALSE` has to be selected. However, this is contradictory to the information of the KConfig model from Figure 3.18.

```
1 CONFIG_CONST_FALSE "(CONFIG_n)"
2 CONFIG_VAR1 "CONFIG_VAR2"
3 CONFIG_VAR2 "CONFIG_CONST_FALSE"
4 UNDERTAKER_SET SCHEMA_VERSION 1.1
5 CONFIG_X86 ""
6 CONFIG_n
7 CONFIG_y
8 CONFIG_m
```

Figure 3.20: rsf2model's translation of Figure 3.18 (error in line 3).

Satyr

```
1  c File Format Version: 2.0
2  c Generated by satyr
3  c Type info:
4  c c sym <symbolname> <typeid>
5  c with <typeid> being an integer out of:
6  c enum {S_BOOLEAN=1, S_TRISTATE=2, S_INT=3, S_HEX=4, S_STRING=5, S_OTHER=6}
7  c variable names:
8  c c var <variablename> <cnfvar>
9  c sym CONST_FALSE 1
10 c sym UNAME_RELEASE 5
11 c sym VAR1 1
12 c sym VAR2 1
13 c sym ____MODULES_MAGIC_INTERNAL_VAR____ 1
14 c var CONFIG_CONST_FALSE 5
15 c var CONFIG_UNAME_RELEASE 2
16 c var CONFIG_VAR1 3
17 c var CONFIG_VAR2 4
18 c var CONFIG_____MODULES_MAGIC_INTERNAL_VAR____ 1
19 p cnf 8 13
20 -1 0
21 6 4 0
22 6 -5 0
23 -6 -4 5 0
24 7 -6 0
25 7 -3 0
26 -7 6 3 0
27 7 0
28 8 3 0
29 8 -4 0
30 -8 -3 4 0
31 8 0
32 -5 0
```

Figure 3.21: Satyr's CNF translation of Figure 3.10 on page 60 (correct).

The boolean formula of Figure 3.21 force the selection of `VAR2` when `VAR1` was selected. Also `CONST_FALSE` is permanently set to `false` (cf. line 32). Thus, the translation is correct.

3.3.2 KConfig Reader

RSF

```
1  Item          CONST_FALSE   boolean
2  HasPrompts    CONST_FALSE   0
3  Default       CONST_FALSE   "'n'"          "y"
4  Item          VAR1          boolean
5  Prompt        VAR1          "y"
6  HasPrompts    VAR1          1
7  ItemSelects   VAR1          "VAR2"         "y"
8  Item          VAR2          boolean
9  Depends       VAR2          "CONST_FALSE"
10 Prompt        VAR2          "CONST_FALSE"
11 HasPrompts    VAR2          1
```

Figure 3.22: KConfig Reader's RSF translation of Figure 3.18 on the previous page (correct).

Figure 3.22 shows the same information as Figure 3.19 on page 65. KConfig Reader additionally calculates when the variable is visible to the user and can be configured by him (`Prompt`). Also this extra information is correct.

Model

```
1 #item CONST_FALSE
2 !def(CONST_FALSE)
3 #item VAR1
4 #item VAR2
5 (def(CONST_FALSE)|def(VAR1)|!def(VAR2))
6 (def(CONST_FALSE)|def(VAR1)|!def(VAR2))
7 (!def(VAR1)|def(VAR2))
8 (!def(VAR1)|def(VAR2))
```

Figure 3.23: KConfig Reader's Model translation of Figure 3.18 on page 65 (correct).

The constraint in line 5 and 6 of Figure 3.23 will only be violated if `CONST_FALSE` and `VAR2` are set to `false` (`depends on` constraint fulfilled) and `VAR1` is set to `true` (select constraint violated). Thus, the precedence of the `select` and `depends on` constraints is translated correctly.

CNF

```
1  c 1 CONST_FALSE
2  c 2 VAR1
3  c 3 VAR2
4  c 4 MODULES
5  p cnf 4 6
6  1 2 -3 0
7  1 2 -3 0
8  -2 3 0
9  -2 3 0
10 -1 0
11 -4 0
```

Figure 3.24: KConfig Reader's CNF translation of Figure 3.18 on page 65 (correct).

Figure 3.24 contains the same constraints as Figure 3.23 on the previous page plus a constraint denoting that the third state of tristate config options cannot be activated (cf. line 11). Thus, this translation is correct.

3.3.3 KConfig Reader (XML)

Model & CNF

KConfig Reader (XML)'s Model and CNF translations of Figure 3.18 on page 65 are identical to KConfig Reader's translations (cf. Section 3.3.2) and therefore correct.

3.3.4 LVAT

After removing all constant variables (e.g. line 30) and resolving related constraints (e.g. line 32) from Figure 3.25 on the next page there are only constraints left, that require that `VAR1` and `VAR2` are equal. Thus, this translation is correct.

```
c 1$ _X9
c 2$ _X5
c 3$ _X1_m
c 4$ _X10
c 5$ _X11
c 6$ _X5_m
c 7$ _X7_m
c 8$ _X2
c 9$ _X1
c 10$ _X11_m
c 11$ _X6
c 12 VAR1
c 13 VAR1_m
c 14 CONST_FALSE
c 15$ _X10_m
c 16$ _X3
c 17$ _X4_m
c 18 VAR2
c 19$ _X7
c 20 VAR2_m
c 21$ _X3_m
c 22$ _X4
c 23$ _X8_m
c 24$ _X6_m
c 25$ _X9_m
c 26$ _X8
c 27 CONST_FALSE_m
c 28$ _X2_m
p cnf 28 61
−9 0
−3 0
−9 14 27 0
−3 14 27 0
−9 −3 −14 27 0
−14 27 0
14 −27 0
8 0
−28 0
−8 −14 9 0
−8 −9 14 0
−8 −27 3 0
−8 −3 27 0
−16 0
−21 0
−16 −14 9 0
−16 −9 14 0
−16 −27 3 0
−16 −3 27 0
−22 0
−17 0
−2 0
−6 0
−2 12 13 0
−6 12 13 0
−2 −6 −12 13 0
−12 13 0
12 −13 0
−11 0
−24 0
−11 −12 2 0
−11 −2 12 0
−11 −13 6 0
−11 −6 13 0
−19 0
−7 0
−26 12 0
−12 26 0
−23 13 0
−13 23 0
−26 1 0
−1 26 0
−23 25 0
−25 23 0
−1 18 20 0
−25 18 20 0
−1 −25 −18 20 0
−18 20 0
18 −20 0
14 −27 0
12 −13 0
−4 −14 0
−4 −27 0
14 27 4 0
−15 0
−4 −18 1 0
−4 −1 18 0
−4 −20 25 0
−4 −25 20 0
−5 0
−10 0
```

Figure 3.25: LVAT's CNF translation of Figure 3.18 on page 65 (correct).

3.4 Missing Config Options

In Section 2.1, we explained that KConfig is also able to handle constraints containing not existing config options. This is needed, since different architectures may reuse some but not all KConfig files. In this case, the missing config variables will be treated as `n` (`false`).

Figure 3.26 shows the example, which we used to analyse whether the tools are capable of handling constraints containing such missing config options. `VAR` is dependant of `NOT_EXISTING`. Since `NOT_EXISTING` does not exist in this model, `VAR` will be permanently deselected.

```
1 config VAR
2     bool "A variable"
3     depends on NOT_EXISTING
```

Figure 3.26: KConfig example for testing the translation of a dependency to a config option, which is not part of the KConfig model.

3.4.1 Undertaker

dumpconf

```
1 Depends       VAR     "NOT_EXISTING"
2 HasPrompts    VAR     1
3 Item          VAR     boolean
```

Figure 3.27: dumpconf's translation of Figure 3.26 on the next page (correct).

Figure 3.27 contains the same logical information as the related KConfig model. Thus, we categorized this translation as correct.

rsf2model

```
1 CONFIG_VAR "CONFIG_NOT_EXISTING"
2 UNDERTAKER_SET SCHEMA_VERSION 1.1
3 CONFIG_X86 ""
4 CONFIG_n
5 CONFIG_y
6 CONFIG_m
```

Figure 3.28: rsf2model's translation of Figure 3.27 (correct).

Figure 3.28 makes `CONFIG_VAR` dependant of `CONFIG_NOT_EXISTING`, which is also not part of the translated model. Thus, we categorized this translation as correct.

Satyr

```
1  c File Format Version: 2.0
2  c Generated by satyr
3  c Type info:
4  c c sym <symbolname> <typeid>
5  c with <typeid> being an integer out of:
6  c enum {S_BOOLEAN=1, S_TRISTATE=2, S_INT=3, S_HEX=4, S_STRING=5, S_OTHER=6}
7  c variable names:
8  c c var <variablename> <cnfvar>
9  c sym UNAME_RELEASE 5
10 c sym VAR 1
11 c sym ____MODULES_MAGIC_INTERNAL_VAR____ 1
12 c var CONFIG_UNAME_RELEASE 2
13 c var CONFIG_VAR 3
14 c var CONFIG_____MODULES_MAGIC_INTERNAL_VAR____ 1
15 p cnf 3 2
16 -1 0
17 -3 0
```

Figure 3.29: Satyr's CNF translation of Figure 3.26 (correct).

Figure 3.29 on the next page does not translate `NOT_EXISTING`, but it creates a constraint for the permanent deselection of `CONFIG_VAR` in line 17. Thus, we categorized also this translation as correct.

3.4.2 KConfig Reader

RSF

```
1  Item          VAR          boolean
2  Depends       VAR          " 'NOT_EXISTING' "
3  Prompt        VAR          " 'NOT_EXISTING' "
4  HasPrompts    VAR          1
```

Figure 3.30: KConfig Reader's RSF translation of Figure 3.26 (correct).

Figure 3.30 contains the same logical information as the related KConfig model. Thus, we categorized this translation as correct.

Model

```
1 #item VAR
2 !def(VAR)
3 !def(VAR)
```

Figure 3.31: KConfig Reader's Model translation of Figure 3.26 (correct).

Figure 3.31 shows how KConfig Reader translates the model from Figure 3.26 on the previous page. The constraint in lines 2 and 3 permanently deselects `VAR`, which is correct.

CNF

```
1 c 1 VAR
2 c 2 MODULES
3 p cnf 2 3
4 -1 0
5 -1 0
6 -2 0
```

Figure 3.32: KConfig Reader's CNF translation of Figure 3.26 on page 70 (correct).

Figure 3.32 shows how KConfig Reader translates the KConfig model of Figure 3.26 on page 70 into CNF formula. The constraint in lines 4 and 5 permanently deselects `VAR`, which is correct.

3.4.3 KConfig Reader (XML)

Model & CNF

KConfig Reader (XML)'s Model and CNF translations of Figure 3.26 are identical to KConfig Reader's translations (cf. Section 3.4.2 on the previous page) and therefore correct.

3.4.4 LVAT

```
1  c 1$ _X1_m
2  c 2 NOT_EXISTING
3  c 3$ _X2
4  c 4$ _X1
5  c 5 VAR_m
6  c 6$ _X3
7  c 7$ _X3_m
8  c 8 NOT_EXISTING_m
9  c 9 VAR
10 c 10$ _X2_m
11 p cnf 10 19
12 -4 0
13 -1 0
14 -4 9 5 0
15 -1 9 5 0
16 -4 -1 -9 5 0
17 -9 5 0
18 9 -5 0
19 2 -8 0
20 -2 0
21 -3 -2 0
22 -3 -8 0
23 2 8 3 0
24 -10 0
25 -3 -9 4 0
26 -3 -4 9 0
27 -3 -5 1 0
28 -3 -1 5 0
29 -6 0
30 -7 0
```

Figure 3.33: LVAT's CNF translation of Figure 3.26 on page 70 (correct).

In Figure 3.33 `NOT_EXISTING` is correctly set to be permanently false in line 20. `VAR` depends on `NOT_EXISTING` via `_X2` and thus can only be false. Thus, this translation is correct.

3.5 Selection of Nested Config Options

In Section 2.4.1, we showed that it is also possible to model a `select` constraint to a nested config option of a choice. However, such a `select` constraint is inoperable. We used the model from Figure 2.15 on page 28 for our analysis.

3.5.1 Undertaker

dumpconf

```
1  Choice          CHOICE_1        required        boolean
2  ChoiceItem      BOOL_VAL1       CHOICE_1
3  ChoiceItem      BOOL_VAL2       CHOICE_1
4  Depends         BOOL_VAL1       "CHOICE_1"
5  Depends         BOOL_VAL2       "CHOICE_1"
6  HasPrompts      BOOL_VAL1       1
7  HasPrompts      BOOL_VAL2       1
8  HasPrompts      BOOL_VAR        1
9  Item            BOOL_VAL1       boolean
10 Item            BOOL_VAL2       boolean
11 Item            BOOL_VAR        boolean
12 ItemSelects     BOOL_VAR        "BOOL_VAL2"     "y"
```

Figure 3.34: dumpconf's translation of Figure 2.15 (problematic).

Figure 3.34 shows that the `select` constraint was translated (cf. line 12). However, the constraint was not be rewritten. Downstream analysis tools must be aware of the KConfig behaviour described in Section 2.4.1, otherwise the interpretation of such RSF files will result in an inconsistent model. Thus, this strict translation of the unused `select` constraint is not wrong per se, but it could lead to translation errors at the next analysis step. Hence, we categorized this translation as problematic.

rsf2model

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

Figure 3.35: rsf2model's translation of Figure 3.34 (error in line 3).

Figure 3.35 on the preceding page shows that the `select` constraint was translated into boolean formula (cf. line 3). The selection of `BOOL_VAR` causes the automatic selection of `BOOL_VAL2`, which is not correct.

Satyr

```
1  c File Format Version: 2.0
2  c Generated by satyr
3  c Type info:
4  c c sym <symbolname> <typeid>
5  c with <typeid> being an integer out of:
6  c enum {S_BOOLEAN=1, S_TRISTATE=2, S_INT=3, S_HEX=4, S_STRING=5, S_OTHER=6}
7  c variable names:
8  c c var <variablename> <cnfvar>
9  c meta_value ALWAYS_ON CONFIG_CHOICE_0
10 c sym BOOL_VAL1 1
11 c sym BOOL_VAL2 1
12 c sym BOOL_VAR 1
13 c sym CHOICE_0 1
14 c sym UNAME_RELEASE 5
15 c sym ____MODULES_MAGIC_INTERNAL_VAR____ 1
16 c var CONFIG_BOOL_VAL1 2
17 c var CONFIG_BOOL_VAL2 3
18 c var CONFIG_BOOL_VAR 10
19 c var CONFIG_CHOICE_0 1
20 c var CONFIG_UNAME_RELEASE 9
21 c var CONFIG_____MODULES_MAGIC_INTERNAL_VAR____ 8
22 p cnf 12 23
23 1 0
24 -4 2 0
25 -4 -3 0
26 4 -2 3 0
27 -5 -2 0
28 -5 3 0
29 5 2 -3 0
30 6 -4 0
31 6 -5 0
32 -6 4 5 0
33 7 1 0
34 7 -6 0
35 -7 -1 6 0
36 7 0
37 -8 0
38 11 3 0
39 11 -1 0
40 -11 -3 1 0
41 11 0
42 12 2 0
43 12 -1 0
44 -12 -2 1 0
45 12 0
```

Figure 3.36: Satyr's CNF translation of Figure 2.15 on page 28 (correct).

Figure 3.36 on the preceding page shows that `BOOL_VAR` (No. 10) is not included in one of the constraints. Consequently the `select` constraint between the top level config option `BOOL_VAR` and the nested config option of the choice `BOOL_VAL2` was not translated. Thus, the translation of `select` constraints to config options nested in a choice is correct.

3.5.2 KConfig Reader

RSF

```
1  Item            BOOL_VAR        boolean
2  Prompt          BOOL_VAR        "y"
3  HasPrompts      BOOL_VAR        1
4  ItemSelects     BOOL_VAR        "BOOL_VAL2"     "y"
5  #startchoice
6  Choice          CHOICE_1        required        boolean
7  Prompt          CHOICE_1        "y"
8  HasPrompts      CHOICE_1        1
9  #choice value
10 ChoiceItem      BOOL_VAL1       CHOICE_1
11 Item            BOOL_VAL1       boolean
12 Depends         BOOL_VAL1       "CHOICE_1"
13 Prompt          BOOL_VAL1       "CHOICE_1"
14 HasPrompts      BOOL_VAL1       1
15 #choice value
16 ChoiceItem      BOOL_VAL2       CHOICE_1
17 Item            BOOL_VAL2       boolean
18 Depends         BOOL_VAL2       "CHOICE_1"
19 Prompt          BOOL_VAL2       "CHOICE_1"
20 HasPrompts      BOOL_VAL2       1
21 #choice value
22 #endchoice
```

Figure 3.37: KConfig Reader's RSF translation of Figure 2.15 on page 28 (problematic).

Figure 3.37 shows that KConfig Reader is handling the `select` constraint in the same way as dumpconf does (cf. line 4). Thus, also this translation is categorized as problematic.

Model

Lines 3 and 4 of Figure 3.38 on the following page show that the selection of `BOOL_VAR` implicates the selection of `BOOL_VAL2`. Thus, the `select` constraint was translated incorrectly.

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

Figure 3.38: KConfig Reader's Model translation of Figure 2.15 on page 28 (error in line 3 and 4).

CNF

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

Figure 3.39: KConfig Reader's CNF translation of Figure 2.15 on page 28 (error in line 7 and 8).

Lines 7 and 8 of Figure 3.39 show that the selection of `BOOL_VAR` implicates the selection of `BOOL_VAL2`. Thus, the `select` constraint was translated incorrectly.

3.5.3 KConfig Reader (XML)

Model

In Figure 3.40 on the following page, line 7 contains the same incorrect constraint as KConfig Reader's Model translation (cf. Section 3.5.2 on the previous page). Thus, this translation is also incorrect.

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

Figure 3.40: KConfig Reader (XML)'s Model translation of Figure 2.15 on page 28 (error in line 7).

CNF

In Figure 3.41, line 7 contains the same incorrect constraint as KConfig Reader's CNF translation (cf. Section 3.5.2). Thus, this translation is also incorrect.

```
1  c 2 CHOICE_1
2  c 3 BOOL_VAR
3  c 1 BOOL_VAL1
4  c 4 BOOL_VAL2
5  c 5 MODULES
6  p cnf 5 12
7  -1 2 0
8  -1 2 0
9  2 0
10 2 3 -4 0
11 2 3 -4 0
12 -3 4 0
13 -3 4 0
14 4 1 -2 0
15 -4 2 0
16 -1 2 0
17 -4 -1 0
18 -5 0
```

Figure 3.41: KConfig Reader (XML)'s CNF translation of Figure 2.15 on page 28 (error in line 7).

3.5.4 LVAT

In Figure 3.42, `BOOL_VAR` selects `BOOL_VAL2` via `_X7` and `_X8`. Thus, the translation is incorrect, as the select constraint is not ignored.

```
c 1$ _X9
c 2 BOOL_VAR
c 3$ _X5
c 4$ _X1_m
c 5$ _X10
c 6 BOOL_VAR_m
c 7$ _X5_m
c 8$ _X7_m
c 9 BOOL_VAL2_m
c 10$ _X2
c 11$ _X1
c 12$ _X6
c 13$ _X10_m
c 14 BOOL_VAL2
c 15$ _X3
c 16$ _X4_m
c 17$ _X7
c 18 BOOL_VAL1_m
c 19$ _X3_m
c 20$ _X4
c 21 BOOL_VAL1
c 22$ _X8_m
c 23$ _X6_m
c 24$ _X9_m
c 25$ _X8
c 26$ _X2_m
p cnf 26 54
−11 0
−4 0
−11 2 6 0
−4 2 6 0
−11 −4 −2 6 0
−2 6 0
2 −6 0
−10 0
−26 0
−10 −2 11 0
−10 −11 2 0
−10 −6 4 0
−10 −4 6 0
−15 0
−19 0
−20 0
−16 0
−20 21 18 0
−16 21 18 0
−20 −16 −21 18 0
−21 18 0
21 −18 0
−3 0
−7 0
−3 −21 20 0
−3 −20 21 0
−3 −18 16 0
−3 −16 18 0
−12 0
−23 0
−17 2 0
−2 17 0
−8 6 0
−6 8 0
−17 25 0
−25 17 0
−8 22 0
−22 8 0
−25 14 9 0
−22 14 9 0
−25 −22 −14 9 0
−14 9 0
14 −9 0
2 −6 0
−1 0
−24 0
−1 −14 25 0
−1 −25 14 0
−1 −9 22 0
−1 −22 9 0
−5 0
−13 0
21 14 0
−21 −14 0
```

Figure 3.42: LVAT's CNF translation of Figure 2.15 on page 28 (error).

3.6 Default Value `m` for Booleans

In Section 2.4.2, we demonstrated how config options of type `bool` or `tristate` handle the default value `m`. In this section, we show how the different tools cope with this default value if the third state is enabled. We modified the example from Figure 2.18 on page 30 and tested the translation with the example of Figure 3.43. The modified example does not allow any user input to fore the tools to create more meaningful outputs.

```
1 config MODULES
2     def_bool y
3     option modules
4
5 config VAR1
6     def_bool m
7
8 config VAR2
9     def_tristate m
```

Figure 3.43: Modification of Figure 2.18 (no user input is possible).

The model of Figure 3.43 does not allow the user to perform any changes. Although the third state was permanently activated (cf. lines 2 and 3), `VAR1` may not be selected to `m` as it is of type `bool`. The resulting `.config` file will always contain the selection of `MODULES` and `VAR1` to `y`. `VAR2` will always be set to `m`.

3.6.1 Undertaker

dumpconf

```
1 Default      MODULES   "y"   "y"
2 Default      VAR1      "m"   "y"
3 Default      VAR2      "m"   "y"
4 HasPrompts   MODULES   0
5 HasPrompts   VAR1      0
6 HasPrompts   VAR2      0
7 Item         MODULES   boolean
8 Item         VAR1      boolean
9 Item         VAR2      tristate
```

Figure 3.44: dumpconf's translation of Figure 3.43 (problematic).

Figure 3.44 shows that dumpconf does translate the default statements as well as the types directly 1:1 into the RSF format. Thus, the downstream analysis tool have to handle the `default` statements correctly.

Note: The translated RSF file does not contain the complete information to handle the default values correctly, because of the absence of the `option modules` information inside the translated RSF file. However, this was already analysed in Section 3.2.

Satyr

```
c File Format Version: 2.0
c Generated by satyr
c Type info:
c c sym <symbolname> <typeid>
c with <typeid> being an integer out of:
c enum {S_BOOLEAN=1, S_TRISTATE=2, S_INT=3, S_HEX=4, S_STRING=5, S_OTHER=6}
c variable names:
c c var <variablename> <cnfvar>
c meta_value ALWAYS_ON CONFIG_VAR1 CONFIG_MODULES
c sym MODULES 1
c sym UNAME_RELEASE 5
c sym VAR1 1
c sym VAR2 2
c sym ____MODULES_MAGIC_INTERNAL_VAR____ 1
c var CONFIG_MODULES 2
c var CONFIG_UNAME_RELEASE 6
c var CONFIG_VAR1 7
c var CONFIG_VAR2 8
c var CONFIG_VAR2_MODULE 9
c var CONFIG_____MODULES_MAGIC_INTERNAL_VAR____ 1
p cnf 12 23
3 1 0
3 -2 0
-3 -1 2 0
4 2 0
4 -1 0
-4 -2 1 0
-5 3 0
-5 4 0
5 -3 -4 0
5 0
7 0
10 -8 0
10 -9 0
-10 8 9 0
-11 -8 0
-11 10 0
11 8 -10 0
11 0
-12 8 0
-12 9 0
12 -8 -9 0
-12 0
2 0
```

Figure 3.45: Satyr's CNF translation of Figure 3.43 on the previous page (correct).

Figure 3.45 forces the permanent selection of `MODULES`, `VAR1`, and `VAR2_MODULE`. Therefore, the translation is correct.

rsf2model

```
1  CONFIG_MODULES "(___FREE___1)"
2  CONFIG_VAR1 "(___FREE___0)"
3  CONFIG_VAR2 "!CONFIG_VAR2_MODULE"
4  CONFIG_VAR2_MODULE "!CONFIG_VAR2 && CONFIG_MODULES"
5  UNDERTAKER_SET ALWAYS_ON "CONFIG_MODULES"
6  UNDERTAKER_SET SCHEMA_VERSION 1.1
7  CONFIG_X86 ""
8  CONFIG_n
9  CONFIG_y
10 CONFIG_m
```

Figure 3.46: rsf2model's translation of Figure 3.44 on page 79 (error in line 5).

In Figure 3.46, line 5 should also list `VAR1` as permanently selected (always on). Undertaker will list this variable as permanently selected, if the default value of `VAR1` in Figure 3.43 on page 79 is changed to `y`. But KConfig will handle both default values exactly in the same manner. We categorized this translation as incorrect, because Undertaker treats both default values differently.

3.6.2 KConfig Reader

RSF

```
1 Item          MODULES     boolean
2 HasPrompts    MODULES     0
3 Default       MODULES     "'y'"       "y"
4 Item          VAR1        boolean
5 HasPrompts    VAR1        0
6 Default       VAR1        "'m'"       "y"
7 Item          VAR2        tristate
8 HasPrompts    VAR2        0
9 Default       VAR2        "'m'"       "y"
```

Figure 3.47: KConfig Reader's RSF translation of Figure 3.43 on page 79 (proplematic).

Figure 3.47 contains the same information as Figure 3.44 on page 79. Thus, also this translation is categorized as problematic.

Model

```
1 #item MODULES
2 def(MODULES)
3 #item VAR1
4 def(VAR1)
5 #item VAR2
6 (!def(VAR2_MODULE) | def(MODULES))
7 (!def(VAR2) |!def(VAR2_MODULE))
8 (def(VAR2) | def(VAR2_MODULE))
9 (!def(MODULES) |!def(VAR2))
```

Figure 3.48: KConfig Reader's Model translation of Figure 3.43 on page 79 (correct).

The translated model shown in Figure 3.48 defines `MODULES` and `VAR1` as permanently selected. The constrains of lines 7 and 8 force that `VAR2` is either set to `m` or `y`, which is the correct translation of a `default m` statement of a tristate config option. Finally, line 9 forces `VAR2` set to `m`, if the third state is enabled. Thus, this translation is correct.

CNF

```
1  c 1 VAR2_MODULE
2  c 2 MODULES
3  c 3 VAR2
4  c 4 VAR1
5  p cnf 4 6
6  -1 2 0
7  -3 -1 0
8  3 1 0
9  -2 -3 0
10 2 0
11 4 0
```

Figure 3.49: KConfig Reader's CNF translation of Figure 3.43 on page 79 (correct).

Figure 3.49 contains the same constraines as Figure 3.48. Thus, also this translation is correct.

3.6.3 KConfig Reader (XML)

Model & CNF

KConfig Reader (XML)'s Model and CNF translations of Figure 3.43 on page 79 are identical to KConfig Reader's translations (cf. Section 3.6.2) and therefore correct.

3.6.4 LVAT

After removing all constant variables (e.g. line 32) and resolving related constraints (e.g. line 34) from Figure 3.50 on the next page there are constraints left that permanently set

`VAR1` to `true` and `VAR1_MODULE` to `false`. This is an invalid state for Boolean variables (cf. Section 3.1.3 on page 55). Thus, the translation is incorrect.

```
c 1$ _X9
c 2$ _X5
c 3$ _X12_m
c 4$ _X1_m
c 5$ _X10
c 6$ _X11
c 7$ _X5_m
c 8$ _X7_m
c 9$ _X2
c 10$ _X1
c 11$ _X11_m
c 12$ _X6
c 13 VAR1
c 14 MODULES_m
c 15 VAR1_m
c 16 MODULES
c 17$ _X10_m
c 18$ _X3
c 19$ _X4_m
c 20 VAR2
c 21$ _X7
c 22 VAR2_m
c 23$ _X3_m
c 24$ _X4
c 25$ _X8_m
c 26$ _X6_m
c 27$ _X12
c 28$ _X9_m
c 29$ _X8
c 30$ _X2_m
p cnf 30 59
−10 0
−4 0
−10 16 14 0
−4 16 14 0
−10 −4 −16 14 0
−16 14 0
16 −14 0
9 0
−30 0
−9 16 0
−9 14 0
−18 0
−23 0
−18 −16 10 0
−18 −10 16 0
−18 −14 4 0
−18 −4 14 0
−24 0
−19 0
−2 0
−7 0
−2 13 15 0
−7 13 15 0
−2 −7 −13 15 0
−13 15 0
13 −15 0
12 0
−26 0
−12 13 0
−12 −15 7 0
−12 −7 15 0
−21 0
−8 0
−21 −13 2 0
−21 −2 13 0
−21 −15 7 0
−21 −7 15 0
−29 0
−25 0
−1 0
−28 0
−1 20 22 0
−28 20 22 0
−1 −28 −20 22 0
20 −22 0
−20 22 16 0
5 0
−17 0
−5 20 0
−5 −22 28 0
−5 −28 22 0
−6 0
−11 0
−6 −20 1 0
−6 −1 20 0
−6 −22 28 0
−6 −28 22 0
−27 0
−3 0
```

Figure 3.50: LVAT's CNF translation of Figure 3.43 on page 79 (error).

3.7 Default Value `m` for Tristates

In Section 2.4.2, we demonstrated how config options of type `bool` or `tristate` handle the default value `m`. In this section, we analyse how the tools handle default values of `tristate` config options if the third state is disabled.

```
config MODULES
    def_bool n
    option modules

config VAR
    def_tristate m
```

Figure 3.51: Modification of Figure 2.18 on page 30: The tristate config option will be permanently selected (no user input possible).

3.7.1 Undertaker

dumpconf

```
Default       MODULES     "n"          "y"
Default       VAR         "m"          "y"
HasPrompts    MODULES     0
HasPrompts    VAR         0
Item          MODULES     boolean
Item          VAR         tristate
```

Figure 3.52: dumpconf's translation of Figure 3.51 (correct).

The translated model of Figure 3.52 contains exactly the same information as the origin model from Figure 3.51. Downstream analysis tools need to be aware that the third state of `tristate` config options is dependant of the `MODULES` variable, but this should not be problematic since this is a fundamental concept of KConfig. Therefore, we categorized this translation as correct.

rsf2model

```
CONFIG_MODULES "(CONFIG_n)"
CONFIG_VAR "!CONFIG_VAR_MODULE"
CONFIG_VAR_MODULE "!CONFIG_VAR && CONFIG_MODULES"
UNDERTAKER_SET SCHEMA_VERSION 1.1
CONFIG_X86 ""
CONFIG_n
CONFIG_y
CONFIG_m
```

Figure 3.53: rsf2model's translation of Figure 3.52 (correct).

In Figure 3.53 on the previous page, `CONFIG_VAR` (`VAR` = `y`) was made dependant of the deselection of `CONFIG_MODULES` (cf. line 2). We categorized this translation as correct, since `CONFIG_MODULES` is also permanently deselected (cf. line 1).

Satyr

```
1  c File Format Version: 2.0
2  c Generated by satyr
3  c Type info:
4  c c sym <symbolname> <typeid>
5  c with <typeid> being an integer out of:
6  c enum {S_BOOLEAN=1, S_TRISTATE=2, S_INT=3, S_HEX=4, S_STRING=5, S_OTHER=6}
7  c variable names:
8  c c var <variablename> <cnfvar>
9  c sym MODULES 1
10 c sym UNAME_RELEASE 5
11 c sym VAR 2
12 c sym ____MODULES_MAGIC_INTERNAL_VAR____ 1
13 c var CONFIG_MODULES 2
14 c var CONFIG_UNAME_RELEASE 6
15 c var CONFIG_VAR 7
16 c var CONFIG_VAR_MODULE 8
17 c var CONFIG_____MODULES_MAGIC_INTERNAL_VAR____ 1
18 p cnf 11 22
19 3 1 0
20 3 -2 0
21 -3 -1 2 0
22 4 2 0
23 4 -1 0
24 -4 -2 1 0
25 -5 3 0
26 -5 4 0
27 5 -3 -4 0
28 5 0
29 -2 0
30 9 -7 0
31 9 -8 0
32 -9 7 8 0
33 -10 -7 0
34 -10 9 0
35 10 7 -9 0
36 10 0
37 -11 7 0
38 -11 8 0
39 11 -7 -8 0
40 -11 0
```

Figure 3.54: Satyr's CNF translation of Figure 3.51 (error in lines 33 and 36).

Line 36 of Figure 3.54 specifies that variable no. 10 should be permanently set to `true`. Variable no. 10 is used to model some non CNF constraints via implications. In line 33, variable no. 10 is used to permanently deselect `CONFIG_VAR` (`VAR` = `n`), which is incorrect.

3.7.2 KConfig Reader

RSF

```
1 Item          MODULES     boolean
2 HasPrompts    MODULES     0
3 Default       MODULES     "'n'"       "y"
4 Item          VAR         tristate
5 HasPrompts    VAR         0
6 Default       VAR         "'m'"       "y"
```

Figure 3.55: KConfig Reader's RSF translation of Figure 3.51 on page 84 (correct).

Figure 3.55 contains the same information as Figure 3.52 on page 84. Thus, also this translation is categorized as correct.

Model

```
1 #item MODULES
2 !def(MODULES)
3 #item VAR
4 (!def(VAR_MODULE) | def(MODULES))
5 (!def(VAR) |!def(VAR_MODULE))
6 (def(VAR) | def(VAR_MODULE))
7 (!def(MODULES) |!def(VAR))
```

Figure 3.56: KConfig Reader's Model translation of Figure 3.51 on page 84 (correct).

Lines 5 and 6 of Figure 3.56 encode `VAR XOR VAR_MODULE`. Line 4 specifies that if `VAR_MODULE` was selected that `MODLUES` has also to be selected, but this variable is permanently deselected. As a consequence, `VAR_MODULE` must also be deselected and `VAR` must be permanently selected. Thus, the translation is correct.

CNF

```
1 c 1 VAR_MODULE
2 c 2 MODULES
3 c 3 VAR
4 p cnf 3 5
5 -1 2 0
6 -3 -1 0
7 3 1 0
8 -2 -3 0
9 -2 0
```

Figure 3.57: KConfig Reader's CNF translation of Figure 3.51 on page 84 (correct).

Figure 3.57 contains the same constrains as Figure 3.56. Thus, also this translation is correct.

3.7.3 KConfig Reader (XML)

Model & CNF

KConfig Reader (XML)'s Model and CNF translations of Figure 3.51 on page 84 are identical to KConfig Reader's translations (cf. Section 3.7.2 on the preceding page) and therefore correct.

3.7.4 LVAT

```
1  c 1$ _X5
2  c 2$ _X1_m
3  c 3$ _X5_m
4  c 4$ _X7_m
5  c 5$ _X2
6  c 6$ _X1
7  c 7$ _X6
8  c 8 VAR_m
9  c 9 MODULES_m
10 c 10 MODULES
11 c 11$ _X3
12 c 12$ _X4_m
13 c 13$ _X7
14 c 14$ _X3_m
15 c 15$ _X4
16 c 16$ _X8_m
17 c 17$ _X6_m
18 c 18 VAR
19 c 19$ _X8
20 c 20$ _X2_m
21 p cnf 20 41
22 -6 0
23 -2 0
24 -6 10 9 0
25 -2 10 9 0
26 -6 -2 -10 9 0
27 -10 9 0
28 10 -9 0
29 5 0
30 -20 0
31 -5 -10 6 0
32 -5 -6 10 0
33 -5 -9 2 0
34 -5 -2 9 0
35 -11 0
36 -14 0
37 -11 -10 6 0
38 -11 -6 10 0
39 -11 -9 2 0
40 -11 -2 9 0
41 -15 0
42 -12 0
43 -1 0
44 -3 0
45 -1 18 8 0
46 -3 18 8 0
47 -1 -3 -18 8 0
48 18 -8 0
49 -18 8 10 0
50 7 0
51 -17 0
52 -7 18 0
53 -7 -8 3 0
54 -7 -3 8 0
55 -13 0
56 -4 0
57 -13 -18 1 0
58 -13 -1 18 0
59 -13 -8 3 0
60 -13 -3 8 0
61 -19 0
62 -16 0
```

Figure 3.58: LVAT's CNF translation of Figure 3.51 on page 84 (error).

After removing all constant variables (e.g. line 22) and resolving related constraints (e.g. line 24) from Figure 3.58, there are constraints left that permanently set `VAR` to the Tristate state, regardless of the `MODULES` variable. Thus, this translation is incorrect.

3.8 Tristate Choice with Boolean Config Options

In Section 2.4.3, we discussed how KConfig handles choices with config options of a different type than the surrounding choice. In this section, we analyse how the tools handle the combination of `tristate` choices with nested config options of type `bool`. In Section 3.9 we analyse the contrary combination.

Figure 3.59 shows the KConfig model, which we used for the analysis of this section. `BOOL_VAL` may only be selected if the surrounding choice was set to `y` (permanently selected). In this case, also `TRISTATE_VAL` may only be selected as `y` and only if `BOOL_VAL` was not selected. If the choice was selected as `m`, `TRISTATE_VAL` maybe selected as `m` and `BOOL_VAL` must not be selected.

```
config MODULES
    def_bool y
    option modules

choice
    tristate "Tristate Choice"

    config TRISTATE_VAL
        tristate "Tristate Value"
    config BOOL_VAL
        bool "Boolean Value"
endchoice
```

Figure 3.59: KConfig example for testing the translation of a tristate choice with nested boolean config options.

3.8.1 Undertaker

dumpconf

```
Choice          CHOICE_1        required        tristate
ChoiceItem      BOOL_VAL        CHOICE_1
ChoiceItem      TRISTATE_VAL    CHOICE_1
Default         MODULES         "y"             "y"
Depends         BOOL_VAL        "<choice>=y && CHOICE_1"
Depends         TRISTATE_VAL    "CHOICE_1"
HasPrompts      BOOL_VAL        1
HasPrompts      MODULES         0
HasPrompts      TRISTATE_VAL    1
Item            BOOL_VAL        boolean
Item            MODULES         boolean
Item            TRISTATE_VAL    tristate
```

Figure 3.60: dumpconf's translation of Figure 3.59 (correct).

Figure 3.60 shows how dumpconf translates the KConfig model from 3.59 on the previous page into RSF format. In Lines 2 and 3, `BOOL_VAL` and `TRISTATE_VAL` are specified as nested elements of the choice. Lines 5 and 6 model the situations when the nested elements can be selected. While `TRISTATE_VAL` is only dependant from the choice, `BOOL_VAL` was explicitly made dependant of the situation that the choice is permanently selected (`<choice>=y`). Thus, the translation is correct.

rsf2model

```
1  CONFIG_BOOL_VAL "CONFIG_CHOICE_1"
2  CONFIG_CHOICE_1 "!CONFIG_CHOICE_1_MODULE && CONFIG_CHOICE_1_META &&
       ((CONFIG_BOOL_VAL && !CONFIG_TRISTATE_VAL) || (!CONFIG_BOOL_VAL
       && CONFIG_TRISTATE_VAL))"
3  CONFIG_CHOICE_1_META "((CHOICE_1 && !CHOICE_1_MODULE) || (!CHOICE_1
       && CHOICE_1_MODULE))"
4  CONFIG_CHOICE_1_MODULE "!CONFIG_CHOICE_1 && CONFIG_MODULES &&
       CONFIG_CHOICE_1_META && !CONFIG_BOOL_VAL && !CONFIG_TRISTATE_VAL"
5  CONFIG_MODULES "(__FREE__0)"
6  CONFIG_TRISTATE_VAL "!CONFIG_TRISTATE_VAL_MODULE && CONFIG_CHOICE_1"
7  CONFIG_TRISTATE_VAL_MODULE "!CONFIG_TRISTATE_VAL && CONFIG_MODULES
       && CONFIG_CHOICE_1_MODULE"
8  UNDERTAKER_SET ALWAYS_ON "CONFIG_CHOICE_1_META" "CONFIG_MODULES"
9  UNDERTAKER_SET SCHEMA_VERSION 1.1
10 CONFIG_X86 ""
11 CONFIG_n
12 CONFIG_y
13 CONFIG_m
```

Figure 3.61: rsf2model's translation of Figure 3.60 on the preceding page (correct).

Figure 3.61 shows how rsf2model translates the KConfig model from 3.59 on the previous page into Model format. Lines 2 and 4 list the possible combinations of `BOOL_VAL` and `TRISTATE_VAL` if the choice was set to `y` or `m` respectively. Lines 3 and 8 specify that the choice has to be selected either as `y` or as `m`, which is also correct since the choice does not contain the `optional` attribute to allow the permanently deselection (cf. Section 2.1 on page 14). Thus, this translation is correct.

Satyr

```
1  c File Format Version: 2.0
2  c Generated by satyr
3  c Type info:
4  c c sym <symbolname> <typeid>
5  c with <typeid> being an integer out of:
6  c enum {S_BOOLEAN=1, S_TRISTATE=2, S_INT=3, S_HEX=4, S_STRING=5, S_OTHER=6}
7  c variable names:
8  c c var <variablename> <cnfvar>
9  c meta_value ALWAYS_ON CONFIG_MODULES
10 c sym BOOL_VAL 1
```

```
c sym CHOICE_0 2
c sym MODULES 1
c sym TRISTATE_VAL 2
c sym UNAME_RELEASE 5
c sym ____MODULES_MAGIC_INTERNAL_VAR____ 1
c var CONFIG_BOOL_VAL 6
c var CONFIG_CHOICE_0 1
c var CONFIG_CHOICE_0_MODULE 2
c var CONFIG_MODULES 16
c var CONFIG_TRISTATE_VAL 5
c var CONFIG_TRISTATE_VAL_MODULE 11
c var CONFIG_UNAME_RELEASE 20
c var CONFIG_____MODULES_MAGIC_INTERNAL_VAR____ 15
p cnf 29 75
3 -1 0
3 -2 0
-3 1 2 0
3 0
-4 1 0
-4 2 0
4 -1 -2 0
-4 0
-7 5 0
-7 -6 0
7 -5 6 0
-8 -5 0
-8 6 0
8 5 -6 0
9 -7 0
9 -8 0
-9 7 8 0
10 1 0
10 -9 0
-10 -1 9 0
10 0
12 1 0
12 11 0
-12 -1 -11 0
12 0
13 -1 0
13 -2 0
-13 1 2 0
13 0
-14 1 0
-14 2 0
14 -1 -2 0
-14 0
17 15 0
17 -16 0
-17 -15 16 0
18 16 0
18 -15 0
-18 -16 15 0
-19 17 0
-19 18 0
```

```
19 −17 −18 0
19 0
21 5 0
21 −1 0
−21 −5 1 0
21 0
22 11 0
22 −1 0
−22 −11 1 0
23 −22 0
23 −2 0
−23 22 2 0
23 0
−24 5 0
−24 11 0
24 −5 −11 0
−24 0
16 0
25 6 0
25 −1 0
−25 −6 1 0
−26 2 0
−26 1 0
26 −2 −1 0
27 −1 0
27 −2 0
−27 1 2 0
−28 26 0
−28 27 0
28 −26 −27 0
29 −25 0
29 −28 0
−29 25 28 0
29 0
```

Figure 3.62: Satyr's CNF translation of Figure 3.59 on page 88 (correct).

The model in Figure 3.62 facilitates the selection of either `CONFIG_CHOICE_0` or `CONFIG_CHOICE_0_MODULE`. If `CONFIG_CHOICE_0` was selected, exactly one of `CONFIG_TRISTATE_VAL` or `CONFIG_BOOL_VAL` must also be selected. `CONFIG_TRISTATE_VAL_MODULE` may only be selected if `CONFIG_CHOICE_0_MODULE` is also selected. In this case, `CONFIG_TRISTATE_VAL` and `CONFIG_BOOL_VAL` must not be selected. It is also possible to select `CONFIG_CHOICE_0_MODULE` without selecting `CONFIG_TRISTATE_VAL_MODULE`. Thus, the behaviour of the choice from Figure 3.59 on page 88 was translated correctly.

3.8.2 KConfig Reader

RSF

```
1  Item          MODULES        boolean
2  HasPrompts    MODULES        0
3  Default       MODULES        "'y'"          "y"
4  #startchoice
5  Choice        CHOICE_1       required       tristate
6  Prompt        CHOICE_1       "y"
7  HasPrompts    CHOICE_1       1
8  #choice value
9  ChoiceItem    TRISTATE_VAL   CHOICE_1
10 Item          TRISTATE_VAL   tristate
11 Depends       TRISTATE_VAL   "CHOICE_1"
12 Prompt        TRISTATE_VAL   "CHOICE_1"
13 HasPrompts    TRISTATE_VAL   1
14 #choice value
15 ChoiceItem    BOOL_VAL       CHOICE_1
16 Item          BOOL_VAL       boolean
17 Depends       BOOL_VAL       "<choice>='y' && CHOICE_1"
18 Prompt        BOOL_VAL       "<choice>='y' && CHOICE_1"
19 HasPrompts    BOOL_VAL       1
20 #choice value
21 #endchoice
```

Figure 3.63: KConfig Reader's RSF translation of Figure 3.59 on page 88 (correct).

Figure 3.63 contains the same information as Figure 3.60 plus some additional information under which conditions prompts are displayed. Thus, we categorized this translation also as correct.

Model & CNF

```
1 [error] (run-main-6) java.lang.Exception: error parsing <choice>='y' &&
    CHOICE_1 ''' expected but '<' found
2 java.lang.Exception: error parsing <choice>='y' && CHOICE_1 ''' expected
    but '<' found
3     at de.fosd.typechef.kconfig.RSFReader$ConstraintParser.parseExpr(
    RSFReader.scala:157)
```

Figure 3.64: Stacktrace while trying to translate Figure 3.63 into CNF and Model (error).

Figure 3.64 shows that KConfig Reader was not able to parse the `<choice>='y'` part from the constrains in line 17 and 18. Thus, KConfig Reader was not able to create correct Model or CNF representations from the KConfig model shown in Figure 3.59 on page 88.

3.8.3 KConfig Reader (XML)

Model

In Figure 3.65, lines 2 and 18 ensure that `BOOL_VAL` and `TRISTATE_VAL` are only selectable if `CHOICE_1` is permanently selected and line 25 ensures that `TRISTATE_VAL_MODULE` is only selectable when `CHOICE_1` is selected as module. Thus, this translation is correct.

```
1  #item BOOL_VAL
2  ((def(CHOICE_1)&!def(CHOICE_1_MODULE))|(def(CHOICE_1)&!def(
       CHOICE_1_MODULE)&(def(CHOICE_1)|def(CHOICE_1_MODULE)))|!def(
       BOOL_VAL))
3  ((def(CHOICE_1)&!def(CHOICE_1_MODULE))|(def(CHOICE_1)&!def(
       CHOICE_1_MODULE)&(def(CHOICE_1)|def(CHOICE_1_MODULE)))|!def(
       BOOL_VAL))
4  #item CHOICE_1
5  (def(CHOICE_1)|def(CHOICE_1_MODULE))
6  (!def(CHOICE_1_MODULE)|def(MODULES))
7  (!def(CHOICE_1)|!def(CHOICE_1_MODULE))
8  #item MODULES
9  def(MODULES)
10 #item TRISTATE_VAL
11 (!def(TRISTATE_VAL_MODULE)|def(MODULES))
12 (!def(TRISTATE_VAL)|!def(TRISTATE_VAL_MODULE))
13 (!def(TRISTATE_VAL)|def(CHOICE_1)|(!def(CHOICE_1)&!def(
       CHOICE_1_MODULE)))
14 (def(CHOICE_1)|def(CHOICE_1_MODULE)|def(MODULES)|!def(TRISTATE_VAL))
15 (def(CHOICE_1)|def(CHOICE_1_MODULE)|!def(MODULES)|!def(
       TRISTATE_VAL_MODULE))
16 (def(CHOICE_1)|def(CHOICE_1_MODULE)|!def(MODULES)|!def(TRISTATE_VAL)
       )
17 (def(CHOICE_1)|def(CHOICE_1_MODULE)|!def(TRISTATE_VAL_MODULE))
18 (!def(TRISTATE_VAL)|def(CHOICE_1))
19 #choice CHOICE_1
20 (!def(CHOICE_1_MODULE)|def(MODULES))
21 (!def(CHOICE_1)|!def(CHOICE_1_MODULE))
22 (def(BOOL_VAL)|def(TRISTATE_VAL)|!def(CHOICE_1))
23 (def(CHOICE_1)|def(CHOICE_1_MODULE)|!def(BOOL_VAL))
24 (def(CHOICE_1)|def(CHOICE_1_MODULE)|(!def(TRISTATE_VAL)&!def(
       TRISTATE_VAL_MODULE)))
25 (!def(TRISTATE_VAL_MODULE)|def(CHOICE_1_MODULE))
26 (!def(BOOL_VAL)|!def(TRISTATE_VAL))
27 (!def(BOOL_VAL)|!def(TRISTATE_VAL_MODULE))
28 (!def(TRISTATE_VAL)|!def(TRISTATE_VAL_MODULE))
```

Figure 3.65: KConfig Reader (XML)'s Model translation of Figure 3.59 on page 88 (correct).

CNF

In Figure 3.65 on the preceding page, lines 9 and 24 ensure that `BOOL_VAL` and `TRISTATE_VAL` are only selectable if `CHOICE_1` is permanently selected and line 34 ensures that `TRISTATE_VAL_MODULE` is only selectable if `CHOICE_1` is selected as module. Thus, this translation carries the same logical information as Figure 3.65 on the previous page and is therefore correct.

```
1  c 1 MODULES
2  c 4 CHOICE_1_MODULE
3  c 6 TRISTATE_VAL
4  c 2 CHOICE_1
5  c 5 TRISTATE_VAL_MODULE
6  c 3 BOOL_VAL
7  p cnf 6 30
8  1 0
9  2 -3 0
10 2 4 -3 0
11 -4 2 -3 0
12 -4 -3 0
13 2 -3 0
14 2 4 -3 0
15 -4 2 -3 0
16 -4 -3 0
17 -5 1 0
18 -6 -5 0
19 -6 2 -4 0
20 2 4 1 -6 0
21 2 4 -1 -5 0
22 2 4 -1 -6 0
23 2 4 -5 0
24 -6 2 0
25 2 4 0
26 -4 1 0
27 -2 -4 0
28 -4 1 0
29 -2 -4 0
30 3 6 -2 0
31 2 4 -3 0
32 2 4 -5 0
33 2 4 -6 0
34 -5 4 0
35 -3 -6 0
36 -3 -5 0
37 -6 -5 0
```

Figure 3.66: KConfig Reader (XML)'s CNF translation of Figure 3.59 on page 88 (correct).

3.8.4 LVAT

Figure 3.67 on the following page does not contain any variable to control the choice, thus this translation is incorrect.

```
c 1$ _X9
c 2$ _X5
c 3$ _X1_m
c 4$ _X10
c 5$ _X5_m
c 6$ _X7_m
c 7$ _X2
c 8$ _X1
c 9 BOOL_VAL
c 10$ _X6
c 11 MODULES_m
c 12 MODULES
c 13$ _X10_m
c 14$ _X3
c 15$ _X4_m
c 16$ _X7
c 17 TRISTATE_VAL_m
c 18 TRISTATE_VAL
c 19$ _X3_m
c 20$ _X4
c 21$ _X8_m
c 22$ _X6_m
c 23 BOOL_VAL_m
c 24$ _X9_m
c 25$ _X8
c 26$ _X2_m
p cnf 26 49
-8 0
-3 0
-8 12 11 0
-3 12 11 0
-8 -3 -12 11 0
-12 11 0
12 -11 0
7 0
-26 0
-7 12 0
-7 11 0
-14 0
-19 0
-14 -12 8 0
-14 -8 12 0
-14 -11 3 0
-14 -3 11 0
-20 0
-15 0
-2 0
-5 0
-2 18 17 0
-5 18 17 0
-2 -5 -18 17 0
18 -17 0
-18 17 12 0
-10 0
-22 0
-10 -18 2 0
-10 -2 18 0
-10 -17 5 0
-10 -5 17 0
-16 0
-6 0
-25 0
-21 0
-25 9 23 0
-21 9 23 0
-25 -21 -9 23 0
-9 23 0
9 -23 0
-1 0
-24 0
-1 -9 25 0
-1 -25 9 0
-1 -23 21 0
-1 -21 23 0
-4 0
-13 0
```

Figure 3.67: LVAT's CNF translation of Figure 3.59 on page 88 (error).

3.9 Boolean Choice with Tristate Config Options

In Section 2.4.3, we discussed how KConfig handles choices with config options of a different type than the surrounding choice and in Section 3.8 we analysed how the different analysis tools handle a `tristate` `choice`s containing config options of type `bool`. In this section, we analyse how `boolean` choices containing `tristate` config options are handled.

We used the model from Figure 3.68 for our analysis. This model allows to permanently select either `TRISTATE_VAL` or `BOOL_VAL`. Conceptually, it does not matter whether `TRISTATE_VAL` is of type `tristate` or `bool`, since it is nested inside a `choice` of type `bool`.

```
config MODULES
    def_bool y
    option modules

choice
    bool "Boolean Choice"

    config TRISTATE_VAL
        tristate "Tristate Value"
    config BOOL_VAL
        bool "Boolean Value"
endchoice
```

Figure 3.68: KConfig example for testing the translation of a boolean choice with nested tristate config options.

3.9.1 Undertaker

dumpconf

```
Choice          CHOICE_1        required        boolean
ChoiceItem      BOOL_VAL        CHOICE_1
ChoiceItem      TRISTATE_VAL    CHOICE_1
Default         MODULES         "y"             "y"
Depends         BOOL_VAL        "CHOICE_1"
Depends         TRISTATE_VAL    "CHOICE_1"
HasPrompts      BOOL_VAL        1
HasPrompts      MODULES         0
HasPrompts      TRISTATE_VAL    1
Item            BOOL_VAL        boolean
Item            MODULES         boolean
Item            TRISTATE_VAL    tristate
```

Figure 3.69: dumpconf's translation of Figure 3.68 (correct).

Figure 3.69 contains the same logical information as the related KConfig model. Thus, we categorized this translation as correct.

rsf2model

```
1  CONFIG_BOOL_VAL "CONFIG_CHOICE_1"
2  CONFIG_CHOICE_1 "((CONFIG_BOOL_VAL && !CONFIG_TRISTATE_VAL) ||
       (!CONFIG_BOOL_VAL && CONFIG_TRISTATE_VAL))"
3  CONFIG_MODULES "(__FREE__0)"
4  CONFIG_TRISTATE_VAL "!CONFIG_TRISTATE_VAL_MODULE && CONFIG_CHOICE_1"
5  CONFIG_TRISTATE_VAL_MODULE "!CONFIG_TRISTATE_VAL && CONFIG_MODULES"
6  UNDERTAKER_SET ALWAYS_ON "CONFIG_CHOICE_1" "CONFIG_MODULES"
7  UNDERTAKER_SET SCHEMA_VERSION 1.1
8  CONFIG_X86 ""
9  CONFIG_n
10 CONFIG_y
11 CONFIG_m
```

Figure 3.70: rsf2model's translation of Figure 3.69 on the preceding page (correct).

Figure 3.70 selects permanently the choice in line 6. Line 2 defines a `XOR` relation between `CONFIG_BOOL_VAL` and `CONFIG_TRISTATE_VAL` if the choice is selected. Further, lines 4 and 5 are defining a `XOR` relation between `CONFIG_TRISTATE_VAL` and `CONFIG_TRISTATE_VAL_MODULE`. Only valid configurations are accepted by the translated model. Thus, we categorized this translation as correct.

Satyr

Figure 3.71 on the following page demonstrates Satyr's translation of Figure 3.68 on the previous page. The constrains in lines 10 – 18 model a `XOR` relation between `CONFIG_BOOL_VAL` and `CONFIG_TRISTATE_VAL`. The constrains in lines 19 – 21 define an implication that if `CONFIG_CHOICE_0` is selected that also the `XOR` relation from lines 10 – 18 must be fulfilled. The error lies in line 24 and 25. Line 25 defines that the constraint from 24 must be fulfilled in all configurations. The constraint in line 24 defines an `OR` relation between the constraint modelled in lines 10 – 21 and `CONFIG_TRISTATE_VAL_MODULE`. Thus, it is possible to select `CONFIG_CHOICE_0` and `CONFIG_TRISTATE_VAL_MODULE` without selecting one of the nested boolean config options. Both selections are not correct:

- `CONFIG_TRISTATE_VAL_MODULE` must not be selectable, because `TRISTATE_VAL` cannot be set to `m` in the related KConfig model.
- Either one of `CONFIG_BOOL_VAL` and `CONFIG_TRISTATE_VAL` must be selected, since the containing choice is also permanently selected.

```
1  c var CONFIG_BOOL_VAL 3
2  c var CONFIG_CHOICE_0 1
3  c var CONFIG_MODULES 11
4  c var CONFIG_TRISTATE_VAL 2
5  c var CONFIG_TRISTATE_VAL_MODULE 8
6  c var CONFIG_UNAME_RELEASE 15
7  c var CONFIG_____MODULES_MAGIC_INTERNAL_VAR____ 10
8  p cnf 19 44
9  1 0
10 -4 2 0
11 -4 -3 0
12 4 -2 3 0
13 -5 -2 0
14 -5 3 0
15 5 2 -3 0
16 6 -4 0
17 6 -5 0
18 -6 4 5 0
19 7 1 0
20 7 -6 0
21 -7 -1 6 0
22 9 -7 0
23 9 -8 0
24 -9 7 8 0
25 9 0
26 12 10 0
27 12 -11 0
28 -12 -10 11 0
29 13 11 0
30 13 -10 0
31 -13 -11 10 0
32 -14 12 0
33 -14 13 0
34 14 -12 -13 0
35 14 0
36 16 2 0
37 16 -1 0
38 -16 -2 1 0
39 16 0
40 17 8 0
41 17 -1 0
42 -17 -8 1 0
43 17 0
44 -18 2 0
45 -18 8 0
46 18 -2 -8 0
47 -18 0
48 11 0
49 19 3 0
50 19 -1 0
51 -19 -3 1 0
52 19 0
```

Figure 3.71: Satyr's CNF translation of Figure 3.68 on page 96 (error in lines 24 and 25).

3.9.2 KConfig Reader

RSF

```
1  Item            MODULES         boolean
2  HasPrompts      MODULES         0
3  Default         MODULES         "'y'"           "y"
4  #startchoice
5  Choice          CHOICE_1        required        boolean
6  Prompt          CHOICE_1        "y"
7  HasPrompts      CHOICE_1        1
8  #choice value
9  ChoiceItem      TRISTATE_VAL    CHOICE_1
10 Item            TRISTATE_VAL    tristate
11 Depends         TRISTATE_VAL    "CHOICE_1"
12 Prompt          TRISTATE_VAL    "CHOICE_1"
13 HasPrompts      TRISTATE_VAL    1
14 #choice value
15 ChoiceItem      BOOL_VAL        CHOICE_1
16 Item            BOOL_VAL        boolean
17 Depends         BOOL_VAL        "CHOICE_1"
18 Prompt          BOOL_VAL        "CHOICE_1"
19 HasPrompts      BOOL_VAL        1
20 #choice value
21 #endchoice
```

Figure 3.72: KConfig Reader's RSF translation of Figure 3.68 on page 96 (correct).

We categorized Figure 3.72 as correct, because it contains the same logical information as Figure 3.69 on page 96.

Model

```
1  #item BOOL_VAL
2  #item CHOICE_1
3  def(CHOICE_1)
4  #item MODULES
5  def(MODULES)
6  #item TRISTATE_VAL
7  (!def(TRISTATE_VAL_MODULE)|def(MODULES))
8  (!def(TRISTATE_VAL)|!def(TRISTATE_VAL_MODULE))
9  #choice CHOICE_1
10 (def(TRISTATE_VAL)|def(TRISTATE_VAL_MODULE)|def(BOOL_VAL)|!def(
       CHOICE_1))
11 (!def(BOOL_VAL)|def(CHOICE_1))
12 (def(CHOICE_1)|(!def(TRISTATE_VAL)&!def(TRISTATE_VAL_MODULE)))
13 (!def(BOOL_VAL)|!def(TRISTATE_VAL))
```

Figure 3.73: KConfig Reader's Model translation of Figure 3.68 (error in lines 10 & 12).

Figure 3.73 on the preceding page contains an error in lines 10 and 12. Both lines facilitate the selection of `CHOICE_1` together with `TRISTATE_VAL_MODULE`.

CNF

```
1  c 5 BOOL_VAL
2  c 2 MODULES
3  c 4 CHOICE_1
4  c 3 TRISTATE_VAL
5  c 1 TRISTATE_VAL_MODULE
6  p cnf 5 9
7  -1 2 0
8  -3 -1 0
9  2 0
10 4 0
11 3 1 5 -4 0
12 -5 4 0
13 -1 4 0
14 -3 4 0
15 -5 -3 0
```

Figure 3.74: KConfig Reader's CNF translation of Figure 3.68 on page 96 (error).

The translated CNF formula shown in Figure 3.74 facilitates the selection of `TRISTATE_VAL_MODULE` and `CHOICE_1`, which is not a valid configuration. The translated formula needs a further constraint, that `TRISTATE_VAL_MODULE` can only be selected if `CHOICE_1` is also selected as `m`, but which is not possible.

3.9.3 KConfig Reader (XML)

Model

In Figure 3.75 on the next page, there is no constraint that forbids `TRISTATE_VAL_MODULE` to be set to true if `BOOL_VAL` is selected. Thus, this translation is incorrect.

```
#item BOOL_VAL
(!def(BOOL_VAL)|def(CHOICE_1))
(!def(BOOL_VAL)|def(CHOICE_1))
#item CHOICE_1
def(CHOICE_1)
#item MODULES
def(MODULES)
#item TRISTATE_VAL
(!def(TRISTATE_VAL_MODULE)|def(MODULES))
(!def(TRISTATE_VAL)|!def(TRISTATE_VAL_MODULE))
(def(MODULES)|!def(TRISTATE_VAL)|def(CHOICE_1))
(!def(MODULES)|!def(TRISTATE_VAL_MODULE)|def(CHOICE_1))
(!def(MODULES)|!def(TRISTATE_VAL)|def(CHOICE_1))
(!def(TRISTATE_VAL_MODULE)|def(CHOICE_1))
(!def(TRISTATE_VAL)|def(CHOICE_1))
#choice CHOICE_1
(def(BOOL_VAL)|def(TRISTATE_VAL)|!def(CHOICE_1))
(!def(BOOL_VAL)|def(CHOICE_1))
(def(CHOICE_1)|(!def(TRISTATE_VAL)&!def(TRISTATE_VAL_MODULE)))
(!def(BOOL_VAL)|!def(TRISTATE_VAL))
```

Figure 3.75: KConfig Reader (XML)'s Model translation of Figure 3.68 on page 96 (error).

CNF

Figure 3.76 contains the same logical information as Figure 3.75 and is therefore also incorrect.

```
c 4 CHOICE_1
c 5 BOOL_VAL
c 2 TRISTATE_VAL_MODULE
c 3 TRISTATE_VAL
c 1 MODULES
p cnf 5 16
1 0
-2 1 0
-3 -2 0
1 -3 4 0
-1 -2 4 0
-1 -3 4 0
-2 4 0
-3 4 0
-5 4 0
-5 4 0
4 0
5 3 -4 0
-5 4 0
-2 4 0
-3 4 0
-5 -3 0
```

Figure 3.76: KConfig Reader (XML)'s CNF translation of Figure 3.68 on page 96 (error).

3.9.4 LVAT

The translated model of Figure 3.77 allows to select `TRISTATE_VAL` as a module, as no constraint is added to prevent the Tristate selection of `TRISTATE_VAL`. Thus, the translation is incorrect.

```
c 1$ _X9
c 2$ _X5
c 3$ _X1_m
c 4$ _X10
c 5$ _X5_m
c 6$ _X7_m
c 7$ _X2
c 8$ _X1
c 9 BOOL_VAL
c 10$ _X6
c 11 MODULES_m
c 12 MODULES
c 13$ _X10_m
c 14$ _X3
c 15$ _X4_m
c 16$ _X7
c 17 TRISTATE_VAL_m
c 18 TRISTATE_VAL
c 19$ _X3_m
c 20$ _X4
c 21$ _X8_m
c 22$ _X6_m
c 23 BOOL_VAL_m
c 24$ _X9_m
c 25$ _X8
c 26$ _X2_m
p cnf 26 51
-8 0
-3 0
-8 12 11 0
-3 12 11 0
-8 -3 -12 11 0
-12 11 0
12 -11 0
7 0
-26 0
-7 12 0
-7 11 0
-14 0
-19 0
-14 -12 8 0
-14 -8 12 0
-14 -11 3 0
-14 -3 11 0
-20 0
-15 0
-2 0
-5 0
-2 18 17 0
-5 18 17 0
-2 -5 -18 17 0
18 -17 0
-18 17 12 0
-10 0
-22 0
-10 -18 2 0
-10 -2 18 0
-10 -17 5 0
-10 -5 17 0
-16 0
-6 0
-25 0
-21 0
-25 9 23 0
-21 9 23 0
-25 -21 -9 23 0
-9 23 0
9 -23 0
-1 0
-24 0
-1 -9 25 0
-1 -25 9 0
-1 -23 21 0
-1 -21 23 0
-4 0
-13 0
18 9 0
-18 -9 0
```

Figure 3.77: LVAT's CNF translation of Figure 3.68 on page 96 (error).

3.10 Structured Choices

In Section 2.4.4, we demonstrated that KConfig supports dependencies between nested config options of a choice.

3.10.1 Undertaker

dumpconf

```
1  Choice          CHOICE_1        required        boolean
2  ChoiceItem      SUB_VAL         CHOICE_1
3  ChoiceItem      VAL_1           CHOICE_1
4  ChoiceItem      VAL_2           CHOICE_1
5  Depends         SUB_VAL         "CHOICE_1 && VAL_2"
6  Depends         VAL_1           "CHOICE_1"
7  Depends         VAL_2           "CHOICE_1"
8  HasPrompts      SUB_VAL         1
9  HasPrompts      VAL_1           1
10 HasPrompts      VAL_2           1
11 Item            SUB_VAL         boolean
12 Item            VAL_1           boolean
13 Item            VAL_2           boolean
```

Figure 3.78: dumpconf's translation of Figure 2.24 on page 35 (error).

Figure 3.78 shows how dumpconf translates a structured choice. In line 5, the `Depends` constraint is calculated correctly. However, dumpconf reorders the elements of the KConfig model without storing the old information. As a consequence, it is not possible to distinguish between a "Structured Choice" (cf. Section 2.4.4) and a "Recursive Dependency inside a Choice" (cf. Section 2.4.5). Thus, we categorized this translation as incorrect.

rsf2model

Figure 3.79 on the following page demonstrates that rsf2model is not aware of "Structured Choices". Line 1 models a `XOR` relation between all nested elements of the choice. Thus, `CONFIG_SUB_VAL` becomes a dead feature, which is not correct.

```
1  CONFIG_CHOICE_1 "((CONFIG_SUB_VAL && !CONFIG_VAL_1 &&
       !CONFIG_VAL_2) || (!CONFIG_SUB_VAL && CONFIG_VAL_1 &&
       !CONFIG_VAL_2) || (!CONFIG_SUB_VAL && !CONFIG_VAL_1 &&
       CONFIG_VAL_2))"
2  CONFIG_SUB_VAL "CONFIG_VAL_2 && CONFIG_CHOICE_1"
3  CONFIG_VAL_1 "CONFIG_CHOICE_1"
4  CONFIG_VAL_2 "CONFIG_CHOICE_1"
5  UNDERTAKER_SET ALWAYS_ON "CONFIG_CHOICE_1"
6  UNDERTAKER_SET SCHEMA_VERSION 1.1
7  CONFIG_X86 ""
8  CONFIG_n
9  CONFIG_y
10 CONFIG_m
```

Figure 3.79: rsf2model's translation of Figure 3.78 on page 103 (error in line 1).

Satyr

Figure 3.80 on the following page shows that Satyr is able to handle "Structured Choices". Lines 8 – 21 specify that the choice is permanently selected and exactly one of its direct nested elements must be selected. Further, lines 31 – 37 model constraints to allow the selection of `CONFIG_SUB_VAL` if and only if `CONFIG_VAL_2` is also selected. Thus, we categorized this translation as correct.

```
c var CONFIG_CHOICE_0 1
c var CONFIG_SUB_VAL 12
c var CONFIG_UNAME_RELEASE 9
c var CONFIG_VAL_1 2
c var CONFIG_VAL_2 3
c var CONFIG____MODULES_MAGIC_INTERNAL_VAR____ 8
p cnf 14 30
1 0
-4 2 0
-4 -3 0
4 -2 3 0
-5 -2 0
-5 3 0
5 2 -3 0
6 -4 0
6 -5 0
-6 4 5 0
7 1 0
7 -6 0
-7 -1 6 0
7 0
-8 0
10 3 0
10 -1 0
-10 -3 1 0
10 0
11 2 0
11 -1 0
-11 -2 1 0
11 0
-13 1 0
-13 3 0
13 -1 -3 0
14 12 0
14 -13 0
-14 -12 13 0
14 0
```

Figure 3.80: Satyr's CNF translation of Figure 2.24 on page 35 (correct).

3.10.2 KConfig Reader

RSF

Contrary to dumpconf, KConfig Reader seems to take over the ordering from the KConfig files to the translated RSF files as shown in Figure 3.81 on the following page. Thus, we categorized the translation as correct.

```
1  #startchoice
2  Choice          CHOICE_1        required        boolean
3  Prompt          CHOICE_1        "y"
4  HasPrompts      CHOICE_1        1
5  #choice value
6  ChoiceItem      VAL_1           CHOICE_1
7  Item            VAL_1           boolean
8  Depends         VAL_1           "CHOICE_1"
9  Prompt          VAL_1           "CHOICE_1"
10 HasPrompts      VAL_1           1
11 #choice value
12 ChoiceItem      VAL_2           CHOICE_1
13 Item            VAL_2           boolean
14 Depends         VAL_2           "CHOICE_1"
15 Prompt          VAL_2           "CHOICE_1"
16 HasPrompts      VAL_2           1
17 #choice value
18 ChoiceItem      SUB_VAL         CHOICE_1
19 Item            SUB_VAL         boolean
20 Depends         SUB_VAL         "CHOICE_1 && VAL_2"
21 Prompt          SUB_VAL         "CHOICE_1 && VAL_2"
22 HasPrompts      SUB_VAL         1
23 #endchoice
```

Figure 3.81: KConfig Reader's RSF translation of Figure 2.24 on page 35 (correct).

Model

```
1  #item CHOICE_1
2  def(CHOICE_1)
3  #item SUB_VAL
4  (!def(SUB_VAL)|def(VAL_2))
5  (!def(SUB_VAL)|def(VAL_2))
6  #item VAL_1
7  #item VAL_2
8  #choice CHOICE_1
9  (def(SUB_VAL)|def(VAL_2)|def(VAL_1)|!def(CHOICE_1))
10 (!def(SUB_VAL)|def(CHOICE_1))
11 (!def(VAL_2)|def(CHOICE_1))
12 (!def(VAL_1)|def(CHOICE_1))
13 (!def(SUB_VAL)|!def(VAL_2))
14 (!def(SUB_VAL)|!def(VAL_1))
15 (!def(VAL_2)|!def(VAL_1))
```

Figure 3.82: KConfig Reader's Model translation of Figure 2.24 on page 35 (error in 13).

Figure 3.82 demonstrates that KConfig Reader cannot translate "Structured Choices" into valid Model files. Starting from line 9, KConfig Reader models constraints to force

that exactly one nested element must be selected if the surrounding choice is selected. Line 13 models a mutual exclusive constraint between `SUB_VAL` and `VAL_2`, which is not correct.

CNF

```
1  c 4 VAL_1
2  c 1 SUB_VAL
3  c 3 CHOICE_1
4  c 5 MODULES
5  c 2 VAL_2
6  p cnf 5 11
7  -1 2 0
8  -1 2 0
9  3 0
10 1 2 4 -3 0
11 -1 3 0
12 -2 3 0
13 -4 3 0
14 -1 -2 0
15 -1 -4 0
16 -2 -4 0
17 -5 0
```

Figure 3.83: KConfig Reader's CNF translation of Figure 2.24 on page 35 (error in line 14).

Also the CNF formula created by KConfig Reader contains the same error as the related Model file. Line 14 of Figure 3.83 represents a mutual exclusive constraint between `SUB_VAL` and `VAL_2`, which is not correct.

3.10.3 KConfig Reader (XML)

Model

Starting with line 13 in Figure 3.84 on the next page, `SUB_VAL` is not part of the XOR constraints of the nested choice items. Line 4 ensures the dependency from `SUB_VAL` on `VAL_2`. Thus, this translation is correct.

```
#item CHOICE_1
def(CHOICE_1)
#item SUB_VAL
(!def(SUB_VAL)|(def(CHOICE_1)&def(VAL_2)))
(!def(SUB_VAL)|(def(CHOICE_1)&def(VAL_2)))
#item VAL_1
(!def(VAL_1)|def(CHOICE_1))
(!def(VAL_1)|def(CHOICE_1))
#item VAL_2
(!def(VAL_2)|def(CHOICE_1))
(!def(VAL_2)|def(CHOICE_1))
#choice CHOICE_1
(def(VAL_2)|def(VAL_1)|!def(CHOICE_1))
(!def(VAL_2)|def(CHOICE_1))
(!def(VAL_1)|def(CHOICE_1))
(!def(VAL_2)|!def(VAL_1))
```

Figure 3.84: KConfig Reader (XML)'s Model translation of Figure 2.24 on page 35 (correct).

CNF

Figure 3.85 contains the same logical information as Figure 3.84. Thus, this translations is also correct.

```
c 3 CHOICE_1
c 1 VAL_2
c 2 SUB_VAL
c 5 MODULES
c 4 VAL_1
p cnf 5 14
1 -2 0
3 -2 0
1 -2 0
3 -2 0
-1 3 0
-1 3 0
-4 3 0
-4 3 0
3 0
1 4 -3 0
-1 3 0
-4 3 0
-1 -4 0
-5 0
```

Figure 3.85: KConfig Reader (XML)'s CNF translation of Figure 2.24 on page 35 (correct).

3.10.4 LVAT

In Figure 3.86, `VAL_1` and `VAL_2` are connected via an xor (cf. lines 74 and 75) and `SUB_VAL` depends on `VAL_2` via `_X8`. Thus, this translation is correct.

```
1  c 1$ _X9
2  c 2$ _X5
3  c 3$ _X1_m
4  c 4 VAL_2
5  c 5$ _X5_m
6  c 6$ _X7_m
7  c 7 SUB_VAL_m
8  c 8 VAL_2_m
9  c 9 VAL_1
10 c 10$ _X2
11 c 11$ _X1
12 c 12$ _X6
13 c 13 SUB_VAL
14 c 14$ _X3
15 c 15$ _X4_m
16 c 16 VAL_1_m
17 c 17$ _X7
18 c 18$ _X3_m
19 c 19$ _X4
20 c 20$ _X8_m
21 c 21$ _X6_m
22 c 22$ _X9_m
23 c 23$ _X8
24 c 24$ _X2_m
25 p cnf 24 50
26 -11 0
27 -3 0
28 -11 9 16 0
29 -3 9 16 0
30 -11 -3 -9 16 0
31 -9 16 0
32 9 -16 0
33 -10 0
34 -24 0
35 -10 -9 11 0
36 -10 -11 9 0
37 -10 -16 3 0
38 -10 -3 16 0
39 -14 0
40 -18 0
41 -19 0
42 -15 0
43 -19 4 8 0
44 -15 4 8 0
45 -19 -15 -4 8 0
46 -4 8 0
47 4 -8 0
48 -2 0
49 -5 0
50 -2 -4 19 0
51 -2 -19 4 0
52 -2 -8 15 0
53 -2 -15 8 0
54 -12 0
55 -21 0
56 -17 0
57 -6 0
58 -17 13 7 0
59 -6 13 7 0
60 -17 -6 -13 7 0
61 -13 7 0
62 13 -7 0
63 4 -8 0
64 -23 -4 0
65 -23 -8 0
66 4 8 23 0
67 -20 0
68 -23 -13 17 0
69 -23 -17 13 0
70 -23 -7 6 0
71 -23 -6 7 0
72 -1 0
73 -22 0
74 9 4 0
75 -9 -4 0
```

Figure 3.86: LVAT's CNF translation of Figure 2.24 on page 35 (correct).

3.11 Recursive Dependency inside a Choice

In Section 2.4.5, we discussed the behaviour of recursive dependencies in KConfig. Even if KConfig is able to detect such recursive dependencies during the execution, we also analysed how analysis tools handle such recursive dependencies.

For the analysis, we shortened the example from Section 2.4.5 and used the model shown in Figure 3.87. `SUB_VAL` will not be selectable, since it is dependant of `VAL_1` but written below `VAL_2` inside the choice.

```
1  choice
2      bool "A structured Choice"
3
4      config VAL_1
5          bool "Value 1"
6      config VAL_2
7          bool "Value 2"
8
9      config SUB_VAL
10         bool "A sub value"
11         depends on VAL_1
12 endchoice
```

Figure 3.87: Modification of Figure 2.27 on page 38: `SUB_VAL` will not be selectable.

3.11.1 Undertaker

dumpconf

```
1  Choice          CHOICE_1        required        boolean
2  ChoiceItem      SUB_VAL         CHOICE_1
3  ChoiceItem      VAL_1           CHOICE_1
4  ChoiceItem      VAL_2           CHOICE_1
5  Depends         SUB_VAL         "CHOICE_1 && VAL_1"
6  Depends         VAL_1           "CHOICE_1"
7  Depends         VAL_2           "CHOICE_1"
8  HasPrompts      SUB_VAL         1
9  HasPrompts      VAL_1           1
10 HasPrompts      VAL_2           1
11 Item            SUB_VAL         boolean
12 Item            VAL_1           boolean
13 Item            VAL_2           boolean
```

Figure 3.88: dumpconf's translation of Figure 3.87 (error).

dumpconf and rsf2model display a warning while translating the model from Figure 3.87 (cf. Figure 3.89 on the following page). However, it still produces a complete RSF file. Figure 3.88 contains exactly the same information in the same structure as Figure

3.78 with the exception of the changed constrained in line 5. Thus, we categorized this translation as incorrect, because of the absence of the correct ordering of the nested elements.

```
1 Generating Format 1.0 (RSF) models
2 Calculating RSF model for x86
3 using arch x86
4 arch/x86/Kconfig:1:error: recursive dependency detected!
5 arch/x86/Kconfig:1: choice <choice> contains symbol SUB_VAL
6 arch/x86/Kconfig:9: symbol SUB_VAL depends on VAL_1
7 arch/x86/Kconfig:4: symbol VAL_1 is part of choice <choice>
```

Figure 3.89: Displayed warning of dumpconf and rsf2model while translating Figure 3.87.

rsf2model

```
1 CONFIG_CHOICE_1 "((CONFIG_SUB_VAL && !CONFIG_VAL_1 &&
    !CONFIG_VAL_2) || (!CONFIG_SUB_VAL && CONFIG_VAL_1 &&
    !CONFIG_VAL_2) || (!CONFIG_SUB_VAL && !CONFIG_VAL_1 &&
    CONFIG_VAL_2))"
2 CONFIG_SUB_VAL "CONFIG_VAL_1 && CONFIG_CHOICE_1"
3 CONFIG_VAL_1 "CONFIG_CHOICE_1"
4 CONFIG_VAL_2 "CONFIG_CHOICE_1"
5 UNDERTAKER_SET ALWAYS_ON "CONFIG_CHOICE_1"
6 UNDERTAKER_SET SCHEMA_VERSION 1.1
7 CONFIG_X86 ""
8 CONFIG_n
9 CONFIG_y
10 CONFIG_m
```

Figure 3.90: rsf2model's translation of Figure 3.88 (correct).

Figure 3.90 has the same structure as Figure 3.79 on page 104. The translated model also avoids the selection of `SUB_VAL`, which is correct due to the contained recursive dependency.

Satyr

Figure 3.91 on the next page demonstrates how Satyr handles recursive dependencies. The translated formula forces to select either `CONFIG_VAL_1` or `CONFIG_VAL_2` and avoids the selection of `CONFIG_SUB_VAL`. Thus, the translated CNF formula is correct.

```
c var CONFIG_CHOICE_0 1
c var CONFIG_SUB_VAL 5
c var CONFIG_UNAME_RELEASE 15
c var CONFIG_VAL_1 2
c var CONFIG_VAL_2 3
c var CONFIG_____MODULES_MAGIC_INTERNAL_VAR____ 14
p cnf 19 45
1 0
-4 2 0
-4 -3 0
4 -2 3 0
-6 4 0
-6 -5 0
6 -4 5 0
-7 -2 0
-7 3 0
7 2 -3 0
-8 7 0
-8 -5 0
8 -7 5 0
9 -6 0
9 -8 0
-9 6 8 0
-10 -2 0
-10 -3 0
10 2 3 0
-11 10 0
-11 5 0
11 -10 -5 0
12 -9 0
12 -11 0
-12 9 11 0
13 1 0
13 -12 0
-13 -1 12 0
13 0
-14 0
16 3 0
16 -1 0
-16 -3 1 0
16 0
17 2 0
17 -1 0
-17 -2 1 0
17 0
-18 1 0
-18 2 0
18 -1 -2 0
19 5 0
19 -18 0
-19 -5 18 0
19 0
```

Figure 3.91: Satyr's CNF translation of Figure 3.87 on the preceding page (correct).

Also Satyr displays a warning while translating the model from Figure 3.87 into the correct model shown in Figure 3.91. This warning is shown in Figure 3.92.

```
Generating Format 2.0 (CNF) models
Calculating CNF model for x86
using arch x86
arch/x86/Kconfig:1:error: recursive dependency detected!
arch/x86/Kconfig:1: choice <choice> contains symbol SUB_VAL
arch/x86/Kconfig:9: symbol SUB_VAL depends on VAL_1
arch/x86/Kconfig:4: symbol VAL_1 is part of choice <choice>
Calling rsf2cnf for arch x86...
```

Figure 3.92: Displayed warning of Satyr while creating Figure 3.91 on page 112.

3.11.2 KConfig Reader

RSF

```
#startchoice
Choice          CHOICE_1        required        boolean
Prompt          CHOICE_1        "y"
HasPrompts      CHOICE_1        1
#choice value
ChoiceItem      VAL_1           CHOICE_1
Item            VAL_1           boolean
Depends         VAL_1           "CHOICE_1"
Prompt          VAL_1           "CHOICE_1"
HasPrompts      VAL_1           1
#choice value
ChoiceItem      VAL_2           CHOICE_1
Item            VAL_2           boolean
Depends         VAL_2           "CHOICE_1"
Prompt          VAL_2           "CHOICE_1"
HasPrompts      VAL_2           1
#choice value
ChoiceItem      SUB_VAL         CHOICE_1
Item            SUB_VAL         boolean
Depends         SUB_VAL         "CHOICE_1 && VAL_1"
Prompt          SUB_VAL         "CHOICE_1 && VAL_1"
HasPrompts      SUB_VAL         1
#choice value
#endchoice
```

Figure 3.93: KConfig Reader's RSF translation of Figure 3.87 on page 110 (correct).

KConfig Reader creates a RSF file which looks very similar to the RSF file of Figure 3.88 on page 110. However, KConfig Reader keeps the ordering of the config options, which makes it possible to detect recursive dependencies during a downstream analysis.

Thus, we categorized this translation as correct. It also produces a warning like the other tools (cf. Figure 3.94).

```
dumping model
setting archusing arch x86
arch/x86/Kconfig:1:error: recursive dependency detected!
arch/x86/Kconfig:1: choice <choice> contains symbol SUB_VAL
arch/x86/Kconfig:9: symbol SUB_VAL depends on VAL_1
arch/x86/Kconfig:4: symbol VAL_1 is part of choice <choice>
reading model
getting constraints
checking combined constraint
writing model
writing dimacs
done.
```

Figure 3.94: Displayed warning of KConfig Reader while translating Figure 3.87 on page 110.

Model

```
#item CHOICE_1
def(CHOICE_1)
#item SUB_VAL
(!def(SUB_VAL)|def(VAL_1))
(!def(SUB_VAL)|def(VAL_1))
#item VAL_1
#item VAL_2
#choice CHOICE_1
(def(SUB_VAL)|def(VAL_2)|def(VAL_1)|!def(CHOICE_1))
(!def(SUB_VAL)|def(CHOICE_1))
(!def(VAL_2)|def(CHOICE_1))
(!def(VAL_1)|def(CHOICE_1))
(!def(SUB_VAL)|!def(VAL_2))
(!def(SUB_VAL)|!def(VAL_1))
(!def(VAL_2)|!def(VAL_1))
```

Figure 3.95: KConfig Reader's Model translation of Figure 3.87 on page 110 (correct).

Figure 3.95 has exactly the same structure as Figure 3.82 on page 106 including the mutual exclusive constraint between `SUB_VAL` and `VAL_1`. This time is the constraint correct, because of the modelled recursive dependency inside the containing choice. Thus, this translation is categorized as correct.

CNF

Figure 3.96 demonstrates how KConfig Reader translates a recursive dependency into CNF formula. This translation has the same structure as Figure 3.83 on page 107. The recursive dependency is translated correctly as a mutual exclusive constraint in line 14. Thus, we categorized also this translation as correct.

```
c 1 SUB_VAL
c 2 VAL_1
c 4 VAL_2
c 3 CHOICE_1
c 5 MODULES
p cnf 5 11
-1 2 0
-1 2 0
3 0
1 4 2 -3 0
-1 3 0
-4 3 0
-2 3 0
-1 -4 0
-1 -2 0
-4 -2 0
-5 0
```

Figure 3.96: KConfig Reader's CNF translation of Figure 3.87 on page 110 (correct).

3.11.3 KConfig Reader (XML)

Model

Figure 3.97 only differs in line 4 and 5 from KConfig Reader's Model translation (cf. Section 3.11.2). KConfig Reader (XML) adds also the containing choice to the implication constraint, which is also contained in line 14. Both translations do not differ logically from each other. Thus, this translation is correct.

```
#item CHOICE_1
def(CHOICE_1)
#item SUB_VAL
(!def(SUB_VAL)|(def(CHOICE_1)&def(VAL_1)))
(!def(SUB_VAL)|(def(CHOICE_1)&def(VAL_1)))
#item VAL_1
(!def(VAL_1)|def(CHOICE_1))
(!def(VAL_1)|def(CHOICE_1))
#item VAL_2
(!def(VAL_2)|def(CHOICE_1))
(!def(VAL_2)|def(CHOICE_1))
#choice CHOICE_1
(def(SUB_VAL)|def(VAL_2)|def(VAL_1)|!def(CHOICE_1))
(!def(SUB_VAL)|def(CHOICE_1))
(!def(VAL_2)|def(CHOICE_1))
(!def(VAL_1)|def(CHOICE_1))
(!def(SUB_VAL)|!def(VAL_2))
(!def(SUB_VAL)|!def(VAL_1))
(!def(VAL_2)|!def(VAL_1))
```

Figure 3.97: KConfig Reader (XML)'s Model translation of Figure 3.87 (correct).

CNF

Figure 3.98 contains the same logical information as KConfig Reader's CNF translation in Figure 3.96 and is therefore correct.

```
1  c 1 CHOICE_1
2  c 4 SUB_VAL
3  c 5 MODULES
4  c 2 VAL_2
5  c 3 VAL_1
6  p cnf 5 17
7  1 0
8  -2 1 0
9  -2 1 0
10 -3 1 0
11 -3 1 0
12 3 -4 0
13 -4 1 0
14 3 -4 0
15 -4 1 0
16 4 2 3 -1 0
17 -4 1 0
18 -2 1 0
19 -3 1 0
20 -4 -2 0
21 -4 -3 0
22 -2 -3 0
23 -5 0
```

Figure 3.98: KConfig Reader (XML)'s CNF translation of Figure 3.87 (correct).

3.11.4 LVAT

In Figure 3.99 on the next page, `VAL_1`, `VAL_2`, and `SUB_VAL` are all connected via an XOR relation (cf. lines 74 – 77) and `SUB_VAL` depends on `VAL_2` via `_X8`. As a result, `SUB_VAL` is not selectable and the translation is correct.

```
c 1$ _X9
c 2$ _X5
c 3$ _X1_m
c 4 VAL_2
c 5$ _X5_m
c 6$ _X7_m
c 7 SUB_VAL_m
c 8 VAL_2_m
c 9 VAL_1
c 10$ _X2
c 11$ _X1
c 12$ _X6
c 13 SUB_VAL
c 14$ _X3
c 15$ _X4_m
c 16 VAL_1_m
c 17$ _X7
c 18$ _X3_m
c 19$ _X4
c 20$ _X8_m
c 21$ _X6_m
c 22$ _X9_m
c 23$ _X8
c 24$ _X2_m
p cnf 24 52
-11 0
-3 0
-11 9 16 0
-3 9 16 0
-11 -3 -9 16 0
-9 16 0
9 -16 0
-10 0
-24 0
-10 -9 11 0
-10 -11 9 0
-10 -16 3 0
-10 -3 16 0
-14 0
-18 0
-19 0
-15 0
-19 4 8 0
-15 4 8 0
-19 -15 -4 8 0
-4 8 0
4 -8 0
-2 0
-5 0
-2 -4 19 0
-2 -19 4 0
-2 -8 15 0
-2 -15 8 0
-12 0
-21 0
-17 0
-6 0
-17 13 7 0
-6 13 7 0
-17 -6 -13 7 0
-13 7 0
13 -7 0
9 -16 0
-23 -9 0
-23 -16 0
9 16 23 0
-20 0
-23 -13 17 0
-23 -17 13 0
-23 -7 6 0
-23 -6 7 0
-1 0
-22 0
9 4 13 0
-9 -4 0
-9 -13 0
-4 -13 0
```

Figure 3.99: LVAT's CNF translation of Figure 3.87 on page 110 (correct).

3.12 Empty Choices

In Section 2.4.6, we showed that it is allowed to model empty choices, even if exactly one nested element must be selected. KConfig is able to handle such situations and will not force the selection of a not existing config option to keep the configuration valid.

For testing the translation, we adapted the model from Figure 2.30 on page 41 and replaced the `NOT_EXISTING` config option by a constant config option which was set to `false`. The modified model is presented in Figure 3.100.

```
config CONST_FALSE
    def_bool n

choice
    bool "An Empty Choice"

    config VAL_1
        bool "Value 1"
        depends on CONST_FALSE
    config VAL_2
        bool "Value 2"
        depends on CONST_FALSE
endchoice
```

Figure 3.100: Modification of Figure 2.30 on page 41: The choice still does not contain any selectable nested config options.

3.12.1 Undertaker

dumpconf

```
Choice          CHOICE_1        required        boolean
ChoiceItem      VAL_1           CHOICE_1
ChoiceItem      VAL_2           CHOICE_1
Default         CONST_FALSE     "n"             "y"
Depends         VAL_1           "CHOICE_1 && CONST_FALSE"
Depends         VAL_2           "CHOICE_1 && CONST_FALSE"
HasPrompts      CONST_FALSE     0
HasPrompts      VAL_1           1
HasPrompts      VAL_2           1
Item            CONST_FALSE     boolean
Item            VAL_1           boolean
Item            VAL_2           boolean
```

Figure 3.101: dumpconf's translation of Figure 3.100 (correct).

dumpconf translates the information of the KConfig file correctly into RSF, which is shown in Figure 3.101. The existing `depends on` constraints are extended and contain

an additional dependency to the containing choice. Thus, we categorized the translation as correct.

rsf2model

```
1 CONFIG_CHOICE_1 "((CONFIG_VAL_1 && !CONFIG_VAL_2) || (!CONFIG_VAL_1
    && CONFIG_VAL_2))"
2 CONFIG_CONST_FALSE "(CONFIG_n)"
3 CONFIG_VAL_1 "CONFIG_CONST_FALSE && CONFIG_CHOICE_1"
4 CONFIG_VAL_2 "CONFIG_CONST_FALSE && CONFIG_CHOICE_1"
5 UNDERTAKER_SET ALWAYS_ON "CONFIG_CHOICE_1"
6 UNDERTAKER_SET SCHEMA_VERSION 1.1
7 CONFIG_X86 ""
8 CONFIG_n
9 CONFIG_y
10 CONFIG_m
```

Figure 3.102: rsf2model's translation of Figure 3.101 (error in line 1).

Figure 3.102 on page 119 demonstrates that rsf2model is not able to handle empty Choices. The constraint in line 1 forces the selection of either `CONFIG_VAL_1` or `CONFIG_VAL_2` if `CONFIG_CHOICE_1` is selected. Further, `CONFIG_CHOICE_1` is permanently selected and `CONFIG_CONST_FALSE` is permanently deselected. Together with the constraints from line 3 and 4, the model becomes unsatisfiable. Thus, the translation is not correct.

Satyr

Figure 3.103 on the next page demonstrates that also Satyr is not able to handle empty Choices. The constraint in line 23 forces the permanent selection of the `choice`. The constraints in lines 24 – 36 force the selection of either `CONFIG_VAL_1` or `CONFIG_VAL_2` if `CONFIG_CHOICE_0` is selected, which makes the model unsatisfiable. Thus, the translation is not correct.

```
1  c File Format Version: 2.0
2  c Generated by satyr
3  c Type info:
4  c c sym <symbolname> <typeid>
5  c with <typeid> being an integer out of:
6  c enum {S_BOOLEAN=1, S_TRISTATE=2, S_INT=3, S_HEX=4, S_STRING=5, S_OTHER=6}
7  c variable names:
8  c c var <variablename> <cnfvar>
9  c meta_value ALWAYS_ON CONFIG_CHOICE_0
10 c sym CHOICE_0 1
11 c sym CONST_FALSE 1
12 c sym UNAME_RELEASE 5
13 c sym VAL_1 1
14 c sym VAL_2 1
15 c sym ____MODULES_MAGIC_INTERNAL_VAR____ 1
16 c var CONFIG_CHOICE_0 1
17 c var CONFIG_CONST_FALSE 10
18 c var CONFIG_UNAME_RELEASE 9
19 c var CONFIG_VAL_1 2
20 c var CONFIG_VAL_2 3
21 c var CONFIG_____MODULES_MAGIC_INTERNAL_VAR____ 8
22 p cnf 14 30
23 1 0
24 -4 2 0
25 -4 -3 0
26 4 -2 3 0
27 -5 -2 0
28 -5 3 0
29 5 2 -3 0
30 6 -4 0
31 6 -5 0
32 -6 4 5 0
33 7 1 0
34 7 -6 0
35 -7 -1 6 0
36 7 0
37 -8 0
38 -11 1 0
39 -11 10 0
40 11 -1 -10 0
41 12 3 0
42 12 -11 0
43 -12 -3 11 0
44 12 0
45 -13 1 0
46 -13 10 0
47 13 -1 -10 0
48 14 2 0
49 14 -13 0
50 -14 -2 13 0
51 14 0
52 -10 0
```

Figure 3.103: Satyr's CNF translation of Figure 3.100 (error in lines 23 – 36).

3.12.2 KConfig Reader

RSF

```
Item            CONST_FALSE    boolean
HasPrompts      CONST_FALSE    0
Default         CONST_FALSE    "'n'"          "y"
#startchoice
Choice          CHOICE_1       required       boolean
Prompt          CHOICE_1       "y"
HasPrompts      CHOICE_1       1
#choice value
ChoiceItem      VAL_1          CHOICE_1
Item            VAL_1          boolean
Depends         VAL_1          "CHOICE_1 && CONST_FALSE"
Prompt          VAL_1          "CHOICE_1 && CONST_FALSE"
HasPrompts      VAL_1          1
#choice value
ChoiceItem      VAL_2          CHOICE_1
Item            VAL_2          boolean
Depends         VAL_2          "CHOICE_1 && CONST_FALSE"
Prompt          VAL_2          "CHOICE_1 && CONST_FALSE"
HasPrompts      VAL_2          1
#choice value
#endchoice
```

Figure 3.104: KConfig Reader's RSF translation of Figure 3.100 (correct).

Figure 3.104 contains the same logical information as Figure 3.101 on the preceding page. Thus, also KConfig Reader's translation into RSF is correct.

Model & CNF

```
dumping model
setting archusing arch x86
reading model
getting constraints
checking combined constraint
checking each constraint
[error] (run-main-1) java.lang.AssertionError: assertion failed: extracted
    model is not satisfiable
java.lang.AssertionError: assertion failed: extracted model is not
    satisfiable
     at scala.Predef$.assert(Predef.scala:179)
```

Figure 3.105: Stacktrace while trying to translate Figure 3.104 into CNF and Model (error).

KConfig Reader is not able to translate an empty Choice into boolean formula. Instead it produces a stack trace showing that the given model is not satisfiable (cf. Figure 3.105 on page 121). Thus, the translation into Model and CNF files are not correct.

3.12.3 KConfig Reader (XML)

Model & CNF

KConfig Reader (XML) aborts the translation of Figure 3.100 on page 118 with the same stacktrace as KConfig Reader (cf. Section 3.12.2).

3.12.4 LVAT

In Figure 3.106 on the following page `VAL_1` and `VAL_2` are connected via an xor (cf. lines 85 and 86) and both depend on `CONST_FALSE` (via `_X6` and `_X9` respectively). Thus, the model is not solvable and not correctly translated.

```
c 1$ _X9
c 2$ _X5
c 3$ _X1_m
c 4 VAL_2
c 5$ _X10
c 6$ _X5_m
c 7$ _X7_m
c 8 VAL_2_m
c 9 VAL_1
c 10$ _X2
c 11$ _X1
c 12$ _X6
c 13 CONST_FALSE
c 14$ _X10_m
c 15$ _X3
c 16$ _X4_m
c 17 VAL_1_m
c 18$ _X7
c 19$ _X3_m
c 20$ _X4
c 21$ _X8_m
c 22$ _X6_m
c 23$ _X9_m
c 24$ _X8
c 25 CONST_FALSE_m
c 26$ _X2_m
p cnf 26 59
−11 0
−3 0
−11 13 25 0
−3 13 25 0
−11 −3 −13 25 0
−13 25 0
13 −25 0
10 0
−26 0
−10 −13 11 0
−10 −11 13 0
−10 −25 3 0
−10 −3 25 0
−15 0
−19 0
−15 −13 11 0
−15 −11 13 0
−15 −25 3 0
−15 −3 25 0
−20 0
−16 0
−2 0
−6 0
−2 9 17 0
−6 9 17 0
−2 −6 −9 17 0
−9 17 0
9 −17 0
13 −25 0
−12 −13 0
−12 −25 0
13 25 12 0
−22 0
−12 −9 2 0
−12 −2 9 0
−12 −17 6 0
−12 −6 17 0
−18 0
−7 0
−24 0
−21 0
−24 4 8 0
−21 4 8 0
−24 −21 −4 8 0
−4 8 0
4 −8 0
13 −25 0
−1 −13 0
−1 −25 0
13 25 1 0
−23 0
−1 −4 24 0
−1 −24 4 0
−1 −8 21 0
−1 −21 8 0
−5 0
−14 0
9 4 0
−9 −4 0
```

Figure 3.106: LVAT's CNF translation of Figure 3.100 on page 118 (error).

3.13 Choices Without a Prompt

In Section 2.4.7, we showed that even mandatory choices without a prompt are not selectable. Even if a default value was specified, the resulting configuration will not include any selected nested elements. For the analysis, we used the model from Figure 2.33 on page 42.

3.13.1 Undertaker

dumpconf

```
Choice          CHOICE_1        required        boolean
ChoiceItem      BOOL_VAL1       CHOICE_1
ChoiceItem      BOOL_VAL2       CHOICE_1
Depends         BOOL_VAL1       "CHOICE_1"
Depends         BOOL_VAL2       "CHOICE_1"
HasPrompts      BOOL_VAL1       1
HasPrompts      BOOL_VAL2       1
Item            BOOL_VAL1       boolean
Item            BOOL_VAL2       boolean
```

Figure 3.107: dumpconf's translation of Figure 2.33 on page 42 (error).

Figure 3.107 displays the translated RSF file of dumpconf. It does not contain any information whether the choice has a prompt or not. This is also the case if the choice has a prompt. Thus, the produced RSF file does not differentiate between a choice with and without a prompt, which is not correct. However, Undertaker displays a warning while translating KConfig files (cf. Figure 3.108).

```
Generating Format 1.0 (RSF) models
Calculating RSF model for x86
using arch x86
arch/x86/Kconfig:1:warning: choice must have a prompt
suh@ubuntu:/data/src/Linux-Releases/UndertakerTest$ /tools/installed/
    Undertaker-1.6.1/bin/undertaker-kconfigdump -c
Generating Format 2.0 (CNF) models
Calculating CNF model for x86
using arch x86
arch/x86/Kconfig:1:warning: choice must have a prompt
Calling rsf2cnf for arch x86...
```

Figure 3.108: Displayed warning while translating Figure 2.33 on page 42 into RSF and Model.

rsf2model

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

Figure 3.109: rsf2model's translation of Figure 3.107 on page 124 (error in line 4).

Figure 3.109 on the next page shows the translated model of rsf2model. The translated model forces the permanent selection of the choice (cf. line 4), which is not correct, since the choice has no prompt.

Satyr

Figure 3.110 shows how Satyr translates the model from Figure 2.33 on page 42. It does not contain a constraint for the permanent selection of the choice, which is usually created for mandatory choices. However, it also does not contain a constraint for the permanent deselection of the choice. Thus, we categorized this translation as not correct.

```
c File Format Version: 2.0
c Generated by satyr
c Type info:
c c sym <symbolname> <typeid>
c with <typeid> being an integer out of:
c enum {S_BOOLEAN=1, S_TRISTATE=2, S_INT=3, S_HEX=4, S_STRING=5, S_OTHER=6}
c variable names:
c c var <variablename> <cnfvar>
c sym BOOL_VAL1 1
c sym BOOL_VAL2 1
c sym CHOICE_0 1
c sym UNAME_RELEASE 5
c sym ____MODULES_MAGIC_INTERNAL_VAR____ 1
c var CONFIG_BOOL_VAL1 2
c var CONFIG_BOOL_VAL2 6
c var CONFIG_CHOICE_0 1
c var CONFIG_UNAME_RELEASE 12
c var CONFIG_____MODULES_MAGIC_INTERNAL_VAR____ 11
p cnf 14 32
3 1 0
3 -2 0
-3 -1 2 0
4 2 0
4 -1 0
-4 -2 1 0
-5 3 0
-5 4 0
5 -3 -4 0
5 0
-7 2 0
-7 -6 0
7 -2 6 0
-8 -2 0
-8 6 0
8 2 -6 0
9 -7 0
9 -8 0
-9 7 8 0
10 1 0
10 -9 0
-10 -1 9 0
10 0
-11 0
13 6 0
13 -1 0
-13 -6 1 0
13 0
14 2 0
14 -1 0
-14 -2 1 0
14 0
```

Figure 3.110: Satyr's CNF translation of Figure 2.33 on page 42 (error).

3.13.2 KConfig Reader

RSF

```
1  #startchoice
2  Choice          CHOICE_1        required        boolean
3  HasPrompts      CHOICE_1        0
4  Default         CHOICE_1        "BOOL_VAL1"     "y"
5  #choice value
6  ChoiceItem      BOOL_VAL1       CHOICE_1
7  Item            BOOL_VAL1       boolean
8  Depends         BOOL_VAL1       "CHOICE_1"
9  Prompt          BOOL_VAL1       "CHOICE_1"
10 HasPrompts      BOOL_VAL1       1
11 #choice value
12 ChoiceItem      BOOL_VAL2       CHOICE_1
13 Item            BOOL_VAL2       boolean
14 Depends         BOOL_VAL2       "CHOICE_1"
15 Prompt          BOOL_VAL2       "CHOICE_1"
16 HasPrompts      BOOL_VAL2       1
17 #choice value
18 #endchoice
```

Figure 3.111: KConfig Reader's RSF translation of Figure 2.33 on page 42 (correct).

Contrary to dumpconf (cf. Figure 3.107 on the previous page), KConfig Reader adds further information about prompts of choices to the written RSF file (cf. line 3). Though this additional information, the translation becomes correct. However, KConfig Reader also throws a warning while translating the model (cf. Figure 3.112)

```
1  [info] Running de.fosd.typechef.kconfig.KConfigReader --dumpconf /tools/
       build/KConfigReader/undertaker/scripts/kconfig/dumpconf --writeDimacs /
       data/src/Linux-Releases/UndertakerTest/ /data/results/KConfigReader/
       Report/InvisibleChoice
2  dumping model
3  setting archusing arch x86
4  arch/x86/Kconfig:1:warning: choice must have a prompt
5  reading model
6  getting constraints
7  checking combined constraint
8  writing model
9  writing dimacs
10 done.
11 [success] Total time: 2 s, completed May 18, 2015 2:28:03 AM
```

Figure 3.112: Displayed warning while translating Figure 2.33 on page 42.

Model

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

Figure 3.113: KConfig Reader's Model translation of Figure 2.33 on page 42 (error in line 4).

The Model translation of KConfig Reader does not use the additional information about the choice's prompt and creates a Model with a permanently selected choice (cf. Figure 3.113 on the next page, line 4). Thus, we categorized the translation as not correct.

CNF

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

Figure 3.114: KConfig Reader's CNF translation of Figure 2.33 on page 42 (error in line 6).

Figure 3.114 demonstrates how KConfig Reader translates a choice without any prompts. The constraint in line 6 forces the permanent selection of the choice. Thus, also this translation is categorized as not correct.

3.13.3 KConfig Reader (XML)

Model

Figure 3.115 on page 129 permanently deselects `CHOICE_1` in line 8. Lines 2 and 5 force the deselection of `BOOL_VAL1` and `BOOL_VAL2`, if `CHOICE_1` is deselected. Thus, this translation is correct.

```
#item BOOL_VAL1
(!def(BOOL_VAL1)|def(CHOICE_1))
(!def(BOOL_VAL1)|def(CHOICE_1))
#item BOOL_VAL2
(!def(BOOL_VAL2)|def(CHOICE_1))
(!def(BOOL_VAL2)|def(CHOICE_1))
#item CHOICE_1
!def(CHOICE_1)
(!def(BOOL_VAL1)|def(CHOICE_1))
(def(BOOL_VAL1)|!def(CHOICE_1))
#choice CHOICE_1
(def(BOOL_VAL2)|def(BOOL_VAL1)|!def(CHOICE_1))
(!def(BOOL_VAL2)|def(CHOICE_1))
(!def(BOOL_VAL1)|def(CHOICE_1))
(!def(BOOL_VAL2)|!def(BOOL_VAL1))
```

Figure 3.115: KConfig Reader (XML)'s Model translation of Figure 2.33 (correct).

CNF

```
c 1 CHOICE_1
c 2 BOOL_VAL1
c 3 BOOL_VAL2
c 4 MODULES
p cnf 4 12
-1 0
-2 1 0
2 -1 0
-2 1 0
-2 1 0
-3 1 0
-3 1 0
3 2 -1 0
-3 1 0
-2 1 0
-3 -2 0
-4 0
```

Figure 3.116: KConfig Reader (XML)'s CNF translation of Figure 2.33 (correct).

Figure 3.116 on the following page contains the same logical information as Figure 3.115 on the next page. Thus, this translation is also correct.

3.13.4 LVAT

LVAT fails to translate Figure 2.33 on page 42. The stacktrace contains a message about a failed regular expression match in a binary file, for which reason we did not show it here.

3.14 Recursive Dependency inside a Choice via an `if`

In Section 2.4.8, we discussed the special situation of a recursive dependency caused by an if expression inside a structured choice.

For the analysis, we shortened the example from Section 2.4.8 and used the model shown in Figure 3.117. `SUB_VAL` will not be selectable, since the hierarchy of the nested elements of the choice is not also considered inside the `if` expression in line 12.

```
1  config VAR
2      bool "Not nested Variable"
3
4  choice
5      bool "Choice"
6      config VAL_1
7          bool "Value 1"
8
9      config VAL_2
10         bool "Value 2"
11
12         if VAR
13             config SUB_VAL
14                 bool "Dead Sub Value"
15                 depends on VAL_2
16         endif
17 endchoice
```

Figure 3.117: Modification of Figure 2.36 on page 44: `SUB_VAL` will not be selectable.

3.14.1 Undertaker

dumpconf

While translating the model from Figure 3.117, dumpconf and rsf2model display the same warning as already shown in Figure 3.89 on page 111. The `if` statement of line 12 and the `depends on` constraint of line 15 are combined to one `Depends` statement inside the resulting RSF file. Hence, the resulting RSF files does not facilitate the detection of the recursive dependency. Thus, we categorized this translation is incorrect.

```
1  Choice          CHOICE_1        required        boolean
2  ChoiceItem      SUB_VAL         CHOICE_1
3  ChoiceItem      VAL_1           CHOICE_1
4  ChoiceItem      VAL_2           CHOICE_1
5  Depends         SUB_VAL         "CHOICE_1 && VAR && VAL_2"
6  Depends         VAL_1           "CHOICE_1"
7  Depends         VAL_2           "CHOICE_1"
8  HasPrompts      SUB_VAL         1
9  HasPrompts      VAL_1           1
10 HasPrompts      VAL_2           1
11 HasPrompts      VAR             1
12 Item            SUB_VAL         boolean
13 Item            VAL_1           boolean
14 Item            VAL_2           boolean
15 Item            VAR             boolean
```

Figure 3.118: dumpconf's translation of Figure 3.117 (error in line 5).

rsf2model

```
1  CONFIG_CHOICE_1 "((CONFIG_SUB_VAL && !CONFIG_VAL_1 &&
       !CONFIG_VAL_2) || (!CONFIG_SUB_VAL && CONFIG_VAL_1 &&
       !CONFIG_VAL_2) || (!CONFIG_SUB_VAL && !CONFIG_VAL_1 &&
       CONFIG_VAL_2))"
2  CONFIG_SUB_VAL "(CONFIG_VAR && CONFIG_VAL_2) && CONFIG_CHOICE_1"
3  CONFIG_VAL_1 "CONFIG_CHOICE_1"
4  CONFIG_VAL_2 "CONFIG_CHOICE_1"
5  CONFIG_VAR
6  UNDERTAKER_SET ALWAYS_ON "CONFIG_CHOICE_1"
7  UNDERTAKER_SET SCHEMA_VERSION 1.1
8  CONFIG_X86 ""
9  CONFIG_n
10 CONFIG_y
11 CONFIG_m
```

Figure 3.119: rsf2model's translation of Figure 3.118 on page 131 (error).

Figure 3.119 shows how rsf2model translates the RSF file from Figure 3.118 into boolean formula. Lines 1 and 2 avoid the selection of `CONFIG_SUB_VAL`, which is correct. However, fixing the KConfig model by moving the dependency of line 15 into the `if` statement in line 12, will result in the same Model file. Thus, we categorized this translation as not correct.

Satyr

Also Satyr displays a warning while translating the model from Figure 3.117 into the model shown in Figure 3.120 on page 133. This model does not allow the selection

of `CONFIG_SUB_VAL`. After fixing the KConfig model as described above, the resulting DIMACS file allows the selection of `CONFIG_SUB_VAL` if `CONFIG_VAR` and `CONFIG_VAL_2` are also selected. Thus, we categorized this translation as correct.

```
c File Format Version: 2.0
c Generated by satyr
c Type info:
c c sym <symbolname> <typeid>
c with <typeid> being an integer out of:
c enum {S_BOOLEAN=1, S_TRISTATE=2, S_INT=3, S_HEX=4, S_STRING=5, S_OTHER=6}
c variable names:
c c var <variablename> <cnfvar>
c meta_value ALWAYS_ON CONFIG_CHOICE_0
c sym CHOICE_0 1
c sym SUB_VAL 1
c sym UNAME_RELEASE 5
c sym VAL_1 1
c sym VAL_2 1
c sym VAR 1
c sym ____MODULES_MAGIC_INTERNAL_VAR____ 1
c var CONFIG_CHOICE_0 1
c var CONFIG_SUB_VAL 5
c var CONFIG_UNAME_RELEASE 15
c var CONFIG_VAL_1 2
c var CONFIG_VAL_2 3
c var CONFIG_VAR 18
c var CONFIG_____MODULES_MAGIC_INTERNAL_VAR____ 14
p cnf 21 48
1 0
-4 2 0
-4 -3 0
4 -2 3 0
-6 4 0
-6 -5 0
6 -4 5 0
-7 -2 0
-7 3 0
7 2 -3 0
-8 7 0
-8 -5 0
8 -7 5 0
9 -6 0
9 -8 0
-9 6 8 0
-10 -2 0
-10 -3 0
10 2 3 0
-11 10 0
-11 5 0
11 -10 -5 0
12 -9 0
12 -11 0
-12 9 11 0
13 1 0
13 -12 0
```

```
52 −13 −1 12 0
53 13 0
54 −14 0
55 16 3 0
56 16 −1 0
57 −16 −3 1 0
58 16 0
59 17 2 0
60 17 −1 0
61 −17 −2 1 0
62 17 0
63 −19 1 0
64 −19 18 0
65 19 −1 −18 0
66 −20 19 0
67 −20 3 0
68 20 −19 −3 0
69 21 5 0
70 21 −20 0
71 −21 −5 20 0
72 21 0
```

Figure 3.120: Satyr's CNF translation of Figure 3.117 (correct).

3.14.2 KConfig Reader

RSF

KConfig Reader creates a RSF file which looks very similar to the RSF file of Figure 3.118. This translation is not correct, since also KConfig Reader combines the `if` statement and the `depends on` constraint into one `Depends` statements. KConfig Reader is creating a warning while writing the RSF file as already shown in Figure 3.94 on page 114.

```
1  Item            VAR             boolean
2  Prompt          VAR             "y"
3  HasPrompts      VAR             1
4  #startchoice
5  Choice          CHOICE_1        required        boolean
6  Prompt          CHOICE_1        "y"
7  HasPrompts      CHOICE_1        1
8  #choice value
9  ChoiceItem      VAL_1           CHOICE_1
10 Item            VAL_1           boolean
11 Depends         VAL_1           "CHOICE_1"
12 Prompt          VAL_1           "CHOICE_1"
13 HasPrompts      VAL_1           1
14 #choice value
15 ChoiceItem      VAL_2           CHOICE_1
16 Item            VAL_2           boolean
17 Depends         VAL_2           "CHOICE_1"
18 Prompt          VAL_2           "CHOICE_1"
19 HasPrompts      VAL_2           1
20 #choice value
21 ChoiceItem      SUB_VAL         CHOICE_1
22 Item            SUB_VAL         boolean
23 Depends         SUB_VAL         "CHOICE_1 && VAR && VAL_2"
24 Prompt          SUB_VAL         "CHOICE_1 && VAR && VAL_2"
25 HasPrompts      SUB_VAL         1
26 #choice value
27 #endchoice
```

Figure 3.121: KConfig Reader's RSF translation of Figure 3.117 on page 130 (error in line 23).

Model

The produced Model file shown in Figure 3.122 allows the selection of exactly one nested config option (cf. lines 10 – 16). It also models the dependency between `SUB_VAL`, `VAR` and `VAL_2` (cf. lines 4 and 5), which is correct. However, KConfig Reader produces the same output if the KConfig model was fixed before. Thus, we categorized this translation as incorrect.

```
#item CHOICE_1
def(CHOICE_1)
#item SUB_VAL
(!def(SUB_VAL)|(def(VAR)&def(VAL_2)))
(!def(SUB_VAL)|(def(VAR)&def(VAL_2)))
#item VAL_1
#item VAL_2
#item VAR
#choice CHOICE_1
(def(SUB_VAL)|def(VAL_2)|def(VAL_1)|!def(CHOICE_1))
(!def(SUB_VAL)|def(CHOICE_1))
(!def(VAL_2)|def(CHOICE_1))
(!def(VAL_1)|def(CHOICE_1))
(!def(SUB_VAL)|!def(VAL_2))
(!def(SUB_VAL)|!def(VAL_1))
(!def(VAL_2)|!def(VAL_1))
```

Figure 3.122: KConfig Reader's Model translation of Figure 3.117 on page 130 (error).

CNF

```
c 5 VAL_1
c 3 VAR
c 1 VAL_2
c 2 SUB_VAL
c 4 CHOICE_1
c 6 MODULES
p cnf 6 13
1 -2 0
3 -2 0
1 -2 0
3 -2 0
4 0
2 1 5 -4 0
-2 4 0
-1 4 0
-5 4 0
-2 -1 0
-2 -5 0
-1 -5 0
-6 0
```

Figure 3.123: KConfig Reader's CNF translation of Figure 3.117 on page 130 (error).

KConfig Reader produces also the same DIMACS file independently whether the KConfig model was fixed before. The result is shown in Figure 3.123. Thus, also this kind of translation is not correct.

3.14.3 KConfig Reader (XML)

Model

KConfig Reader (XML)'s Model translation of Figure 3.117 on page 130 is identical to KConfig Reader's translations (cf. Section 3.14.2).

Figure 3.124 on the next page shows KConfig Reader (XML)'s Model translation of the fixed model, which is different from KConfig Reader's. `SUB_VAL` is correctly excluded from the XOR connection on the other choice items. Thus, this translation is correct.

```
#item CHOICE_1
def(CHOICE_1)
#item SUB_VAL
(!def(SUB_VAL)|(def(VAR)&def(VAL_2)&def(CHOICE_1)))
(!def(SUB_VAL)|(def(VAR)&def(VAL_2)&def(CHOICE_1)))
#item VAL_1
(!def(VAL_1)|def(CHOICE_1))
(!def(VAL_1)|def(CHOICE_1))
#item VAL_2
(!def(VAL_2)|def(CHOICE_1))
(!def(VAL_2)|def(CHOICE_1))
#item VAR
#choice CHOICE_1
(def(VAL_2)|def(VAL_1)|!def(CHOICE_1))
(!def(VAL_2)|def(CHOICE_1))
(!def(VAL_1)|def(CHOICE_1))
(!def(VAL_2)|!def(VAL_1))
```

Figure 3.124: KConfig Reader (XML)'s Model translation of the fixed version of Figure 3.117 on page 130 (correct).

CNF

KConfig Reader (XML)'s CNF translation of Figure 3.117 on page 130 is identical to KConfig Reader's translations (cf. Section 3.14.2 on page 135).

Figure 3.125 on the following page shows KConfig Reader (XML)'s CNF translation of the fixed model, which is different from KConfig Reader's. It contains the same logical information as Figure 3.124 and is therefore correct.

```
c 3 VAR
c 2 SUB_VAL
c 6 MODULES
c 4 CHOICE_1
c 1 VAL_2
c 5 VAL_1
p cnf 6 16
1 -2 0
3 -2 0
4 -2 0
1 -2 0
3 -2 0
4 -2 0
-1 4 0
-1 4 0
-5 4 0
-5 4 0
4 0
1 5 -4 0
-1 4 0
-5 4 0
-1 -5 0
-6 0
```

Figure 3.125: KConfig Reader (XML)'s CNF translation of the fixed version of Figure 3.117 on page 130 (correct).

3.14.4 LVAT

In Figure 3.126 on page 138 `SUB_VAL` is not present at all. Usually LVAT keeps not selectable variables and adds constraints to deselect permanently (e.g. see `CONST_FALSE` in Figure 3.25 on page 69). Thus, the translation is considered false, since there are no constraints that require `SUB_VAL` to be not selected. Thus, we categorized this translation as incorrect.

When translating the fixed KConfig model however, `SUB_VAL` is correctly added to the CNF model.

```
c 1$ _X9
c 2$ _X5
c 3$ _X1_m
c 4 VAL_2
c 5$ _X5_m
c 6$ _X7_m
c 7 VAL_2_m
c 8 VAL_1
c 9$ _X2
c 10$ _X1
c 11$ _X6
c 12 VAR_m
c 13$ _X3
c 14$ _X4_m
c 15 VAL_1_m
c 16$ _X7
c 17$ _X3_m
c 18$ _X4
c 19$ _X8_m
c 20$ _X6_m
c 21$ _X9_m
c 22 VAR
c 23$ _X8
c 24$ _X2_m
p cnf 24 47
-10 0
-3 0
-10 22 12 0
-3 22 12 0
-10 -3 -22 12 0
-22 12 0
22 -12 0
-9 0
-24 0
-9 -22 10 0
-9 -10 22 0
-9 -12 3 0
-9 -3 12 0
-13 0
-17 0
-18 0
-14 0
-18 8 15 0
-14 8 15 0
-18 -14 -8 15 0
-8 15 0
8 -15 0
-2 0
-5 0
-2 -8 18 0
-2 -18 8 0
-2 -15 14 0
-2 -14 15 0
-11 0
-20 0
-16 0
-6 0
-16 4 7 0
-6 4 7 0
-16 -6 -4 7 0
-4 7 0
4 -7 0
-23 0
-19 0
-23 -4 16 0
-23 -16 4 0
-23 -7 6 0
-23 -6 7 0
-1 0
-21 0
8 4 0
-8 -4 0
```

Figure 3.126: LVAT's CNF translation of Figure 3.117 on page 130 (error).

3.15 Multiple `Default` Values inside a Config Option

In Section 2.4.9, we showed that it is possible that multiple attributes of the same type can become active. Especially `default` values can lead in contradictions.

We used the model from Figure 3.127 for our analysis. `VAR` has two `default` values which are permanently active. Only the first one ('`y`') is considered by KConfig.

```
1 config VAR
2     def_bool y
3     def_bool n
```

Figure 3.127: KConfig model for testing the translation of multiple `default` values.

3.15.1 Undertaker

dumpconf

```
1 Default      VAR    "n"    "y"
2 Default      VAR    "y"    "y"
3 HasPrompts   VAR    0
4 Item         VAR    boolean
```

Figure 3.128: dumpconf's translation of Figure 3.127 (error in line 1).

Figure 3.128 contains the same logical information than the related KConfig model, but it reorders the default values. As a consequence, `default n` becomes the first default value. Thus, we categorized this translation as not correct.

rsf2model

```
1 CONFIG_VAR "(CONFIG_n || __FREE__0)"
2 UNDERTAKER_SET ALWAYS_ON "CONFIG_VAR"
3 UNDERTAKER_SET SCHEMA_VERSION 1.1
4 CONFIG_X86 ""
5 CONFIG_n
6 CONFIG_y
7 CONFIG_m
```

Figure 3.129: rsf2model's translation of Figure 3.128 (error).

Figure 3.129 shows how rsf2model handles multiple `default` attributes. In line 2, `CONFIG_VAR` is permanently selected, which reflects a correct translation of the `default y` attribute. Line 1, specifies that `CONFIG_VAR` can be set to CONFIG_n (translated `default n` attribute) or to an arbitrary value (`|| __FREE__0`). Thus, the translated model allows only valid configurations. However, rsf2model produces the same output if the ordering of the `default` attributes are changed in Figure 3.127, probably related due to the absence of the correct ordering inside the intermediate RSF file. Thus, we categorized this translation as not correct.

Satyr

```
1  c File Format Version: 2.0
2  c Generated by satyr
3  c Type info:
4  c c sym <symbolname> <typeid>
5  c with <typeid> being an integer out of:
6  c enum {S_BOOLEAN=1, S_TRISTATE=2, S_INT=3, S_HEX=4, S_STRING=5, S_OTHER=6}
7  c variable names:
8  c c var <variablename> <cnfvar>
9  c meta_value ALWAYS_ON CONFIG_VAR
10 c sym UNAME_RELEASE 5
11 c sym VAR 1
12 c sym ____MODULES_MAGIC_INTERNAL_VAR____ 1
13 c var CONFIG_UNAME_RELEASE 2
14 c var CONFIG_VAR 3
15 c var CONFIG_____MODULES_MAGIC_INTERNAL_VAR____ 1
16 p cnf 3 2
17 -1 0
18 3 0
```

Figure 3.130: Satyr's CNF translation of Figure 3.127 on page 139 (correct).

Figure 3.130 shows how Satyr translates the model from Figure 2.38 on page 46. The constraint in line 18 considers the correct `default` value. This constraint will be changed correctly, if the `default` values are reordered inside the related KConfig file (cf. Figure 3.131).

```
1 c sym VAR 1
2 c sym ____MODULES_MAGIC_INTERNAL_VAR____ 1
3 c var CONFIG_UNAME_RELEASE 2
4 c var CONFIG_VAR 3
5 c var CONFIG_____MODULES_MAGIC_INTERNAL_VAR____ 1
6 p cnf 3 2
7 -1 0
8 -3 0
```

Figure 3.131: Satyr's CNF translation of Figure 3.127 on the preceding page (excerpt), with changed ordering of the `default` values (correct).

3.15.2 KConfig Reader

RSF

```
Item          VAR          boolean
HasPrompts    VAR          0
Default       VAR          " 'y' "          "y"
Default       VAR          " 'n' "          "y"
```

Figure 3.132: KConfig Reader's RSF translation of Figure 3.127 on page 139 (correct).

Figure 3.132 shows that KConfig Reader is taking the correct ordering of the `default` values while creating the RSF files. The ordering will change, if the ordering inside the related KConfig file is changed. Thus, we categorized the translation as correct.

Model

```
#item VAR
def(VAR)
```

Figure 3.133: KConfig Reader's Model translation of Figure 3.127 on page 139 (correct).

Figures 3.133 and 3.134 show that also the correct `default` value is used for the creation of the Model files. Thus, also this translation is correct.

```
#item VAR
!def(VAR)
```

Figure 3.134: KConfig Reader's Model translation of Figure 3.127 on page 139, with changed ordering of the `default` values (correct).

CNF

```
c 1 VAR
c 2 MODULES
p cnf 2 2
1 0
-2 0
```

Figure 3.135: KConfig Reader's CNF translation of Figure 3.127 on page 139 (correct).

Figures 3.135 and 3.136 show that KConfig Reader is also considering the correct ordering of the `default` values while creating CNF formula. Thus, also this translation is correct.

```
1 c 1 VAR
2 c 2 MODULES
3 p cnf 2 2
4 -1 0
5 -2 0
```

Figure 3.136: KConfig Reader's CNF translation of Figure 3.127 on page 139, with changed ordering of the `default` values (correct).

3.15.3 KConfig Reader (XML)

Model & CNF

Both translations of KConfig Reader (XML) (Model and CNF) of Figure 3.127 on page 139 are identical to the related translations of KConfig Reader (cf. Section 3.15.2) and therefore correct.

3.15.4 LVAT

After removing the constant `_X4` in lines 23 and 24 in Figure 3.137, these constraint show that the correct default value was used. Figure 3.138 shows the translation of the reverse ordering of the default values. The constraints in lines 23 and 25 force the correct constant values. Thus, this translation is correct.

```
1 c 1$ _X5
2 c 2$ _X1_m
3 c 3$ _X5_m
4 c 4$ _X2
5 c 5$ _X1
6 c 6 VAR_m
7 c 7$ _X3
8 c 8$ _X4_m
9 c 9$ _X3_m
10 c 10$ _X4
11 c 11 VAR
12 c 12$ _X2_m
13 p cnf 12 25
14 -5 0
15 -2 0
16 -5 11 6 0
17 -2 11 6 0
18 -5 -2 -11 6 0
19 -11 6 0
20 11 -6 0
21 4 0
22 -12 0
23 -4 11 0
24 -4 6 0
25 -7 0
26 -9 0
27 -7 -11 5 0
28 -7 -5 11 0
29 -7 -6 2 0
30 -7 -2 6 0
31 -10 0
32 -8 0
33 -10 -11 5 0
34 -10 -5 11 0
35 -10 -6 2 0
36 -10 -2 6 0
37 -1 0
38 -3 0
```

Figure 3.137: LVAT's CNF translation of Figure 3.127 on page 139 (correct).

```
c 1$ _X5
c 2$ _X1_m
c 3$ _X5_m
c 4$ _X2
c 5$ _X1
c 6 VAR_m
c 7$ _X3
c 8$ _X4_m
c 9$ _X3_m
c 10$ _X4
c 11 VAR
c 12$ _X2_m
p cnf 12 25
−5 0
−2 0
−5 11 6 0
−2 11 6 0
−5 −2 −11 6 0
−11 6 0
11 −6 0
4 0
−12 0
−4 −11 5 0
−4 −5 11 0
−4 −6 2 0
−4 −2 6 0
−7 0
−9 0
−7 11 0
−7 6 0
−10 0
−8 0
−10 −11 5 0
−10 −5 11 0
−10 −6 2 0
−10 −2 6 0
−1 0
−3 0
```

Figure 3.138: LVAT's CNF translation of Figure 3.127 on page 139, with changed ordering of the `default` values (correct).

3.16 Summary

Table 3.2 on the next page summarizes the findings of our tool analysis. None of the analysed tools was able to handle all observations. Also the `option modules` attribute, which is used to control the Tristate semantics instead of the old mechanism, is not supported by any tool. Instead, all tools are able to handle the correct constraint precedence (`select` over `depends on` / `if`), which is a fundamental concept of KConfig and, thus, much more important as the afore mentioned findings. All tools are also able to handle undefined config options in constraints.

Satyr and KConfig Reader (XML) produce more reliable results than the other tools. Most of the observations not handled by them are not used inside recent versions of Linux, e.g. the Tristate semantics is controlled by a config option attributed with `option modules`, but also named as `MODULES`. Only default values of Tristate config options may violate the CNF formula produced by Satyr, if the containing config option has no prompt and will not be displayed to the user. Also constraints based on Strings and Numbers are not considered. The other two corner cases not handled by Satyr, should not be present in recent version of Linux. KConfig Reader (XML) is not able to handle "Selection of Nested Config Options" and and "Empty Choices" correctly. While we found examples for the later one only in an old version of the Linux kernel (version 2.6.33.3), the first one is still part of a more recent version of Linux (version 3.19). Contrary to Satyr, KConfig Reader (XML) supports constraints containing comparisons of Strings and Numbers.

Also KConfig Reader's translation into RSF files can be used for writing own analysis tools. This tool can handle most of the concepts of KConfig as it relies on a patched version of dumpconf, which in turn relies on menuconfig. However, it does not cover the `option modules` attribute and can not distinguish between a "recursive dependency" and a structured choice, if an `if` statement is modelled inside a choice. Both observations should not be very problematic, since recent versions of Linux still controlled over a config option named as `MODULES` and "recursive dependencies" are displayed as warnings by menuconfig and also during the translation of KConfig Reader. We expect that the XML files produced by KConfig Reader (XML) have at least the same quality, because of the improved results of the downstream translations. However, since we did include the generated XML files into our analysis, we cannot guarantee that.

		Undertaker			KConfigReader			KConfigReader (XML)		LVAT
Observation	**Section**	**Satyr**	**dumpconf**	**rsf2model**	**CNF**	**RSF**	**Model**	**Model**	**CNF**	
Handling Attribute `option modules`	2.1	✗	✗	✗	✗	✗	✗	✗	✗	✗
Constraint Precedence	2.1	✓	✓	✗	✓	✓	✓	✓	✓	✓
Missing Config Options	2.1	✓	✓	✓	✓	✓	✓	✓	✓	✓
Selection of Nested Config Options	2.4.1	✓	🤔	✗	✗	🤔	✗	✗	✗	✗
Default Value `m` for Booleans	2.4.2	✓	🤔	✗	✓	🤔	✓	✓	✓	✗
Default Value `m` for Tristates	2.4.2	✗	✓	✓	✓	✓	✓	✓	✓	✗
Tristate Choice with Boolean Config Options	2.4.3	✓	✓	✓	–	✓	–	✓	✓	✗
Boolean Choice with Tristate Config Options	2.4.3	✗	✓	✓	✗	✓	✗	✗	✗	✗
Structured Choices	2.4.4	✓	✗	✗	✗	✓	✗	✓	✓	✓
Recursive Dependency inside a Choice	2.4.5	✓	✗	✓	✓	✓	✓	✓	✓	✓
Empty Choices	2.4.6	✗	✓	✗	–	✓	–	–	–	✗
Choices Without a Prompt	2.4.7	✗	✗	✗	✗	✓	✗	✓	✓	–
Recursive Dependency inside a Choice via an `if`	2.4.8	✓	✗	✗	✗	✗	✗	✓	✓	✗
Multiple `Default` Values inside a Config Option	2.4.9	✓	✗	✗	✓	✓	✓	✓	✓	✓

Table 3.2: Summary of tool analysis (– = translation aborted; ✗ = incorrect translation; 🤔 = problematic; ✓ = correct translation).

Chapter 4

Summary

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

In the second part, we analysed the capabilities of existing analysis tools, used in scientific research. We started with an introduction of generated output formats and the resulting capabilities. We presented the results of the analysis in Table 3.2. These results can be used to select an appropriate tool for performing own analysis, but also for improvement of these tools.

The results presented in Chapter 3 influence published analysis results, as they were gathered with the tools analysed here. LVAT was used to analyse the structure of KConfig models, like breadth and depth of generated Feature Models or the number of XOR groups [BSL+13]. At least the analysis regarding the breadth and depth of the model should be correct, because LVAT is considering constraint hierarchies correctly. However, we expect problems regarding constraint analysis as LVAT is not aware of all corner cases.

Acknowledgements

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

Bibliography

[BSL+12] Thorsten Berger, Steven She, Rafael Lotufo, Andrezj Wasowski, and Krzysztof Czarnecki. Variability Modeling in the Systems Software Domain. Technical Report GSDLAB-TR 2012-07-06, Generative Software Development Laboratory, University of Waterloo, 2012.

[BSL+13] Thorsten Berger, S. She, R. Lotufo, A. Wasowski, and Krzysztof Czarnecki. A Study of Variability Models and Languages in the Systems Software Domain. *IEEE Transactions on Software Engineering*, 39(12):1611–1640, Dec 2013.

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

[kcob] kconfigreader. `https://github.com/ckaestne/kconfigreader`. Last visited 15.05.2015.

[KCo14] KConfig Language Specification. `https://www.kernel.org/doc/Documentation/kbuild/kconfig-language.txt`, 2014. Last visited 04.03.2015.

[lin] linux-variability-analysis-tools. `https://code.google.com/p/linux-variability-analysis-tools/`. Last visited 08.06.2015.

[Pro] `https://code.google.com/p/linux-variability-analysis-tools/wiki/PropositionalTranslation`. Last visited 29.06.2015.

[Sat93] Satisfiability Suggested Format. `ftp://dimacs.rutgers.edu/pub/challenge/satisfiability/doc/satformat.dvi`, 1993. Last visited 04.03.2015.

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