

Encoding the Linux Kernel Configuration in Propositional Logic

Christoph Zengler¹

Abstract. We present a formalization of the Linux Kernel configuration and propose a set of rules how to encode this formalization in propositional logic. The resulting propositional formulas describe all valid configurations of the Linux Kernel with respect to one specific hardware architecture. The advantage of a formula in propositional logic is that we can use all the elaborate tools and techniques from the SAT community like SAT solvers, parametric SAT solvers, or model counters. We show how we can use these available tools to perform e.g. an automated search for redundant or necessary options or automatically produce configuration variants. We have implemented our approach and compiled the formulas for all available hardware architectures. Based on this implementation, we show some experimental results on size and complexity of the resulting formulas.

1 INTRODUCTION

The monolithic Linux Kernel is a highly configurable software system. The Linux Kernel supports more than 60 different hardware platforms grouped in about 25 hardware architectures. There are various kernel sub-systems e.g. for memory management, file systems, or network protocols. There are thousands of device drivers for all different kinds of hardware (e.g. scanners, ISDN cards, video cards, etc.) [1]. When compiling the kernel one can choose between thousands of options. These options are organized in the so called `KConfig` files for the `kbuild` system. There are over 1000 `KConfig` files, which – combined together – describe the complete Linux Kernel Configuration (LKC).

Our approach is to translate these files into one propositional formula for each hardware architecture. This formula then describes all valid configurations. A formalization of configuration problems in propositional logic is a well studied approach which is suitable for large scale industrial problems [6]. We show how to use standard methods and tools for propositional formulas to test, manipulate and list kernel configurations.

In [4] the LKC is described in terms of a software product line. In [5] the same authors present a Linux Kernel configuration tool. Post and Sinz describe a technique called *Configuration Lifting* [3]. Their idea is to directly encode certain options into a program that should be verified. Thus you cannot only verify that a program is correct for one specific configuration, but for all possible configurations (wrt. to the lifted configuration options).

To the best of our knowledge our approach presented in this paper is new and merits further research. None of the above mentioned approaches compile the complete LKC into one formula, which can then be verified. With our methods we can solve problems that could

to the best of our knowledge not be solved before e.g. automatically produce variants wrt. a dedicated set of options or counting the number of valid configurations. Also, our approach is not only for users, configuring the kernel, but also for developers, checking the validity of their documentation of the options. All described methods are implemented and we present some experimental results in Section 5.

2 THE LINUX KERNEL CONFIGURATION

For most options the user can choose between three states: (*y*) The respective option is compiled directly into the kernel; (*m*) the option is compiled as module and can be loaded and unloaded into the kernel at runtime; (*n*) the option is not available at all. Options can depend on the state of other options (*dependencies*) or can influence the state of other options (*selections*). All options, their dependencies, and their selections are described in the `KConfig` files.

2.1 Configuration Options

A single `KConfig` file is a collection of configuration options (also referred to as configuration symbols). There are five different types *t* of configuration options: `tristate`, `bool`, `string`, `int`, and `hex`. We denote their respective sets by \mathcal{O}_T , \mathcal{O}_B , \mathcal{O}_S , \mathcal{O}_I , and \mathcal{O}_H . The set of all configuration options is denoted by $\mathcal{O} = \mathcal{O}_T \cup \mathcal{O}_B \cup \mathcal{O}_S \cup \mathcal{O}_I \cup \mathcal{O}_H$. We use uppercase letters ‘*A*’, ‘*B*’, ... to denote configuration options of \mathcal{O} . Each set *S* of options has its own domain $\text{dom}(S)$. The domains for a set expand in a natural way to elements of this set such that for each option $s \in S$ with $\text{dom}(S) = D$ we have $\text{dom}(s) = D$. The domains for the five different sets of configuration options are: $\text{dom}(\mathcal{O}_T) = \{0, 1, 2\}$, $\text{dom}(\mathcal{O}_B) = \{0, 2\}$, $\text{dom}(\mathcal{O}_S) = \mathbb{S}$, $\text{dom}(\mathcal{O}_I) = \mathbb{S}_I$, $\text{dom}(\mathcal{O}_H) = \mathbb{S}_H$. \mathbb{S} is the set of all valid strings, \mathbb{S}_I is the set of all valid strings representing an integer, \mathbb{S}_H is the set of all valid strings representing a number in hexadecimal format. We can clearly see, that $\text{dom}(\mathcal{O}_B) \subset \text{dom}(\mathcal{O}_T)$, $\text{dom}(\mathcal{O}_I) \subset \text{dom}(\mathcal{O}_S)$, and $\text{dom}(\mathcal{O}_H) \subset \text{dom}(\mathcal{O}_S)$. Thus all five types can be derived from the two main types `tristate` and `string`.

Since there are three different states for all `tristate` symbols $s \in \mathcal{O}_T$ and `bool` symbols can be derived from `tristate` symbols, the configuration files use a tristate logic for expressions over configuration options of $\mathcal{O}_T \cup \mathcal{O}_B$. This logic uses one more state than normal boolean logic to express the module state. For calculations we identify *n* with 0, *m* with 1, and *y* with 2. An *option assignment* is a partial function $\alpha_{\mathcal{O}} : \mathcal{O} \rightarrow \{0, 1, 2\} \cup \mathbb{S}$ mapping configuration options of $\mathcal{O}_T \cup \mathcal{O}_B$ to one of the three states 0, 1, 2 and options of $\mathcal{O}_S \cup \mathcal{O}_I \cup \mathcal{O}_H$ to their corresponding string value. We can now extend the definitions of ‘not’ (!), ‘and’ (&&), and ‘or’ (||) in

¹ Symbolic Computation Group, WSI Informatics, Universität Tübingen, Germany, email: zengler@informatik.uni-tuebingen.de

a natural way. For an option $A \in \mathcal{O}$ and arbitrary tristate expression e_0, \dots, e_n , we define the evaluation β of a tristate expression:

$$\begin{aligned}\beta(\mathbf{y}) &= 2 \\ \beta(\mathbf{m}) &= 1 \\ \beta(\mathbf{n}) &= 0 \\ \beta(A) &= \alpha_{\mathcal{O}}(A) \\ \beta(!e_0) &= 2 - \beta(e_0) \\ \beta(e_0 \&\& \dots \&\& e_n) &= \min(\beta(e_0), \dots, \beta(e_n)) \\ \beta(e_0 || \dots || e_n) &= \max(\beta(e_0), \dots, \beta(e_n))\end{aligned}$$

An expression symbol s^e is either a regular configuration symbol $s \in \mathcal{O}$ or an arbitrary string $s \in \mathbb{S}$. Expression symbols can be compared with '=' and '!=', which are valid expressions. We extend β with:

$$\begin{aligned}\beta(s_1^e = s_2^e) &= 2 \text{ if } \beta(s_1^e) = \beta(s_2^e), 0 \text{ else} \\ \beta(s_1^e != s_2^e) &= 0 \text{ if } \beta(s_1^e) = \beta(s_2^e), 2 \text{ else}\end{aligned}$$

Remark. In fact, the LKC files contain only (dis)equalities of the form $A = B$ and $X = \mathbf{s}$ with $A, B \in \mathcal{O}_T \cup \mathcal{O}_B$ and $X \in \mathcal{O}_S \cup \mathcal{O}_H \cup \mathcal{O}_I$. For the rest of the paper we will only consider equalities of this form.

2.2 Configuration Files

There are about 1000 files describing the configuration database of the LKC. These files can be merged into one big file in a pre-processing step. We will present only the parts of the grammar of these files that are important for our approach. For this initial formalization we omit structures like help texts, comments, or default values.

Table 1 summarizes the abstract syntax. $s \in \mathcal{O}$ is a configuration option, $\mathbf{s} \in \mathbb{S}$ is an arbitrary string. We state meta symbols for each element. The meta symbols can include the parameters of the element e.g. in $b^I(E, \vec{B})$. If the parameters are not required, we use just the symbol without parameters, thus b^I .

A single configuration file $f(\vec{b})$ consists of a set of blocks \vec{b} . The most important block type is the configuration block $b^C(s, a^T, \vec{a}^D, \vec{a}^S)$, which describes a single configuration option $s \in \mathcal{O}$. An option s must have a type attribute a^T , which fixes the type of s to one of the five possibilities discussed above. s can have an arbitrary number of dependence and selection attributes.

A dependence attribute $a^D(e)$ describes a tristate expression e on which the current option depends. Thus e fixes an upper bound for s , meaning if $\beta(e) = 0$, then $\alpha_{\mathcal{O}}(s)$ must be 0 too, if $\beta(e) = 1$, then $\alpha_{\mathcal{O}}(s)$ can be 0 or 1, and if $\beta(e) = 2$, then s can be assigned to any state of $\{0, 1, 2\}$.

A selection attribute $a^S(s', e)$ specifies a symbol s' which must at least be assigned to the value of the current option s . Thus s fixes a lower bound for s' , meaning if $\alpha_{\mathcal{O}}(s) = 2$, s' must be assigned to 2 too, if $\alpha_{\mathcal{O}}(s) = 1$, s' can be assigned to 1 or 2, and if $\alpha_{\mathcal{O}}(s) = 0$ then s' can be assigned to any state of $\{0, 1, 2\}$. A selection attribute can have an additional tristate expression e as guarding condition. The selection will only be performed if $\beta(e) = 2$.

The menu, conditional, and choice blocks help to organize the configuration options. A menu block $b^M(\mathbf{s}, \vec{a}^D, \vec{b})$ groups several other blocks \vec{b} . The dependencies \vec{a}^D are propagated to all children blocks \vec{b} . Thus

```
menu "some text"
  depends on A

  config B
    type tristate

  config C
    type bool
endmenu
```

can be replaced by

```
config B
  type tristate
  depends on A
```

```
config C
  type bool
  depends on A
```

A conditional block $b^I(e, \vec{b})$ also groups several other blocks \vec{b} but adds only one dependence e to each of them. A choice block $b^H(t, \vec{a}^D, \vec{b}^C)$ groups several configuration options \vec{b}^C . A choice block must have a type t . If t is `bool` it means, that at most one of the enclosed options can be assigned to 2, all other options must be assigned to 0. In a `tristate` choice, at most one option can be set to 2, but an arbitrary number of options can be assigned to 1. If a choice block has dependencies \vec{a}^D , they are propagated to each child configuration option of \vec{b}^C .

3 TRANSLATION INTO PROPOSITIONAL LOGIC

For the propositional logic encoding of the configuration files, we propose a two step procedure. In the first step we build a symbol database containing all configuration symbols with their respective types, dependencies, and selections and a choice database containing all choices with their respective types and options. In the second step we translate these two databases into propositional logic.

We use the standard notation of propositional logic with propositional variables from a suitable infinite set \mathcal{V} , Boolean operators \neg , \vee , \wedge , \rightarrow , and \leftrightarrow , and Boolean constants 1 and 0. An *assignment* (in contrast to an option assignment) is a partial function $\alpha : \mathcal{V} \rightarrow \{0, 1\}$ mapping variables to truth values. We write $x \leftarrow b$ for $\alpha(x) = b$.

3.1 Step 1: Creating the Symbol and Choice Database

The symbol database \mathcal{D}_S is a set of pairs (s, desc) , where $\text{desc} = (t, \vec{e}_1, \overrightarrow{(s', e_2)})$ is a symbol descriptor consisting of the type t of s , the set of all (also the propagated) dependencies \vec{e}_1 of s and the set of all selections $\overrightarrow{(s', e_2)}$ of s . A single selection (s', e_2) contains the symbol s' that is selected from the current symbol s and the condition e_2 which guards this selection. $e_2 = \mathbf{y}$ if there is no guarding condition for this selection.

For the creation of the symbol database we propagate all dependencies of the menu, conditional and choice blocks as described in Subsection 2.2. After this step we have only configuration blocks left, which can then be translated straight forward into the corresponding symbol descriptors. The function `createDBS()` in Figure 1

Table 1. Abstract syntax for the LKC files

Level	Element	Abstract Syntax	Meta Symbol
Root level	Configuration file	$F ::= \vec{B}$	$f(\vec{B})$
Block level	block	$B ::= B_C \mid B_H \mid B_M \mid B_I$	b
	configuration	$B_C ::= \text{config } s \ A_T \ \vec{A}_D \ \vec{A}_S$	$b^C(s, A_T, \vec{A}_D, \vec{A}_S)$
	menu	$B_M ::= \text{menu } \mathfrak{s} \ \vec{A}_D \ \vec{B} \ \text{endmenu}$	$b^M(\mathfrak{s}, \vec{A}_D, \vec{B})$
	conditional	$B_I ::= \text{if } E \ \vec{B} \ \text{endif}$	$b^I(E, \vec{B})$
	choice	$B_H ::= \text{choice } (\text{tristate} \mid \text{bool}) \ \vec{A}_D \ \vec{B}_C \ \text{endchoice}$	$b^H(t, \vec{A}_D, \vec{B}_C)$
Attribute level	attribute	$A ::= A_T \mid A_D \mid A_S \mid A_R$	a
	type	$A_T ::= \text{tristate} \mid \text{bool} \mid \text{string} \mid \text{int} \mid \text{hex}$	a^T
	dependence	$A_D ::= \text{depends on } E$	$a^D(E)$
	selection	$A_S ::= \text{select } s \ [\text{if } E]$	$a^S(s, E)$
Expression level	expression	$E ::= S_E \mid (E) \mid !E \mid E \ \&\& \ E \mid E \ \ \ E \mid S_E = S_E \mid S_E != S_E \mid y \mid n \mid m$	e
	exp-symbol	$S_E ::= \mathfrak{s} \mid s$	s^e

states the exact procedure. For a file $f(\vec{b})$ it is initially called with $\text{createDBs}(\vec{b}, \emptyset)$.

Example 1 Consider the following file $f(\vec{b})$:

```

config X
  string

config B
  tristate

if B
  config C
    tristate
endif

config A
  tristate
  depends on !B
  select C if X = "s"

```

The method $\text{createDBs}(\vec{b}, \emptyset)$ yields the following symbol database:

$$\mathcal{D}_S = \{(X, (\text{string}, \emptyset, \emptyset)), (B, (\text{tristate}, \emptyset, \emptyset)), (C, (\text{tristate}, \{B\}, \emptyset)), (A, (\text{tristate}, \{!B\}, \{(C, X = "s")\}))\}$$

The choice database \mathcal{D}_C is a set of pairs (t, \vec{s}) , where t is the type of the respective choice and \vec{s} is the set of configuration symbols in the respective choice. For the creation of the choice database only choice blocks of the form

$$b^H\left(t, \vec{a}^D, \left\{b^C\left(s_1, a_1^T, \vec{a}_1^D, \vec{a}_1^S\right), \dots, b^C\left(s_n, a_n^T, \vec{a}_n^D, \vec{a}_n^S\right)\right\}\right)$$

have to be considered. This block is then translated into the pair $(t, \{s_1, \dots, s_n\})$.

Example 2 The choice database for the block

```

choice
  tristate

  config A
    tristate

  config B
    tristate

  config C
    bool
endchoice

```

is $\mathcal{D}_C = \{(\text{tristate}, \{A, B, C\})\}$.

3.2 Step 2: Translating the Databases

In the second step \mathcal{D}_S and \mathcal{D}_C are translated into corresponding encodings in propositional logic. Since dependencies and selections are represented as tristate expressions e in the symbol descriptor **desc**, we present a way how to encode the tristate logic of these expressions into a corresponding propositional representation. Afterwards we show how to encode the dependence and selection sets, before finally translating the choice database. At the end of the translation process we will have four sets: The set \mathcal{C}_O of constraints for the different configuration option types, the set \mathcal{C}_D of dependency constraints, the set \mathcal{C}_S of selection constraints, and the set \mathcal{C}_C of choice constraints. A complete encoding φ , describing all valid configurations of the Linux kernel is then given by:

$$\varphi = \bigwedge_{t \in \mathcal{C}_O} t \wedge \bigwedge_{d \in \mathcal{C}_D} d \wedge \bigwedge_{s \in \mathcal{C}_S} s \wedge \bigwedge_{c \in \mathcal{C}_C} c$$

3.2.1 Translation of the Configuration Symbols

As mentioned in Subsection 2.1, all configuration symbol types can be derived from `tristate` and `string`. We encode the three

$$\begin{aligned}
& \text{createDB}_S(\{b_1, \dots, b_n\}, \vec{e}) = \text{createDB}_S(\{b_1\}, \vec{e}) \cup \dots \cup \text{createDB}_S(\{b_n\}, \vec{e}) \\
\text{createDB}_S\left(\left\{b^C\left(s, a^T, \vec{a}^D, \vec{a}^S\right)\right\}, \vec{e}\right) &= \left\{\left(s, \text{analyzeSymb}\left(a^T, \vec{a}^D, \vec{a}^S, \vec{e}\right)\right)\right\} \\
\text{createDB}_S\left(\left\{b^M\left(\mathfrak{s}, \vec{a}^D, \vec{b}\right)\right\}, \vec{e}\right) &= \text{createDB}_S\left(\vec{b}, \vec{e} \cup \text{deps}\left(\vec{a}^D\right)\right) \\
\text{createDB}_S\left(\left\{b^I\left(e_1, \vec{b}\right)\right\}, \vec{e}\right) &= \text{createDB}_S\left(\vec{b}, \vec{e} \cup \{e_1\}\right) \\
\text{createDB}_S\left(\left\{b^H\left(a^T, \vec{a}^D, \vec{b}^C\right)\right\}, \vec{e}\right) &= \text{createDB}_S\left(\vec{b}^C, \vec{e} \cup \text{deps}\left(\vec{a}^D\right)\right) \\
\text{analyzeSymb}\left(a^T, \vec{a}^D, \vec{a}^S, \vec{e}\right) &= \left(a^T, \text{deps}\left(\vec{a}^D\right) \cup \vec{e}, \text{sels}\left(\vec{a}^S\right)\right) \\
\text{deps}\left(\left\{a^D\left(e_1\right), \dots, a^D\left(e_n\right)\right\}\right) &= \{e_1, \dots, e_n\} \\
\text{sels}\left(\left\{a^S\left(s_1, e_1\right), \dots, a^S\left(s_n, e_n\right)\right\}\right) &= \{(s_1, e_1), \dots, (s_n, e_n)\}
\end{aligned}$$

Figure 1. The function $\text{createDB}_S()$ for creating the symbol database

different states of the tristate logic with two bits in propositional logic. Each symbol $A \in \mathcal{O}_T \cup \mathcal{O}_B$ is translated into two variables $a_0, a_1 \in \mathcal{V}$, where \mathcal{V} is the set of propositional variables of our propositional encoding. The assignment $\{a_0 \leftarrow 0, a_1 \leftarrow 0\}$ represents $\alpha_{\mathcal{O}}(A) = 0$, $\{a_0 \leftarrow 0, a_1 \leftarrow 1\}$ represents $\alpha_{\mathcal{O}}(A) = 1$, and $\{a_0 \leftarrow 1, a_1 \leftarrow 0\}$ represents $\alpha_{\mathcal{O}}(A) = 2$. Since the assignment $\{a_0 \leftarrow 1, a_1 \leftarrow 1\}$ is not possible, for each $A \in \mathcal{O}_T$ the constraint $(\neg a_0 \vee \neg a_1)$ is added to \mathcal{C}_O . For each symbol $A \in \mathcal{O}_B$ we add the constraint $\neg a_1$ to \mathcal{C}_O in order to exclude the module state $\{a_0 \leftarrow 0, a_1 \leftarrow 1\}$ (and implicitly also the illegal assignment $\{a_0 \leftarrow 1, a_1 \leftarrow 1\}$).

Remark. In fact one could translate each variable $A \in \mathcal{O}_B$ with only one propositional variable $a \in \mathcal{V}$. For a uniform presentation we chose to treat *bool* and *tristate* variables the same. However, since we add the constraint $\neg a_1$ to \mathcal{C}_O for each $A \in \mathcal{O}_B$, all superfluous variables can be easily eliminated by unit propagation.

As seen in Subsection 2.1, the only valid tristate expressions containing symbols of the type *string*, *int*, or *hex* are equalities and disequalities. For a configuration symbol $X \in \mathcal{O}_S \cup \mathcal{O}_I \cup \mathcal{O}_H$ we collect all strings \mathfrak{s} occurring on the right side of an equality or disequality in a set $\text{eq}(X)$.

$$\text{eq}(X) = \{\mathfrak{s} \in \mathbb{S} \mid \text{there exists an expression } X = \mathfrak{s} \text{ or } X \neq \mathfrak{s}\}$$

We introduce new variables $x_{\mathfrak{s}} \in \mathcal{V}$, for each $\mathfrak{s} \in \text{eq}(X)$. The assignment $x_{\mathfrak{s}} \leftarrow 1$ represents the option assignment $\alpha_{\mathcal{O}}(X) = \mathfrak{s}$. Since each configuration symbol $X \in \mathcal{O}_S \cup \mathcal{O}_I \cup \mathcal{O}_H$ must be assigned to exactly one option of $\text{eq}(X)$, we add for each symbol $X \in \mathcal{O}_S \cup \mathcal{O}_I \cup \mathcal{O}_H$ the following constraint to \mathcal{C}_O :

$$\left(\bigvee_{\mathfrak{s} \in \text{eq}(X)} x_{\mathfrak{s}}\right) \wedge \bigwedge_{\mathfrak{s} \in \text{eq}(X)} \left(x_{\mathfrak{s}} \rightarrow \bigwedge_{\mathfrak{s}' \in \text{eq}(X) \setminus \{\mathfrak{s}\}} \neg x_{\mathfrak{s}'}\right)$$

3.2.2 Translation of the Symbol Database

For each tristate expression e we define two projections $\pi_0(e)$ and $\pi_1(e)$, representing the first and second bit of this expression's propositional encoding. For configuration symbols $A, B \in$

$\mathcal{O}_T \cup \mathcal{O}_B$, $X \in \mathcal{O}_S \cup \mathcal{O}_I \cup \mathcal{O}_H$ and arbitrary tristate expressions e_0, \dots, e_n , expression symbols s_1^e, s_2^e , and strings \mathfrak{s} , the definitions of π_0 and π_1 are stated in Table 2.

Example 3 Consider a tristate expression $e = A \&\&!B$. Then $\pi_0(e) = a_0 \wedge \neg b_0 \wedge \neg b_1$ and $\pi_1(e) = (a_0 \vee a_1) \wedge ((\neg b_0 \wedge \neg b_1) \vee b_1) \wedge (a_1 \vee b_1)$. We present the truth table for all possible values of A and B . One can clearly see the correspondence between $\beta(e)$ and the translations $\pi_0(e)$ and $\pi_1(e)$.

A	B	a_0	a_1	b_0	b_1	$\beta(e)$	$\pi_0(e)$	$\pi_1(e)$
0	0	0	0	0	0	0	0	0
0	1	0	0	0	1	0	0	0
0	2	0	0	1	0	0	0	0
1	0	0	1	0	0	1	0	1
1	1	0	1	0	1	1	0	1
1	2	0	1	1	0	0	0	0
2	0	1	0	0	0	2	1	0
2	1	1	0	0	1	1	0	1
2	2	1	0	1	0	0	0	0

For each pair $(s, \text{desc}) \in \mathcal{D}_S$ we add dependency and selection constraints to \mathcal{C}_D and \mathcal{C}_S . We have already seen in Subsection 2.2 how dependencies fix an upper bound for the current symbol. The following translation $\pi_D(s, e)$ of a dependence e for a symbol s assures that $\alpha_{\mathcal{O}}(s) \leq \beta(e)$:

$$\pi_D(s, e) = \overbrace{(\neg \pi_0(e) \wedge \neg \pi_1(e) \rightarrow \neg s_0 \wedge \neg s_1)}^{\beta(e)=0 \rightarrow \alpha_{\mathcal{O}}(s)=0} \wedge \overbrace{(\pi_1(e) \rightarrow \neg s_0)}^{\beta(e)=1 \rightarrow \alpha_{\mathcal{O}}(s) \neq 2}$$

The selection of a symbol s' for an option s under condition e fixes a lower bound for s' . The translation $\pi_S(s, s', e)$ assures, that (given that e holds) $\alpha_{\mathcal{O}}(s') \geq \alpha_{\mathcal{O}}(s)$:

$$\pi_S(s, s', e) = \overbrace{(s_0 \wedge \pi_0(e) \rightarrow s'_0)}^{\alpha_{\mathcal{O}}(s)=2 \rightarrow \alpha_{\mathcal{O}}(s')=2} \wedge \overbrace{(s_1 \wedge \pi_0(e) \rightarrow s'_0 \vee s'_1)}^{\alpha_{\mathcal{O}}(s)=1 \rightarrow \alpha_{\mathcal{O}}(s') \neq 0}$$

For each entry $(s, \text{desc}) \in \mathcal{D}_S$ with

$$\text{desc} = (t, \{e_1, \dots, e_n\}, \{(s'_1, e'_1), \dots, (s'_m, e'_m)\})$$

Table 2. Translation rules for tristate expressions e'

e'	$\pi_0(e')$	$\pi_1(e')$
A	a_0	a_1
$!e$	$\neg\pi_0(e) \wedge \neg\pi_1(e)$	$\pi_1(e)$
$e_0 \& \dots \& e_n$	$\pi_0(e_0) \wedge \dots \wedge \pi_0(e_n)$	$\bigwedge_{i \in \{0, \dots, n\}} (\pi_0(e_i) \vee \pi_1(e_i)) \wedge \bigvee_{i \in \{0, \dots, n\}} \pi_1(e_i)$
$e_0 \dots e_n$	$\pi_0(e_0) \vee \dots \vee \pi_0(e_n)$	$\bigwedge_{i \in \{0, \dots, n\}} (\neg\pi_0(e_i)) \wedge \bigvee_{i \in \{0, \dots, n\}} \pi_1(e_i)$
$A=B$	$a_0 \leftrightarrow b_0 \wedge a_1 \leftrightarrow b_1$	0
$X=s$	x_s	0
$s_1^e != s_2^e$	$\neg\pi_0(s_1^e = s_2^e)$	0

we add the constraints $\pi_D(s, e_1), \dots, \pi_D(s, e_n)$ to \mathcal{C}_D and we add the constraints $\pi_S(s, s'_1, e'_1), \dots, \pi_S(s, s'_m, e'_m)$ to \mathcal{C}_S .

Example 4 Reconsider the blocks from Example 1. We constructed the symbol database

$$\begin{aligned} \mathcal{D}_S = & \{(X, (string, \emptyset, \emptyset)), \\ & (B, (tristate, \emptyset, \emptyset)), \\ & (C, (tristate, \{B\}, \emptyset)), \\ & (A, (tristate, \{!B\}, \{(C, X = "s")\}))\} \end{aligned}$$

For this database we have

$$\begin{aligned} \mathcal{C}_O &= \{x_s, \neg b_0 \vee \neg b_1, \neg c_0 \vee \neg c_1, \neg a_0 \vee \neg a_1\} \\ \mathcal{C}_D &= \{\pi_D(C, B), \pi_D(A, !B)\} \\ \mathcal{C}_S &= \{\pi_S(A, C, X = "s")\} \end{aligned}$$

with

$$\begin{aligned} \pi_D(C, B) &= ((\neg b_0 \wedge \neg b_1) \rightarrow (\neg c_0 \wedge \neg c_1)) \wedge (b_1 \rightarrow \neg c_0) \\ \pi_D(A, !B) &= (b_0 \rightarrow (\neg a_0 \wedge \neg a_1)) \wedge (b_1 \rightarrow \neg a_0) \end{aligned}$$

and

$$\pi_S(A, C, X = "s") = (a_0 \wedge x_s \rightarrow c_0) \wedge (a_1 \wedge x_s \rightarrow c_0 \vee c_1)$$

3.2.3 Translation of the Choice Database

In Subsection 2.2 we have seen, that there are only two types for choices: `tristate` and `bool`. In a `bool` choice, only one of the enclosed options can be set to 2, all other options must be set to 0 (the module state 1 is not allowed at all). For each `bool` choice $(bool, \{s_1, \dots, s_n\}) \in \mathcal{D}_C$ we add the constraint

$$\bigwedge_{i \in \{1, \dots, n\}} \neg\pi_1(s_i) \wedge \bigwedge_{i \in \{1, \dots, n\}} \left(\pi_0(s_i) \rightarrow \bigwedge_{j \in \{1, \dots, n\} \setminus \{i\}} \neg\pi_0(s_j) \right)$$

to \mathcal{C}_C . In a `tristate` choice also only one option can be set to 2, but an arbitrary number of options can be set to 1 additionally. For each choice $(tristate, \{s_1, \dots, s_n\}) \in \mathcal{D}_C$ we add the constraint

$$\bigwedge_{i \in \{1, \dots, n\}} \left(\pi_0(s_i) \rightarrow \bigwedge_{j \in \{1, \dots, n\} \setminus \{i\}} \neg\pi_0(s_j) \right)$$

to \mathcal{C}_C .

Example 5 For the `tristate` choice of Example 2 we add the following constraint C to \mathcal{C}_C :

$$C = (a_0 \rightarrow \neg b_0 \wedge \neg c_0) \wedge (b_0 \rightarrow \neg a_0 \wedge \neg c_0) \wedge (c_0 \rightarrow \neg a_0 \wedge \neg b_0)$$

If the choice would be of type `bool`, we would add

$$\neg a_1 \wedge \neg b_1 \wedge \neg c_1 \wedge C.$$

4 POSSIBLE APPLICATIONS

The formula $\varphi = \bigwedge_{t \in \mathcal{C}_O} t \wedge \bigwedge_{d \in \mathcal{C}_D} d \wedge \bigwedge_{s \in \mathcal{C}_S} s \wedge \bigwedge_{c \in \mathcal{C}_C} c$ constructed in the last section describes all valid Linux kernel configurations in a single propositional formula. Now we can use all the techniques from the SAT area to process this formula. We will outline two possible fields of application of these techniques: (1) automatic detection of redundant or necessary options (2) automatic listing of all configuration variants.

4.1 Detecting redundant and necessary options

Obviously the formula φ should be satisfiable otherwise there would be no valid Linux Kernel configuration. A state-of-the-art SAT solver can compute this in < 0.1 s. However, the interesting question is if the formula is still satisfiable when some options are fixed. E.g. for $A \in \mathcal{O}_T$ one can check if $\varphi \wedge a_0$ is satisfiable. If not, then the option A can never be compiled directly into the kernel. If $\varphi \wedge a_1$ is not satisfiable, option A cannot be compiled as module. If both $\varphi \wedge a_0$ and $\varphi \wedge a_1$ are not satisfiable, we know that option A cannot be chosen in any valid configuration of the kernel. We have then detected a *redundant* option.

If $\varphi \wedge a_1$ and $\varphi \wedge \neg a_0 \wedge \neg a_1$ are both not satisfiable, we know that option A must be set to 2 in any valid configuration. We then refer to A as *necessary* option. Of course one can build more complex extensions of the formula φ for checking a specific choice of options. One can also automate these tests and search the configuration files systematically for redundant or necessary options.

Example 6 Reconsider the blocks from Example 1 with the sets \mathcal{C}_O , \mathcal{C}_D and \mathcal{C}_S given in Example 4. If we want to compile A directly into the kernel, we have to check the formula

$$\bigwedge_{t \in \mathcal{C}_O} t \wedge \bigwedge_{d \in \mathcal{C}_D} d \wedge \bigwedge_{s \in \mathcal{C}_S} s \wedge a_0$$

with a SAT solver. We get the result “false”, meaning we cannot set A to 2 in any valid configuration. If we check the formula

$$\bigwedge_{t \in \mathcal{C}_O} t \wedge \bigwedge_{d \in \mathcal{C}_D} d \wedge \bigwedge_{s \in \mathcal{C}_S} s \wedge a_1$$

we get the result “true”. So there is at least one configuration such that A can be compiled as module.

4.2 Automatic Listing of Configuration Variants

System builders or also sometimes users have very specific requirements for their Linux Kernel. E.g. one wants to know which ISDN cards work together with which network devices, meaning their

drivers can be compiled (or loaded as module) simultaneously into the kernel.

We can encode our problem as parametric SAT (PSAT) problem [7]. In contrast to a SAT problem, where the input formula is (implicitly) fully existentially quantified, we allow unquantified variables in PSAT. The output is then no longer “satisfiable” or “unsatisfiable” but a propositional formula in the free (unquantified) variables. In [7] we presented a procedure that yields a propositional formula in DNF as output. Each minterm of this DNF describes one possible assignment of the free variables such that the formula is satisfiable under this assignment. So each minterm describes one possible configuration variant of the unquantified options such that the configuration is valid.

We can use this approach and existentially quantify the propositional variables of all options we want to hide from the result. E.g if we want to know all ISDN cards playing together with a current configuration, we quantify all variables except the ones representing the configuration options for the ISDN cards. The output describes then all possible combinations of ISDN cards, that work together with the current configuration.

Example 7 *In the last example we have seen, that A cannot be compiled into the kernel but can be compiled as module. Now we want to know all possible option assignments of B such that A can be compiled as module. We do not care for assignments of the symbols X and C. We check with PSAT:*

$$\exists x_s \exists c_0 \exists c_1 \exists a_0 \exists a_1 \left(\bigwedge_{t \in \mathcal{C}_O} t \wedge \bigwedge_{d \in \mathcal{C}_D} d \wedge \bigwedge_{s \in \mathcal{C}_S} s \wedge a_1 \right)$$

PSAT yields the minterm $m = (\neg b_0 \wedge b_1)$ as result. To satisfy m we need the assignment $\{b_0 \leftarrow 0, b_1 \leftarrow 1\}$ corresponding to the option assignment $B \leftarrow 1$. So in order to compile option A as module, we need to compile B as module.

We shortly explain this result wrt. the configuration file $f(b)$: If $\alpha_O(B) = 2$, $\beta(!B) = 0$ and because of the line depends on $!B$ in the configuration block of A, 0 is then an upper bound for A and thus it cannot be set to 1 anymore. If $\alpha_O(B) = 0$, the upper bound for A is 2, so there is no problem, but since option C depends on B (surrounding if) the upper bound for C is 0 then. But if we set A to 1, we also set the lower bound of C to 1 (because of the `select` statement) and we have a conflict.

5 EXPERIMENTAL RESULTS

We have implemented all algorithms presented in Section 3. Table 3 summarizes our results for the Kernel version 2.6.33. We state the hardware architecture and the cardinality of the sets \mathcal{C}_O , \mathcal{C}_D , \mathcal{C}_S and \mathcal{C}_C . The last two columns state the number of variables and clauses of a CNF encoding of the resulting formula.

6 FURTHER RESEARCH

This paper takes the first step towards a complete propositional formalization of the LKC. As stated above we consider here only a simplified version of the `KConfig` files. Not all of the configuration options are visible to the user; Some are only visible under certain circumstances. We want to upgrade our current prototype implementation to an application allowing users to easily create, alter and complete their kernel configurations. Therefore we have to take the visibilities and default values of symbols into account.

Table 3. Experimental results with kernel version 2.6.33

architecture	$ \mathcal{C}_O $	$ \mathcal{C}_D $	$ \mathcal{C}_S $	$ \mathcal{C}_C $	# vars	# clauses
alpha	5931	36	14525	3132	12807	227610
arm	6844	64	15630	4262	14249	246368
avr32	5971	49	14609	3166	12874	227367
blackfin	6371	68	15018	3156	13597	230667
cris	3981	62	9097	1520	8772	136834
frv	5904	41	14533	3135	12769	227587
h8300	3467	33	8034	1473	7692	112149
ia64	6049	43	14775	3201	12999	230185
m32r	5912	38	14537	3141	12763	227826
m68k	5874	35	14468	3125	12793	225735
m68knommu	5950	39	14553	3135	12821	226718
microblaze	5863	38	14449	3123	12660	225132
mips	6321	53	14762	4026	13463	231300
mn10300	5916	43	14543	3143	12788	226915
parisc	5922	39	14551	3139	12783	227301
powerpc	6403	46	14997	3634	13558	234287
s390	5905	39	14468	3177	12739	225791
score	5830	38	14422	3122	12600	224786
sh	6149	51	14762	3326	13154	231231
sparc	5951	40	14571	3169	12811	226660
um/x86	2769	23	6194	1098	6051	102684
x86	6297	43	14978	3250	13402	250430
xtensa	5880	37	14521	3136	12731	226845

We want to systematically check the configuration files for redundant and necessary options as described in Subsection 4.1. Another interesting question is, how many valid configurations there are for one specific hardware architecture. We recently counted models for valid car configurations of the Daimler-Benz car lines [2]. We want to apply these methods to our new configuration problems.

As a long time goal we want to integrate these methods into the Linux build process. Some developers already expressed their wish for tools helping them writing and organizing their `KConfig` files.

7 ACKNOWLEDGEMENTS

I want to thank Martin Steghöfer, Andreas Kübler, and Wolfgang Küchlin for their valuable comments and corrections.

REFERENCES

- [1] Daniel P. Bovet and Marco Cesati, *Understanding the Linux Kernel*, O’Reilly Media, 3rd edn., nov 2005.
- [2] Andreas Kübler, Christoph Zengler, and Wolfgang Küchlin, ‘Model counting in product configuration (to appear)’, in *Proceedings of the LoCoCo 2010*, (2010).
- [3] Hendrik Post and Carsten Sinz, ‘Configuration lifting: Verification meets software configuration’, in *Proceedings of the ASE 2008*, 347–350, IEEE Computer Society, Washington, DC, USA, (2008).
- [4] Julio Sincero, Horst Schirmeier, Wolfgang Schröder-Preikschat, and Olaf Spinczyk, ‘Is the linux kernel a software product line?’, in *Proceedings of the SPLC-OSSPL 2007*, (2007).
- [5] Julio Sincero and Wolfgang Schröder-Preikschat, ‘The linux kernel configurator as a feature modeling tool’, in *Proceedings of the SPLC 2008*, eds., Steffen Thiel and Klaus Pohl, 257–260, (2008).
- [6] Carsten Sinz, Andreas Kaiser, and Wolfgang Küchlin, ‘Formal methods for the validation of automotive product configuration data’, *Artificial Intelligence for Engineering Design, Analysis, and Manufacturing*, 17(1), 75–97, (feb 2003).
- [7] Thomas Sturm and Christoph Zengler, ‘Parametric quantified SAT solving (to appear)’, in *Proceedings of the ISSAC 2010*, ACM, New York, NY, USA, (2010).