Formal Analysis of the Linux Kernel Configuration with SAT Solving

Martin Walch[1] and **Rouven Walter**[1] and **Wolfgang Küchlin**[1]

Abstract. The Linux kernel is a highly configurable software system. The aim of this paper is to develop a formal method for the analysis of the configuration space. We first develop a Linux product overview formula (L-POF), which is a Boolean formula representing the high-level configuration constraints of the kernel. Using SAT solving on this L-POF, we can then answer many questions, such as which options are possible, mandatory, or impossible for any of the processor architectures for which the kernel may be configured. Other potential applications include building a configurator or counting the number of kernel configurations. Our approach is analogous to the methods we use for automobile configuration. However, in the Linux case the configuration options (e.g. the individual device drivers) are represented by symbols in Tristate Logic, a specialized three-valued logic system with several different data types, and the configuration constraints are encoded in a somewhat arcane language. We take great care to compile the L-POF directly from the files that hold the configuration constraints in order to achieve maximum flexibility and to be able to trace results directly back to the source.

1 Introduction

Linux is a kernel for a broad range of platforms with highly versatile configurations of peripheral components. Static configuration at compile time helps to adapt to the different requirements. The central tool for configuring this is *LinuxKernelConf*, abbreviated *LKC*. The input to LKC uses a domain specific language to describe configuration constraints. A common alternative name for LKC is *Kconfig*, and they are often used interchangeably. In this document, we refer to the configuration system as LKC and we denote the language as Kconfig. The input is large and stored in files which we call *Kconfig files*. They continuously change as kernel development goes on. Automatic checks for semantic consistency on Kconfig files are desirable, but LKC has no such checks implemented.

At the Workshop on Configuration 2010 in Lisbon, Zengler and Küchlin presented an approach [11] to encode the whole Linux kernel configuration in Propositional Logic. Conceptually this work parses the Kconfig files and stores the relevant information in a database. Subsequently, this database is translated into a *product overview formula*[2], abbreviated *POF*, in Propositional Logic. While it demonstrates central ideas and shows the technical feasability of the project, it is a first prototype with only a simplified view on the Kconfig files. As a consequence, the results were of the proof-of-concept type and of very limited use for verification purposes.

The comprehensive *VAMOS project*[3] at Friedrich-Alexander-Universität Erlangen-Nuremberg has lead to numerous results and publications, including the uncovering of hundreds of real world bugs. It also analyses the Linux kernel configuration with the means of Propositional Logic, but goes much further by considering the actual effects on the kernel code and applying the tools to other projects that use LKC as well. The PhD Thesis of Reinhard Tartler [9] from 2013 gives a detailed overview over the model, the implemented tools, and most of the applications and results.

The PhD Thesis of Sarah Nadi [5] from 2014 picks up the VAMOS project and extends it to not only consider the Kconfig files and the kernel code, but to additionally take the build system into account. Even more, it extracts configuration constraints from the implementation.

In this work, we focus solely on the Kconfig files. Although the VAMOS model proves to yield good results, it does not aim at being exact and relies on parts of LKC. We present a fairly precise model and translation process into a product overview formula in Propositional Logic with the goal to account for all details that are relevant in real-world use cases. Our implementation works independently from LKC and we show the results from running it against Linux 4.0.

This work is loosely based on the paper by Zengler and Küchlin from 2010 [11] in that it picks up and uses central ideas, but elaborates on many more details.

There is no precise specification of Kconfig. So we consider what information is available in the documentation, the implementation, and the way the input language is used. Section 2 gives a rough overview over this input language.

Our translation uses several intermediate stages. First we create what we call the *Zengler Model* in Section 3. In this step we abstract from technical details of reading the configuration input and isolate the data that is relevant for our purposes.

The Zengler Model retains some parts of the input structure that have an impact on the meaning. In Section 4, we transform it into the *Attributes Model*, resolving these parts and switching from a representation that focuses on input structure to one that revolves around the constraints.

LKC uses a three-valued logic, called *Tristate Logic*, that is uncommon in academic discourse. We take a look at this logic system in Section 5 and introduce some extensions to it. We then proceed by generating a product overview formula in this extended logic from the Attributes Model, encoding the set of all valid Linux kernel con-

[1] Symbolic Computation Group, WSI Informatics, Universität Tübingen, Germany, `www-sr.informatik.uni-tuebingen.de`

[2] A POF is a single Boolean formula which captures all configuration constraints of a high-level configuration model (cf. [3]). Historically it has been introduced in [4] to capture the high-level configuration model of Mercedes-Benz, which is called “Product overview” (German: Produktübersicht).

[3] VAMOS: Variability Management in Operating Systems, `www4.cs.fau.de/Research/VAMOS/`

figurations.

Eventually, we translate the product overview formula from the three-valued logic into Propositional Logic in Section 6. With common transformation into CNF and SAT solving, we can reason about the set of valid configurations, yielding first results in Section 7.

2 The Linux Kernel Configuration System

Kconfig is a specialized input language for LKC that describes available features in terms of symbols and offers several different means to encode constraints and interdependencies. Kernel developers maintain this information in hierarchically organized Kconfig files.

Listing 1. Example Kconfig file

```
config T0
        tristate
        prompt "feature T0"
        select T1
        depends on T2 && !C0

config T1
        tristate
        prompt "feature T1" if C0

config T2
        tristate
        default y

choice
        tristate
        prompt "CHOICE 0"

        config C0
                prompt "feature C0"

        config C1
                prompt "feature C1"

endchoice

menu "submenu"
        visible if T1

config S0
        string
        default "default string"

config B0
        bool
        prompt "custom S0"

if B0

config S0
        prompt "S0"

endif

endmenu
```

LKC reads these files with a scanner and a parser that are generated using *flex*[4] and *GNU Bison*.[5] Exploring the Bison grammar is clearly beyond the scope of this document as it consists of more than 100 production rules. Suffice it to say that the grammar we use very closely ressembles the grammar of LKC.

To give an overview over the relevant parts of the configuration language, we confine ourselves to explaining an artificial example. A concise formal description of the relevant parts was done by She and Berger [7].

Listing 1 contains most of the relevant language features. The central element everything else is built around is the *configuration block*. Its purpose is to declare and to describe symbols. The keyword `config` starts a configuration block and is followed by the name of the symbol it refers to. A configuration block contains one or more lines that further specify what we call *properties* of that symbol. Each symbol must have at least one configuration block. In practice, for the majority of symbols there is exactly one configuration block for each symbol, but there may be more. In Linux 4.0, there are up to six per symbol.

Each symbol has one of the five types `tristate`, `bool`, `string`, `int`, and `hex`. In our example, the symbols *T0*, *T1*, and *T2* have the type `tristate`, the symbol *B0* has the type `bool`, and the symbol *S0* has the type `string`. We call a symbol *declared* if it has at least one configuration block and one of the five types.

On actual configuration, each symbol is assigned an unambiguous value from their respective domains. The domain of `tristate` symbols is $\{0, 1, 2\}$ and the domain of `bool` symbols is $\{0, 2\}$. The purpose of the three-valued `tristate` type is to encode the three possible activation states that apply to many features in the kernel: Turn it off (0), compile it as a runtime-loadable module (1) or compile it into the kernel (2). The `bool` type is the same except that it is missing the value 1 for the module state.

To identify the three different states in syntax, there are the three constants `n`, `m`, and `y`, which evaluate to 0, 1, and 2, respectively.

The domain of `string` symbols is the set of all valid strings. Similar to the domain of `bool` being a subset of the domain of `tristate`, the domains of `int` and `hex` symbols both are subsets of the domain of `string` symbols, in that their elements are strings that read as valid integers or hexadecimal numbers, respectively.

There are different types of dependencies that affect properties, thus most properties can be active or inactive. The symbol type is the only unconditional property. It is always active and cannot dynamically change during configuration. It is also the only property that is mandatory for every symbol. However, to allow the user to directly assign a value, the symbol needs a `prompt` property. In our example, there are several such `prompt` properties, one of them for the symbol `T0`. The `prompt` keyword is followed by a string parameter *"feature T0"*. This parameter is mandatory and shows up as short description in the configuration interface.

The symbol `T0` also has a `select` property with the argument `T1`. This property is only valid with `bool` and `tristate` symbols. The `select` property sets up a direct relation between the values of two symbols. The value of the selected symbol is always at least as high as the value of the symbol that the `select` property belongs to. So, in our example the value of `T1` is always at least as high as the value of `T0`. The documentation calls this a *reverse dependency*.

The last line of the configuration block of `T0` starts with `depends on` and is followed by an expression in Tristate Logic `T2 && !C0`.

[4] flex: The Fast Lexical Analyzer, `flex.sourceforge.net`

[5] GNU Bison, `www.gnu.org/software/bison/`

This is a dependency that applies to the whole configuration block, i.e. in our case it is a dependency to the `prompt` and to the `select` property. The actual dependency is encoded in the expression. It has no direct influence on the value of the symbol, but rather on the properties which in turn set up constraints to the value.We take a closer look at Tristate Logic and the mechanisms of dependencies in section 5.

It is possible to add a dependency to a single property by appending an expression with `if` to the respective line, like for the configuration prompt of `T1`.

A `default` property is used to non-bindingly suggest a value, but it is also used to automatically set values of symbols that have no active `prompt`. In our example, the symbol `T2` uses the trivial expression `y` as `default` value, but more complicated expressions are possible.

Our example contains a *choice block*. This language construct encloses a series of configuration blocks with `choice` and `endchoice`, constituting a set of symbols that logically exclude each other. This is useful for features that serve the same purpose and which therefore naturally cannot be simultaneously active, like e.g. the compression algorithm of the kernel image or the preemption model. Like a symbol, a *choice block* has a type, but only the two data types `bool` and `tristate` are valid. A choice block of type `bool` enforces that exactly one of the enclosed symbols is set to 2. The `tristate` type relaxes this strict constraint and allows to not assign 2 to any of the symbols, while setting an arbitrary number of symbols to 1. There is a property `optional` that even allows setting all symbols to 0. In contrast to a symbol, a choice block must have a `prompt` property. There may also be `default` properties, suggesting one of the enclosed symbols, and dependencies using `depends on`, which analogously to configuration blocks apply to the whole choice block.

A hierarchical menu can be constructed using the `menu` blocks. Its primary purpose is organizing the features to ease navigation. Just like the `prompt` property, the `menu` keyword is followed by a string that serves as user visible label for the menu. For our point of view it is important that there may be dependencies added using `depends on` that apply to the whole menu, and a second type of dependency that is set up with `visible if`. That second kind of dependency toggles only the parts that concern direct configurability by the user, but leaves all other properties unaffected.

Finally it is also possible to add dependencies to an arbitrary sequence of configuration blocks, choice blocks, and menus as long as it does not violate the logical hierarchy. This is done with the enclosing keywords `if` and `endif` with the dependency expression right behind `if`.

Our example covers those language constructs that we regard as most important for variability, but there is more: For symbols with the `int` or `hex` type, there is a `range` property that fixes a lower and an upper bound for the value. Both bounds may depend on freely changeable values of other symbols. In practice nearly all bounds have constant values and those few bounds that are variable can be manually checked for inconsistencies. From our perspective the impact of `range` properties on the variability of the overall configuration is mostly negligible in current versions of Linux. Therefore, we incorporate `range` properties in our model, but ignore them when creating a product overview formula as we do not expect any observable consequences.

Another language feature that does not occur in our example is the generic `option` line in configuration blocks, which allows extending the language with only minimal changes to the grammar and possibly gaining forward compatibility. It starts with the `option` keyword, followed by one of a few option names, and depending on that, the meaning varies and an argument may be appended. As of Linux 4.0, two different options are defined that may be relevant in our context: `modules` and `env`. The `modules` option makes only sense with a `bool` symbol, and it may be only used once in the whole configuration. It associates the corresponding symbol with the availability of support for loading modules, i.e. as a consequence, deactivating that symbol basically prohibits assigning the value 1 from the `tristate` domain. The `env` option is usually only used for `string` symbols that have no `prompt` property as it imports a value from the runtime environment of the system the configuration system is running on. Only very few symbols use this option at all and even fewer have an effect on the configuration that is worth mentioning. Still, as we want to be precise, we account for them, too.

From our example it is also not apparent how Kconfig files are hierarchically organized. It works similarly to `#include` directives of the C preprocessor: A file inclusion is done with the keyword `string` and a string argument. The string argument is treated as path and the content of the file at that path is pasted at the position of the `string` line.

Kconfig still offers more constructs that we have not mentioned, but they are irrelevant at this point.

3 Zengler Model

Linux 4.0 supports 30 different main architectures. Each architecture has its own tree of Kconfig files, but there are big overlaps across all architectures by using common files. Accordingly we want to be able to reason about the configurations across all architectures. However, capturing several architectures in one data structure without losing precision is very complicated as this requires keeping track of which files are included for each architecture and in which order they are read. Therefore, we create a separate model for each architecture and consider them together if needed.

By considering one fixed architecture, we can deduce the full inclusion hierarchy of Kconfig files for that specific architecture. Accordingly, our parser moves through the file hierarchy in the same way as LKC does, without the need to store which files we include. However, it produces different data structures. We create two databases, a *symbol database* $\mathcal{D}_S$, and a *choice database* $\mathcal{D}_C$. They are heavily extended versions of the databases created by Zengler and Küchlin [11], hence we call these two databases the Zengler Model.

The symbol database contains *symbol descriptors*. Each configuration block corresponds to a symbol descriptor. They are grouped by the symbol they belong to. So, our *symbol database* is a set of pairs $\mathcal{D}_S = (s, \overrightarrow{\mathbf{desc}})$ where s is a symbol and $\overrightarrow{\mathbf{desc}}$ is a list of symbol descriptors $\mathbf{desc}$ with

$$\begin{aligned}
\mathbf{desc} &= (t, \overrightarrow{\mathbf{pmpt}}, \overrightarrow{\mathbf{sel}}, \overrightarrow{\mathbf{def}}, \overrightarrow{\mathbf{range}}, \mathbf{dep}, \mathbf{opt}, k_{desc})\\
\mathbf{pmpt} &= (e_p, k_p)\\
\mathbf{sel} &= (s_s, e_s, k_s)\\
\mathbf{def} &= (e_v, e_d, k_d)\\
\mathbf{range} &= (e_l^s, e_h^s, e_r, k_r)\\
\mathbf{dep} &= (\overrightarrow{e_{if}}, \overrightarrow{e_{dep}}, \overrightarrow{e_{vis}})\\
\mathbf{opt} &= (b_{env}, b_{mod}).
\end{aligned}$$

The type t is one of the five types or it is a special type “unknown” if the configuration block has no type property. It suffices if only one descriptor holds a regular type.

The properties `prompt`, `select`, `default`, and `range` appear in the descriptor as the tuples $\mathbf{pmpt}$, $\mathbf{sel}$, $\mathbf{def}$, and $\mathbf{range}$, respectively. As the input order plays a role, each of them and the descriptor contain unique indices k_{desc}, k_p, k_s, k_d, and k_r that are ascending in input order.

The string argument of a `prompt` property is unimportant for the configuration logic, so we only store the expression of the optional `if` dependency e_p. If there is no such dependency, we store a special symbol as placeholder for an empty expression. Analogously, the optional `if` dependencies for `select`, `default`, and `range` properties are stored as e_s, e_d, and e_r.

The target of a `select` property is a single symbol e_s. A `default` value may be the result of a non-trivial expression e_v. The lower bound of a `range` property e_l^s may be either another symbol or a constant value. The same goes for the upper bound e_h^s.

We do not store any menus or enclosing ifs explicitely. Instead we propagate their dependencies to the contained descriptors and store the dependency expressions in lists that distinguish between the different types. The dependencies of surrounding ifs are stored in $\overrightarrow{e_{if}}$. Those of `depends on` dependencies add up to $\overrightarrow{e_{dep}}$, including any such dependencies contained in the configuration block under consideration. The third list $\overrightarrow{e_{vis}}$ contains the `visible if` dependencies from menus.

The pair of bits $\mathbf{opt}$ indicates if there is a `module` or `env` option. Even though `env` carries an argument in the configuration block, we only care whether it is present.

Our choice database $\mathcal{D}_C$ is a set of tuples which we call *choice descriptors*:

$$(t, b_{opt}, \overrightarrow{\mathbf{pmpt}}, \mathbf{dep}, \overrightarrow{k_{desc}}, k_c)$$

Each choice block translates into a choice descriptor. The type t is one of `bool`, `tristate` or "unknown". The bit b_{opt} indicates whether the choice block carries the optional flag. Just like in a symbol descriptor, there is a list of $\mathbf{pmpt}$ tuples, dependencies lists $\mathbf{dep}$, and an index k_c. Rather than storing the symbol names of the contained symbol descriptors, we create a list of their indices $\overrightarrow{k_{desc}}$.

In contrast to the symbol descriptors, the `default` properties of choice blocks have no influence on variability and we ignore them.

4 Attributes Model

Although the Zengler Model is an abstraction from the underlying Kconfig files, storing the relevant information in a normalized way, its focus lies on the input structure: The symbol database holds individual symbol descriptors $\mathbf{desc}$ and lists of dependencies $\mathbf{dep}$ reflecting artifacts from the input structure. Furthermore, we have yet to determine the final types of the symbols and choices and which symbols a choice includes as this is not trivial in all cases.

In the next step, we resolve all this and create a model that focuses on the effective impact of the descriptor properties on the symbols. For a better distinction from the properties of the descriptors in the Zengler Model, we refer to their resulting equivalents in the new model as attributes. Therefore we call the new model the *Attributes Model*.

First we associate the attributes directly with the dependencies from $\mathbf{dep}$ that control them. As already mentioned in Section 2, the dependencies in $\overrightarrow{e_{vis}}$ do not affect all types of attributes. In fact, they control only our `prompt` attributes a^P. We define them as

$$a^P = (E_p, k_p) = (\overrightarrow{e_{if}} \cup \overrightarrow{e_{dep}} \cup \overrightarrow{e_{vis}} \cup e_p, k_p)$$

We write E to denote sets of expressions. The other attributes are only affected by the other dependencies, hence we define them as follows:

$$\begin{aligned} a^S = (E_s, s, k_s) &\quad = \quad (\overrightarrow{e_{if}} \cup \overrightarrow{e_{dep}} \cup e_s, s, k_s) \\ a^D = (E_d, e_v, k_d) &\quad = \quad (\overrightarrow{e_{if}} \cup \overrightarrow{e_{dep}} \cup e_d, e_v, k_d) \\ a^R = (E_r, e_l^s, e_h^s, k_r) &\quad = \quad (\overrightarrow{e_{if}} \cup \overrightarrow{e_{dep}} \cup e_r, e_l^s, e_h^s, k_r) \end{aligned}$$

With these definitions we collect for each symbol s the `prompt` attributes a^P in a set $P(s)$, the `default` attributes a^D in a set $D(s)$, and the `range` attributes a^R in a set $R(s)$. We also create a set $S(s)$ for each symbol, but note that the symbol in the definition of a^S is the symbol of the descriptor that *contains* the property and not the symbol s_s that is the `select` target. Accordingly we add each a^S to the set of the target symbol. If there is no descriptor of the target symbol we discard the attribute.

To determine the symbol type that results from the types in the corresponding descriptors, we take the type of the symbol descriptor with the lowest k_{desc} that is not "unknown". If there is no such descriptor we also set the type of the symbol to "unkown".

Finally we concatenate the $\mathbf{opt}$ bitvectors component-wise with a logical or and store it with the symbol.

We do not need $\mathbf{dep}$ and k_{desc} anymore and do not transfer them into the Attributes Model.

Next we transform the choice descriptors from the Zengler Model into what we call *choice groups*. Like for symbols, we determine the type of the choice group. If t in the choice descriptor is `bool` or `tristate`, then this is also the type of the choice group. However, it is a frequently used feature to skip the explicit type declaration in choice blocks, resulting in choice descriptors with the special type "unknown". In that case the type is taken from the first symbol that is enclosed in the choice descriptor and has a regular type. If any symbol in the choice descriptor is lacking a regular type then it inherits the type of the choice group.

Now that we have completed the type resolution for choice groups, we determine which symbols are part of the choice group. This is not trivial as not all symbols that correspond to symbol descriptors in the choice descriptor are necessarily transferred. Symbols can be moved into their own submenu by depending on a symbol that is immediately above, excluding them from the choice group. We actually see this constellation intentionally used in Linux 4.0 and have to process it adequately. LKC involves an extensive logic to determine whether to move a symbol into a submenu, but a simple heuristic suffices to correctly capture any real-world case.

As LKC ignores the attributes $S(s)$ and $D(s)$ of all symbols of the choice group, we clear them if they have any content.

To complete the choice group, we generate a `prompt` attribute a^P the same way we do from a symbol descriptor and we keep the `optional` bit b_{opt}.

Our new databases $\mathcal{D}'_S$ and $\mathcal{D}'_C$ are now

$$\begin{aligned} \mathcal{D}'_S &= \overrightarrow{(s, t, P(s), S(s), D(s), R(s), \mathbf{opt})} \\ \mathcal{D}'_C &= \overrightarrow{(t, b_{opt}, a^P, \overrightarrow{s})} \end{aligned}$$

5 Tristate* Logic and POF

Our next step is to translate the Attributes Model into a coherent product overview formula. While our goal is to arrive at a POF in Propositional Logic, we prefer to split this translation into two steps. First we define Tristate* Logic, an extension to Tristate Logic that we specifically design to create an initial POF. Then we translate the POF from Tristate* Logic into Propositional Logic in Section 6.

The following grammar describes the general syntax of expressions in Tristate Logic:

$$\begin{array}{lcl} \langle \textit{expression} \rangle & \rightarrow & \langle \textit{expression symbol} \rangle \\ & \mid & \langle \textit{expression symbol} \rangle \text{`}\texttt{=}\text{'} \langle \textit{expression symbol} \rangle \\ & \mid & \langle \textit{expression symbol} \rangle \text{`}\texttt{!=}\text{'} \langle \textit{expression symbol} \rangle \\ & \mid & \text{`}\texttt{(}\text{'}\ \langle \textit{expression} \rangle \text{`}\texttt{)}\text{'} \\ & \mid & \text{`}\texttt{!}\text{'} \langle \textit{expression} \rangle \\ & \mid & \langle \textit{expression} \rangle \text{`}\texttt{\&\&}\text{'} \langle \textit{expression} \rangle \\ & \mid & \langle \textit{expression} \rangle \text{`}\texttt{||}\text{'} \langle \textit{expression} \rangle \end{array}$$

$$\begin{array}{lcl} \langle \textit{expression symbol} \rangle & \rightarrow & \text{symbol name} \\ & \mid & \text{constant value} \end{array}$$

There are five operators and there are parentheses to override operator precedences. We distinguish two groups of operators: The `tristate` operators `!`, `&&`, and `||`, and the `string` operators `=` and `!=`. Both groups operate on their respective domains.

The `tristate` operators and their domain form a three-valued logic like those of Łukasiewicz and Kleene. A comprehensive overview of these logics has been done by Gabbay and Woods [2].

In fact, the three base operators `!`, `&&`, and `||` correspond to the operators $\neg$, $\wedge$ and $\vee$ in K_3 and Ł$_3$ from Kleene and Łukasiewicz. This observation has already been made in 2010 by Berger et al. [1]. However, Tristate Logic itself is not expressive enough for a full POF. We need to extend it for our purposes.

The nature of our dependencies is Boolean and not three-valued, because either they are met or they are not. There is no third value like the module state in Tristate Logic. Hence, when extending Tristate Logic to allow encoding all constraints in a POF, we look for operators that operate on `tristate` values, but only yield two different values. Such operators are not typically part of K_3 or Ł$_3$ and hence we exclude these logics from further consideration.

Instead we introduce our own new operators $\Rightarrow$ and $\Leftrightarrow$ and define their semantics as shown in Table 1 and Table 2.

Table 1. Truth table of tristate* operator $\Leftrightarrow$

$\Leftrightarrow$	0	1	2
0	2	0	0
1	0	2	0
2	0	0	2

Table 2. Truth table of tristate* operator $\Rightarrow$

$\Rightarrow$	0	1	2
0	2	2	2
1	0	2	2
2	0	0	2

They express equality and "less than or equal to" inequality on `tristate` values. We call this extended version of Tristate Logic the *Tristate* Logic*.

Note that Tristate* Logic also contains the `string` operators `=` and `!=`. They compare values from the `string` domain and also yield one of the two values 0 and 2. Mixing `tristate` and `string` operators and symbols in expressions is actually a feature, leading to frequent conversions between the two domains.

This may become quite complex. Consider the short expression `A=B`. If both, `A` and `B`, are declared `string` symbols, their values are compared, yielding 2 if they are exactly identical and 0 otherwise. However, if `A` is a `tristate` symbol and `B` is undeclared, then the values 0, 1, 2 of `A` are interpreted as the strings "*n*", "*m*", and "*y*" and *B* is interpreted as the `string` "*B*", i.e. in this case a string comparison is done on these letters, always yielding 0. We take all these details into account when producing our POF in Tristate* Logic.

Encoding the constraints of `bool` and `tristate` symbols that originate from their respective attributes, works in an indirect way: For each of these symbols, we add two auxiliary variables which represent a lower and an upper limit to the value of the symbol in consideration, and, using the $\Rightarrow$ operator, we append the constraint that the value of the symbol must not exceed the values set up by the auxiliary variables.

This is in contrast to encoding the choice groups: We encode the exclusiveness of symbols with expressions that directly relate to the symbols instead of their associated auxiliary variables.

Finally, we also take account of the mode without module support by adding two different subformulae for `tristate` symbols and `tristate` choice groups which depend on the symbol that has the b_{mod} bit set.

6 POF in Propositional Logic

Translating the POF from Tristate* Logic into Propositional Logic is mostly straightforward. We encode each symbol with the type `tristate` or `bool` and all auxiliary variables using two variables in Propositional Logic. The three values of the `tristate` domain correspond to three states that the two Propositional Variables can encode. We explicitely prohibit the fourth possible state.

Translating `tristate` constants, symbols, and operators works mostly the same way as in the paper by Zengler and Küchlin [11] with the two projections π_0 and π_1 as listed in table 3. The three constants `y`, `m`, and `n` map to corresponding combinations of $\top$ and $\bot$, and similarly each `tristate` symbol A^T maps to two propositional variables $p_0(A^T)$ and $p_1(A^T)$. Mapping the unary operator `!` is simple, but the entries for the binary operators operators `&&` and `||` stick out by being generalized to n-ary operators. This is an optimization to keep the size of the formulae smaller for long chains of the same operands.

Of course, the Tristate* operators $\Leftrightarrow$ and $\Rightarrow$ also have to be translated into Propositional Logic. However, due to our definition of these operands this is trivial for π_1 as they yield only 0 and 2, and π_0 is intuitive. We use these two translations to also cover the usage of the `string` operator `=` with `tristate` operands.

Finding a Propositional encoding for `string` symbols requires more work, because Propositional Logic is not well suited for encoding arbitrary strings. For each `string` symbol, we iterate over our POF in Tristate* Logic and collect all occurrences in expressions. In our method we consider only cases of equal strings and resulting other equalities and inequalities and define new propositional variables $P_{X^S \leftarrow \mathfrak{s}}$ to encode that the `string` variable X^S has the value $\mathfrak{s}$ and accordingly $(X^S = Y^S)$ to encode if X^S and Y^S have the same value. This suffices, because for the configuration space the actual value of a `string` is not important. The same goes for `int` and `hex` symbols. Finally we use the mappings for `=` and `!` to define the mappings for `!=`.

With these mappings we create our POF in Propositional Logic.

As it encodes the configuration space of the Linux kernel, we call it *L-POF*.

To use modern SAT solvers we do one final step: Our L-POF is not in CNF. So we use the Warthog Logic Framework[6] to generate an equisatisfiable formula using the Plaisted-Greenbaum algorithm [6].

7 Results

Our implementation consists of more than 7,000 lines of Java Code, using the features of Java SE 6. We measured execution times on a computer with an Intel Core2Quad Q6600 using the Java implementation of the IcedTea project in version 6.1.13.7 on Gentoo Linux. Creating the L-POFs from the Kconfig files for all 30 architectures, takes on average less than 3 seconds per architecture. Transformation into CNF varies between 20 and 42 seconds, depending on the architecture.

To give a rough impression of size of the configuration space, we show the distribution of symbol types for each architecture in Table 4. Across all architectures, the vast majority of symbols has one of the two types `tristate` and `bool` which has a major impact on the form of the formula.

Table 4. Distribution of symbol types in Linux 4.0

arch	tristate	bool	string	int	hex	total
alpha	6317	3402	34	192	27	9972
arc	6293	3352	36	194	29	9904
arm	6371	4724	38	209	41	11383
arm64	6366	3511	36	195	27	10135
avr32	6360	3462	36	193	30	10081
blackfin	6380	3676	36	702	43	10837
c6x	6292	3293	35	189	28	9837
cris	6345	3436	45	254	78	10158
frv	6316	3378	35	192	28	9949
hexagon	6292	3292	35	190	27	9836
ia64	6400	3519	36	195	27	10177
m32r	6324	3361	34	194	31	9944
m68k	6322	3453	35	192	37	10039
metag	6293	3363	37	193	28	9914
microblaze	6294	3335	37	195	32	9893
mips	6391	3964	36	198	28	10617
mn10300	6316	3426	35	199	33	10009
nios2	6292	3303	36	191	36	9858
openrisc	6292	3292	36	188	27	9835
parisc	6325	3384	34	191	27	9961
powerpc	6404	3933	37	205	40	10619
s390	6313	3453	35	195	27	10023
score	6292	3292	35	188	28	9835
sh	6374	3640	37	198	34	10283
sparc	6370	3444	37	193	30	10074
tile	6306	3393	36	191	29	9955
um	6297	3362	38	193	27	9917
unicore32	6363	3425	36	191	27	10042
x86	6420	3781	40	206	32	10479
xtensa	6326	3379	41	194	29	9969

Table 5 gives a rough overview of how big the formulae grow. It comes to no surprise that the translation from Tristate* Logic to Propositional Logic and the transformation into CNF using the Plaisted-Greenbaum algorithm both significantly increase the size.

The fact that there are more regular variables in the POF in Tristate* Logic than there are declared symbols on the respective architecture comes from the fact that it still contains expressions with undeclared symbols. They are cleaned in the translation process into Propositional Logic. Each of our formulae in CNF contains roughly one million Propositional variables. For some very basics analyses with SAT solving, we use PicoSAT.[7] Processing the CNF formula without any additional clauses takes PicoSAT between 6 and 10 seconds. We search for redundant `bool` or `tristate` symbols, i.e. symbols that can never be active, and for symbols that are necessary, i.e. it is not possible to fully deactivate them. We find out if a symbol is redundant by assuming that one of the two corresponding Propositional variables is true. If this is not satisfiable, then the symbol is always inactive. Vice versa by assuming that both Propositional variables are false we find out whether a symbol must always be active. More than 99.8% of the individual tests run in less than three seconds.

Table 6. Redundant or necessary symbols in Linux 4.0

arch	redundant (⇔ `n`)	necessary (not ⇔ `n`)
alpha	3223	55
arc	4263	63
arm	1691	75
arm64	3159	135
avr32	4522	54
blackfin	4430	47
c6x	4644	42
cris	3785	34
frv	3700	37
hexagon	4625	45
ia64	3454	74
m32r	4937	32
m68k	3741	32
metag	4301	67
microblaze	3251	71
mips	2773	64
mn10300	3702	39
nios2	4428	45
openrisc	4424	56
parisc	3499	51
powerpc	2652	94
s390	4149	107
score	7068	36
sh	3297	67
sparc	3201	51
tile	3633	57
um	7368	28
unicore32	3541	58
x86	2301	138
xtensa	3248	43
globally	135	1

Table 6 shows the results of these tests. The reason for the high numbers of features that cannot be activated on each architecture is that there are many features that run only on few architectures, but the corresponding Kconfig files are common for all architectures. Symbols that cannot be deactivated on the other side are less, but still a lot. They are symbols that are intentionally not deactivatable and in general they do not represent selectable features, but basic aspects of an architecture.

[6] Warthog Logic Framework: github.com/warthog-logic/warthog

[7] PicoSAT: fmv.jku.at/picosat/

Table 3. Translation rules for tristate* operators

e'	$\pi_0\left(e'\right)$	$\pi_1\left(e'\right)$
$\mathtt{y}$	$\top$	$\bot$
$\mathtt{m}$	$\bot$	$\top$
$\mathtt{n}$	$\bot$	$\bot$
A^T	$p_0(A^T)$	$p_1(A^T)$
$\mathtt{!}\,e$	$\neg\pi_0\left(e\right)\wedge\neg\pi_1\left(e\right)$	$\pi_1\left(e\right)$
$e_0\ \mathtt{\&\&}\cdots\mathtt{\&\&}\ e_n$	$\pi_0\left(e_0\right)\wedge\cdots\wedge\pi_0\left(e_n\right)$	$\bigwedge_{i\in\{0,\ldots,n\}}\left(\pi_0\left(e_i\right)\vee\pi_1\left(e_i\right)\right)\wedge\bigvee_{i\in\{0,\ldots,n\}}\pi_1\left(e_i\right)$
$e_0\ \Vert\cdots\Vert\ e_n$	$\pi_0\left(e_0\right)\vee\cdots\vee\pi_0\left(e_n\right)$	$\bigwedge_{i\in\{0,\ldots,n\}}(\neg\pi_0\left(e_i\right))\wedge\bigvee_{i\in\{0,\ldots,n\}}\pi_1\left(e_i\right)$
$e_1\Leftrightarrow e_2$	$\left(\pi_0\left(e_1\right)\leftrightarrow\pi_0\left(e_2\right)\right)\wedge\left(\pi_1\left(e_1\right)\leftrightarrow\pi_1\left(e_2\right)\right)$	$\bot$
$e_1\Rightarrow e_2$	$\pi_0\left(e_2\right)\vee\neg\pi_0\left(e_1\right)\wedge\left(\neg\pi_1\left(e_1\right)\vee\pi_1\left(e_2\right)\right)$	$\bot$
$A^T=t$	$\pi_0\left(A^T\Leftrightarrow t\right)$	$\bot$
$A^T=B^T$	$\pi_0\left(A^T\Leftrightarrow B^T\right)$	$\bot$
$X^S=\mathfrak{s}$	$P_{X^S\leftarrow\mathfrak{s}}$	$\bot$
$X^S=Y^S$	$P_{X^S=Y^S}$	$\bot$
$e_1^s\ \mathtt{!}=e_2^s$	$\neg\pi_0\left(e_1^s=e_2^s\right)$	$\bot$

To get meaningful results, we merge the results. We collect all symbols that do not have any architecture that allows activating, and we collect all symbols that are declared across all architectures, but may not be deactivated on any of them. These numbers are in the line globally. For most of the 135 symbols that cannot be activated, this is actually the intention of the maintainer. The one `tristate` symbol that can never be deactivated may intentionally only alternate between the two other possible states.

These results do not surprise as similar tests were also done in the context of the VAMOS project and hence many problems have already been uncovered and solved.

8 Conclusion

Our approach successfully leads to a precise product overview formula. Although Linux has reached more than 10.000 features, our implementation quickly creates the L-POF, a product overview formula of the Linux kernel in Propositional Logic. Despite its considerable size, fast SAT solving on the formula is in general possible. If needed there is still much room for optimization.

A future research direction could be to investigate the possibility of further verification tests beyond redundant and necessary symbols, e.g. specialized verification tests for a choice block analogously to verification tests for positions of a Bill of Materials as it is done in automotive configuration [8].

As we do not use parts of LKC in our implementation, we now have the option to extend our program to do fine-grained tests considering individual lines in Kconfig files and easily locate the origin of inconsistencies.

Another interesting topic is re-configuration. Although LKC does not permit invalid configurations by disabling options during the configuration process, it might be useful for users to select all wanted options first without caring about the validation. Afterwards, if the configuration is invalid, we can re-configure the selections of the user in an optimal way, i.e. by solving a MaxSAT optimization problem.

The reverse is also imaginable: If the selections of a user lead to an invalid configuration, the user might want to know which configuration constraints have to change in order to make the configuration valid. Thus, we want to find the minimal set of constraints to remove or change. Such MaxSAT re-configuration use cases have been described in the context of automotive configuration in [10] and could be adopted for the Linux kernel configuration.

Table 5. Sizes of POFs for Linux 4.0

arch	regular variables in POF in Tristate* Logic	auxiliary variables in POF in Tristate* Logic	total number of variables in POF inTristate* Logic	variables in L-POF	variables in CNF	clauses in CNF
alpha	10673	48845	59518	117417	996839	1812856
arc	10602	48481	59083	116538	954428	1718683
arm	11976	55760	67736	134270	1299812	2849653
arm64	10824	49640	60464	119333	1007563	1828031
avr32	10793	49366	60159	118671	1001036	1816464
blackfin	11539	51058	62597	123096	1026513	1861942
c6x	10548	48174	58722	115799	949031	1708805
cris	10867	49279	60146	118559	999006	1812066
frv	10666	48722	59388	117126	990489	1797565
hexagon	10540	48169	58709	115795	981542	1781667
ia64	10866	49850	60716	119837	1010856	1834072
m32r	10643	48681	59324	117039	993691	1804508
m68k	10717	49136	59853	118115	1008800	1836987
metag	10605	48535	59140	116671	955382	1720572
microblaze	10591	48406	58997	116370	985108	1788040
mips	11249	52034	63283	125090	1048937	1909971
mn10300	10721	48974	59695	117738	994158	1804015
nios2	10566	48235	58801	115951	950035	1710610
openrisc	10544	48168	58712	115783	949121	1709044
parisc	10658	48794	59452	117297	996712	1810111
powerpc	11247	51964	63211	124935	1055822	1917736
s390	10699	49084	59783	117997	998901	1813210
score	10539	48168	58707	115783	949788	1710461
sh	10955	50336	61291	121037	1020515	1854779
sparc	10774	49327	60101	118582	1004762	1823946
tile	10655	48748	59403	117195	990749	1798113
um	10606	48550	59156	116723	998908	1821791
unicore32	10753	49191	59944	118246	998333	1811566
x86	11135	51280	62415	123314	1051478	1913811
xtensa	10674	48786	59460	117278	993296	1802906

References

[1] Thorsten Berger, Steven She, Rafael Lotufo, Andrzej Wąsowski, and Krzysztof Czarnecki, 'Variability Modeling in the Real: A Perspective from the Operating Systems Domain', in *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, ASE '10, pp. 73–82, New York, NY, USA, (2010). ACM.

[2] *The Many Valued and Nonmonotonic Turn in Logic, Volume 8 (Handbook of the History of Logic)*, eds., Dov M. Gabbay and John Woods, North Holland, 1 edn., 8 2007.

[3] Albert Haag, 'Sales configuration in business processes', *IEEE Intelligent Systems*, **13**(4), 78–85, (July 1998).

[4] Wolfgang Küchlin and Carsten Sinz, 'Proving consistency assertions for automotive product data management', *J. Automated Reasoning*, **24**(1–2), 145–163, (February 2000).

[5] Sarah Nadi, *Variability Anomalies in Software Product Lines*, Ph.D. dissertation, University of Waterloo, 2014.

[6] David A. Plaisted and Steven Greenbaum, 'A structure-preserving clause form translation', *Journal of Symbolic Computation*, **2**(3), 293–304, (September 1986).

[7] Steven She and Thorsten Berger, 'Formal semantics of the kconfig language', *Technical note, University of Waterloo*, 24, (2010).

[8] 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, (January 2003). Special issue on configuration.

[9] Reinhard Tartler, *Mastering Variability Challenges in Linux and Related Highly-Configurable System Software*, Ph.D. dissertation, Friedrich-Alexander-Universität Erlangen-Nürnberg, 2013.

[10] Rouven Walter, Christoph Zengler, and Wolfgang Küchlin, 'Applications of MaxSAT in automotive configuration', in *Proceedings of the 15th International Configuration Workshop*, eds., Michel Aldanondo and Andreas Falkner, pp. 21–28, Vienna, Austria, (August 2013).

[11] Christoph Zengler and Wolfgang Küchlin, 'Encoding the Linux Kernel Configuration in Propositional Logic', in *Proceedings of the 19th European Conference on Artificial Intelligence (ECAI 2010) Workshop on Configuration*, eds., Lothar Hotz and Alois Haselböck, pp. 51–56, (2010).