Formal Analysis of the Linux Kernel Configuration with SAT Solving

Martin Walch\textsuperscript{1} and Rouven Walter\textsuperscript{1} and Wolfgang Küchlin\textsuperscript{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 \texttt{LinuxKernelConf}, abbreviated LKC. The input to LKC uses a domain-specific language to describe configuration constraints. A common alternative name for LKC is \texttt{Kconfig}, and they are often used interchangeably. In this document, we refer to the configuration system as LKC and we denote the language as \texttt{Kconfig}. 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.

The Zengler Model retains some parts of the input structure that 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\textsuperscript{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-

\textsuperscript{1}Symbolic Computation Group, WSI Informatics, Universität Tübingen, Germany, www-sr.informatik.uni-tuebingen.de
\textsuperscript{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).
\textsuperscript{3}VAMOS: Variability Management in Operating Systems, www4.cs.fau.de/Research/VAMOS/
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

```kconfig
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 and GNU Bison. 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 resembles 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 _ 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 $D_{S}$, and a choice database $D_{C}$. They are heavily extended versions of the databases created by Zengler and Kühlin [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 $D_{S} = \{s, \text{desc}\}$ where $s$ is a symbol and desc is a list of symbol descriptors desc with

\[
\text{desc} = (t, \text{pmpt}, \text{se}, \text{def}, \text{range}, \text{dep}, \text{opt}, k_{exec})
\]

\[
\text{pmpt} = (e_{p}, k_{p})
\]

\[
\text{se} = (s_{e}, e_{s}, k_{s})
\]

\[
\text{def} = (e_{d}, e_{r}, k_{d})
\]

\[
\text{range} = (t_{l}, t_{r}, e_{r}, k_{r})
\]

\[
\text{dep} = (e_{d}, e_{dep}, e_{r}, k_{dep})
\]

\[
\text{opt} = (k_{exec}, k_{mod})
\]

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.

---

Juha Tiihonen, Andreas Falkner and Tomas Axling, Editors
Proceedings of the 17th International Configuration Workshop
September 10-11, 2015, Vienna, Austria
The properties prompt, select, default, and range appear in the descriptor as the tuples pmpt, sel, def, and range, respectively. As the input order plays a role, each of them and the descriptor contain unique indices \( k_{pmpt}, k_{sel}, k_{def}, \) and \( k_{range} \), 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 \( e_d \). The lower bound of a range property \( e_r \) may be either another symbol or a constant value. The same goes for the upper bound \( e_u \).

We do not store any menus or enclosing ifs explicitly. 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 \( \epsilon_{\pm} \).

Those of depends on dependencies add up to \( \epsilon_{\pm} \), including any such dependencies contained in the configuration block under consideration. The third list \( \epsilon_{\pm} \) contains the visible if dependencies from menus.

The pair of bits \( \text{opt} \) 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 \( D_C \) is a set of tuples which we call choice descriptors:

\[
(t, b_{\text{opt}}, \text{pmpt}, \text{dep}, k_{\text{pmpt}}, k_{\text{sel}}, k_{\text{def}}, k_{\text{range}}) = (t, b_{\text{opt}}, (s, k_s), (e_d, k_d), (e_r, k_r))
\]

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

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 \( D_s \) and lists of dependencies \( \text{dep} \) 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 \( \text{dep} \) that control them. As already mentioned in Section 2, the dependencies in \( \epsilon_{\pm} \) do not affect all types of attributes. In fact, they control only our prompt attributes \( a^P \). We define them as

\[
a^P = (E_p, k_p) = (\epsilon_{P} \cup \epsilon_{\text{dep}} \cup \epsilon_{\text{opt}} \cup e_p, k_p)
\]

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

\[
a^S = (E_s, s, k_s) = (\epsilon_{P} \cup \epsilon_{\text{dep}} \cup e_s, s, k_s)
\]

\[
a^D = (E_d, e_d, k_d) = (\epsilon_{P} \cup \epsilon_{\text{dep}} \cup e_d, e_d, k_d)
\]

\[
a^R = (E_r, e_r, k_r) = (\epsilon_{P} \cup \epsilon_{\text{dep}} \cup e_r, e_r, k_r)
\]

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 \( s \) 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 \( k_{\text{pmpt}} \) that is not "unknown". If there is no such descriptor we also set the type of the symbol to "unknown".

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

We do not need \( \text{dep} \) and \( k_{\text{pmpt}} \) 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 \( b_{\text{opt}} \).

Our new databases \( D'_S \) and \( D'_C \) are now

\[
D'_S = (s, t, P(s), S(s), D(s), R(s), \text{opt})
\]

\[
D'_C = (t, b_{\text{opt}}, a^P, \overline{a^S})
\]

### 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 expres-
sions in Tristate Logic:

\[
\text{expression} \quad \rightarrow \quad (\text{expression symbol}) \mid (\text{expression symbol}) = \mid (\text{expression symbol}) \neq \mid (\text{expression symbol}) \land \mid (\text{expression symbol}) \lor \mid (\text{expression})' \mid (\text{expression})' \mid (\text{expression})'\' \mid (\text{expression})'\'\' \mid (\text{expression})'\'\'\' \\
\text{expression symbol} \quad \rightarrow \quad \text{symbol name} \mid \text{constant value}
\]

There are five operators and there are parentheses to override op-
erator precedences. We distinguish two groups of operators: The
\text{tristate} operators \(!, \&\&\text{, and } ||\), and the \text{string} operators =
and \neq. Both groups operate on their respective domains.

The \text{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 \(!, \&\&\text{, and } ||\) correspond to the
operators \(\neg\), \(\land\) and \(\lor\) in \(K_3\) and \(L_3\) 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 Tri-
state Logic to allow encoding all constraints in a POF, we look for
operators that operate on \text{tristate} values, but only yield two dif-
ferent values. Such operators are not typically part of \(K_3\) or \(L_3\)
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>
<thead>
<tr>
<th>(\Rightarrow)</th>
<th>0</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>(\Leftrightarrow)</th>
<th>0</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>2</td>
</tr>
</tbody>
</table>

They express equality and “less than or equal to” inequality on
\text{tristate} values. We call this extended version of Tristate Logic
the \text{Tristate* Logic}.

Note that Tristate* Logic also contains the \text{string} operators =
and \(!=\). They compare values from the \text{string} domain and also
yield one of the two values 0 and 2. Mixing \text{tristate} and \text{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
\(A \Rightarrow B\). If both, \(A\) and \(B\), are declared \text{string} symbols, their values
are compared, yielding 2 if they are exactly identical and 0 otherwise.
However, if \(A\) is a \text{tristate} symbol and \(B\) is undeclared, then the
values 0, 1, 2 of \(A\) are interpreted as the strings “a”, “m”, and “y”
and \(B\) is interpreted as the string “B”, i.e. in this case a string compar-
ison 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 \text{bool} and \text{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 take account of the mode without module support
by adding two different subformulae for \text{tristate} symbols and
\text{tristate} choice groups which depend on the symbol that has the
\(b_{marg}\) bit set.

\section{POF in Propositional Logic}

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

Translating \text{tristate} constants, symbols, and operators works
mostly the same way as in the paper by Zengler and Küchlin [11]
with the two projections \(\pi_0\) and \(\pi_1\) as listed in table 3. The three
constants \(y, m, n\) map to corresponding combinations of \(\top\) and
\(\bot\), and similarly each \text{tristate} symbol \(A^T\) maps to two proposi-
tional 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 opti-
mization to keep the size of the formulae smaller for long chains of
the same operands.

Of course, the \text{Tristate} operators \(\Rightarrow\) and \(\Leftrightarrow\) also have to be
translated into Propositional Logic. However, due to our definition
of these operands this is trivial for \(\pi_1\) as they yield only 0 and 2, and
\(\pi_0\) is intuitive. We use these two translations to also cover the usage
of the \text{string} operator = with \text{tristate} operands.

Finding a Propositional encoding for \text{string} symbols requires
more work, because Propositional Logic is not well suited for encoding
arbitrary strings. For each \text{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 result-
ing other equalities and inequalities and define new propositional vari-
bles \(P_{X=S}\) to encode that the \text{string} variable \(X^S\) has the value \(s\)
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 \text{string} is not important. The same goes for \text{int}
and \text{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

<table>
<thead>
<tr>
<th>arch</th>
<th>tristate</th>
<th>bool</th>
<th>string</th>
<th>int</th>
<th>hex</th>
<th>total</th>
</tr>
</thead>
<tbody>
<tr>
<td>alpha</td>
<td>6317</td>
<td>3402</td>
<td>34</td>
<td>192</td>
<td>27</td>
<td>9972</td>
</tr>
<tr>
<td>arc</td>
<td>6293</td>
<td>3352</td>
<td>36</td>
<td>194</td>
<td>29</td>
<td>9904</td>
</tr>
<tr>
<td>arm</td>
<td>6371</td>
<td>4724</td>
<td>38</td>
<td>209</td>
<td>41</td>
<td>11383</td>
</tr>
<tr>
<td>arm64</td>
<td>6366</td>
<td>3511</td>
<td>36</td>
<td>195</td>
<td>27</td>
<td>10135</td>
</tr>
<tr>
<td>avr32</td>
<td>6360</td>
<td>3462</td>
<td>36</td>
<td>193</td>
<td>30</td>
<td>10081</td>
</tr>
<tr>
<td>blackfin</td>
<td>6380</td>
<td>3676</td>
<td>36</td>
<td>702</td>
<td>43</td>
<td>10837</td>
</tr>
<tr>
<td>c6x</td>
<td>6292</td>
<td>3293</td>
<td>35</td>
<td>189</td>
<td>28</td>
<td>9837</td>
</tr>
<tr>
<td>cris</td>
<td>6345</td>
<td>3436</td>
<td>45</td>
<td>254</td>
<td>78</td>
<td>10158</td>
</tr>
<tr>
<td>fvc</td>
<td>6316</td>
<td>3378</td>
<td>35</td>
<td>192</td>
<td>28</td>
<td>9949</td>
</tr>
<tr>
<td>hexagon</td>
<td>6292</td>
<td>3292</td>
<td>35</td>
<td>190</td>
<td>27</td>
<td>9836</td>
</tr>
<tr>
<td>ia64</td>
<td>6400</td>
<td>3519</td>
<td>36</td>
<td>195</td>
<td>27</td>
<td>10177</td>
</tr>
<tr>
<td>m32r</td>
<td>6324</td>
<td>3361</td>
<td>34</td>
<td>194</td>
<td>31</td>
<td>9944</td>
</tr>
<tr>
<td>m68k</td>
<td>6322</td>
<td>3453</td>
<td>35</td>
<td>192</td>
<td>37</td>
<td>10039</td>
</tr>
<tr>
<td>metag</td>
<td>6293</td>
<td>3363</td>
<td>37</td>
<td>193</td>
<td>28</td>
<td>9914</td>
</tr>
<tr>
<td>microblaze</td>
<td>6294</td>
<td>3335</td>
<td>37</td>
<td>195</td>
<td>32</td>
<td>9893</td>
</tr>
<tr>
<td>mips</td>
<td>6391</td>
<td>3964</td>
<td>36</td>
<td>198</td>
<td>28</td>
<td>10617</td>
</tr>
<tr>
<td>mn10300</td>
<td>6316</td>
<td>3426</td>
<td>35</td>
<td>199</td>
<td>33</td>
<td>10009</td>
</tr>
<tr>
<td>nios2</td>
<td>6292</td>
<td>3303</td>
<td>36</td>
<td>191</td>
<td>36</td>
<td>9858</td>
</tr>
<tr>
<td>openrisc</td>
<td>6292</td>
<td>3292</td>
<td>36</td>
<td>188</td>
<td>27</td>
<td>9835</td>
</tr>
<tr>
<td>parisc</td>
<td>6325</td>
<td>3384</td>
<td>34</td>
<td>191</td>
<td>27</td>
<td>9961</td>
</tr>
<tr>
<td>powerpc</td>
<td>6404</td>
<td>3933</td>
<td>37</td>
<td>205</td>
<td>40</td>
<td>10619</td>
</tr>
<tr>
<td>s390</td>
<td>6313</td>
<td>3453</td>
<td>35</td>
<td>194</td>
<td>27</td>
<td>10023</td>
</tr>
<tr>
<td>score</td>
<td>6292</td>
<td>3292</td>
<td>36</td>
<td>188</td>
<td>28</td>
<td>9835</td>
</tr>
<tr>
<td>sh</td>
<td>6374</td>
<td>3640</td>
<td>37</td>
<td>198</td>
<td>34</td>
<td>10283</td>
</tr>
<tr>
<td>sparcc</td>
<td>6370</td>
<td>3444</td>
<td>37</td>
<td>193</td>
<td>30</td>
<td>10074</td>
</tr>
<tr>
<td>tile</td>
<td>6306</td>
<td>3393</td>
<td>36</td>
<td>191</td>
<td>29</td>
<td>9955</td>
</tr>
<tr>
<td>um</td>
<td>6297</td>
<td>3362</td>
<td>38</td>
<td>193</td>
<td>27</td>
<td>9917</td>
</tr>
<tr>
<td>unicorn32</td>
<td>6363</td>
<td>3425</td>
<td>36</td>
<td>192</td>
<td>27</td>
<td>10042</td>
</tr>
<tr>
<td>x86</td>
<td>6420</td>
<td>3781</td>
<td>40</td>
<td>206</td>
<td>32</td>
<td>10479</td>
</tr>
<tr>
<td>xtensa</td>
<td>6326</td>
<td>3379</td>
<td>41</td>
<td>194</td>
<td>29</td>
<td>9969</td>
</tr>
</tbody>
</table>

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 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.

#### Table 6. Redundant or necessary symbols in Linux 4.0

<table>
<thead>
<tr>
<th>arch</th>
<th>redundant (⇔ n)</th>
<th>necessary (not ⇔ n)</th>
</tr>
</thead>
<tbody>
<tr>
<td>alpha</td>
<td>3223</td>
<td>55</td>
</tr>
<tr>
<td>arc</td>
<td>4263</td>
<td>63</td>
</tr>
<tr>
<td>arm</td>
<td>1691</td>
<td>75</td>
</tr>
<tr>
<td>arm64</td>
<td>3159</td>
<td>135</td>
</tr>
<tr>
<td>m88k</td>
<td>3741</td>
<td>32</td>
</tr>
<tr>
<td>metag</td>
<td>4301</td>
<td>67</td>
</tr>
<tr>
<td>microblaze</td>
<td>3251</td>
<td>71</td>
</tr>
<tr>
<td>maps</td>
<td>2773</td>
<td>64</td>
</tr>
<tr>
<td>mn10300</td>
<td>3702</td>
<td>39</td>
</tr>
<tr>
<td>nios2</td>
<td>4428</td>
<td>45</td>
</tr>
<tr>
<td>openrisc</td>
<td>4424</td>
<td>56</td>
</tr>
<tr>
<td>parisc</td>
<td>3499</td>
<td>51</td>
</tr>
<tr>
<td>powerpc</td>
<td>2652</td>
<td>94</td>
</tr>
<tr>
<td>s390</td>
<td>4149</td>
<td>107</td>
</tr>
<tr>
<td>score</td>
<td>7068</td>
<td>36</td>
</tr>
<tr>
<td>sh</td>
<td>3297</td>
<td>67</td>
</tr>
<tr>
<td>x86</td>
<td>2301</td>
<td>51</td>
</tr>
<tr>
<td>xtensa</td>
<td>3248</td>
<td>43</td>
</tr>
<tr>
<td>globally</td>
<td>135</td>
<td>1</td>
</tr>
</tbody>
</table>

---

\(^6\) Warthog Logic Framework: github.com/warthog-logic/warthog

\(^7\) PicoSAT: fmv.jku.at/picosat/
duration constraints have to change in order to make the configuration invalid. 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 3. Translation rules for tristate* operators

<table>
<thead>
<tr>
<th>$e'$</th>
<th>$\pi_0\left(e'\right)$</th>
<th>$\pi_1\left(e'\right)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$y$</td>
<td>$\top$</td>
<td>$\bot$</td>
</tr>
<tr>
<td>$m$</td>
<td>$\bot$</td>
<td>$\top$</td>
</tr>
<tr>
<td>$n$</td>
<td>$\bot$</td>
<td>$\top$</td>
</tr>
<tr>
<td>$A^T = T$</td>
<td>$p_0(A^T)$</td>
<td>$p_1(A^T)$</td>
</tr>
<tr>
<td>$e$</td>
<td>$\neg \pi_0\left(e\right) \land \neg \pi_1\left(e\right)$</td>
<td>$\pi_1\left(e\right)$</td>
</tr>
<tr>
<td>$e_0 &amp; \cdots &amp; e_n$</td>
<td>$\pi_0\left(e_0\right) \land \cdots \land \pi_0\left(e_n\right)$</td>
<td>$\pi_0\left(e_0\right) \lor \cdots \lor \pi_0\left(e_n\right)$</td>
</tr>
<tr>
<td>$e_0 \lor \cdots \lor e_n$</td>
<td>$\pi_0\left(e_0\right) \lor \cdots \lor \pi_0\left(e_n\right)$</td>
<td>$\neg \pi_0\left(e_0\right) \land \lor i \in {0,\ldots,n} \pi_1\left(e_i\right)$</td>
</tr>
<tr>
<td>$e_1 \iff e_2$</td>
<td>$\pi_0\left(e_1\right) \iff \pi_0\left(e_2\right) \land \pi_1\left(e_1\right) \iff \pi_1\left(e_2\right)$</td>
<td>$\bot$</td>
</tr>
<tr>
<td>$e_1 \implies e_2$</td>
<td>$\pi_0\left(e_2\right) \lor \neg \pi_0\left(e_1\right) \land \pi_1\left(e_1\right) \lor \pi_1\left(e_2\right)$</td>
<td>$\bot$</td>
</tr>
<tr>
<td>$A^T = t$</td>
<td>$\pi_0\left(A^T \iff t\right)$</td>
<td>$\bot$</td>
</tr>
<tr>
<td>$A^T = B^T$</td>
<td>$\pi_0\left(A^T \iff B^T\right)$</td>
<td>$\bot$</td>
</tr>
<tr>
<td>$X^S = S$</td>
<td>$P_{X^S \leftarrow t}$</td>
<td>$\bot$</td>
</tr>
<tr>
<td>$X^S = Y^S$</td>
<td>$P_{X^S \leftarrow Y^S}$</td>
<td>$\bot$</td>
</tr>
<tr>
<td>$e_1^S \neq e_2^S$</td>
<td>$\neg \pi_0\left(e_1^S = e_2^S\right)$</td>
<td>$\bot$</td>
</tr>
</tbody>
</table>

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.
Table 5. Sizes of POFs for Linux 4.0

<table>
<thead>
<tr>
<th>arch</th>
<th>regular variables in POF in Tristate* Logic</th>
<th>auxiliary variables in POF in Tristate* Logic</th>
<th>total number of variables in POF in Tristate* Logic</th>
<th>variables in L-POF</th>
<th>variables in CNF</th>
<th>clauses in CNF</th>
</tr>
</thead>
<tbody>
<tr>
<td>alpha</td>
<td>10673</td>
<td>48845</td>
<td>59518</td>
<td>117417</td>
<td>996839</td>
<td>1812856</td>
</tr>
<tr>
<td>arc</td>
<td>10602</td>
<td>48481</td>
<td>59083</td>
<td>116538</td>
<td>954428</td>
<td>1718683</td>
</tr>
<tr>
<td>arm</td>
<td>11976</td>
<td>55760</td>
<td>67736</td>
<td>134270</td>
<td>1299812</td>
<td>2849653</td>
</tr>
<tr>
<td>arm64</td>
<td>10824</td>
<td>49640</td>
<td>60464</td>
<td>119333</td>
<td>1007563</td>
<td>1820311</td>
</tr>
<tr>
<td>avr32</td>
<td>10793</td>
<td>49366</td>
<td>60159</td>
<td>118671</td>
<td>1001036</td>
<td>1816464</td>
</tr>
<tr>
<td>blackfin</td>
<td>11539</td>
<td>51058</td>
<td>62597</td>
<td>123096</td>
<td>1026513</td>
<td>1861942</td>
</tr>
<tr>
<td>c6x</td>
<td>10548</td>
<td>48174</td>
<td>58722</td>
<td>119333</td>
<td>949031</td>
<td>1708805</td>
</tr>
<tr>
<td>cris</td>
<td>10867</td>
<td>49279</td>
<td>60146</td>
<td>118559</td>
<td>99006</td>
<td>1812066</td>
</tr>
<tr>
<td>frv</td>
<td>10666</td>
<td>48722</td>
<td>59388</td>
<td>117126</td>
<td>990489</td>
<td>1797565</td>
</tr>
<tr>
<td>hexagon</td>
<td>10540</td>
<td>48169</td>
<td>58709</td>
<td>115795</td>
<td>953582</td>
<td>1720572</td>
</tr>
<tr>
<td>ia64</td>
<td>10866</td>
<td>49850</td>
<td>60716</td>
<td>119837</td>
<td>1010856</td>
<td>1834072</td>
</tr>
<tr>
<td>m32r</td>
<td>10643</td>
<td>48681</td>
<td>59324</td>
<td>116671</td>
<td>955382</td>
<td>1780405</td>
</tr>
<tr>
<td>m68k</td>
<td>10717</td>
<td>49136</td>
<td>59853</td>
<td>118115</td>
<td>1008800</td>
<td>1836987</td>
</tr>
<tr>
<td>microblaze</td>
<td>10591</td>
<td>48406</td>
<td>58997</td>
<td>116370</td>
<td>985108</td>
<td>1778040</td>
</tr>
<tr>
<td>mips</td>
<td>11249</td>
<td>52034</td>
<td>63283</td>
<td>125090</td>
<td>1048937</td>
<td>1909971</td>
</tr>
<tr>
<td>mm10300</td>
<td>10721</td>
<td>48974</td>
<td>59695</td>
<td>117738</td>
<td>994158</td>
<td>1804015</td>
</tr>
<tr>
<td>nios2</td>
<td>10566</td>
<td>48235</td>
<td>58801</td>
<td>115951</td>
<td>950035</td>
<td>1710610</td>
</tr>
<tr>
<td>openrisc</td>
<td>10544</td>
<td>48168</td>
<td>58712</td>
<td>115783</td>
<td>949121</td>
<td>1709044</td>
</tr>
<tr>
<td>parisc</td>
<td>10658</td>
<td>48794</td>
<td>59452</td>
<td>117297</td>
<td>996712</td>
<td>1811011</td>
</tr>
<tr>
<td>powerpc</td>
<td>11247</td>
<td>51964</td>
<td>63211</td>
<td>124935</td>
<td>1055822</td>
<td>1917736</td>
</tr>
<tr>
<td>s390</td>
<td>10699</td>
<td>49084</td>
<td>59783</td>
<td>117997</td>
<td>998901</td>
<td>1831201</td>
</tr>
<tr>
<td>score</td>
<td>10539</td>
<td>48168</td>
<td>58707</td>
<td>115783</td>
<td>949788</td>
<td>1710461</td>
</tr>
<tr>
<td>sh</td>
<td>10955</td>
<td>50336</td>
<td>61291</td>
<td>121037</td>
<td>1020515</td>
<td>1854779</td>
</tr>
<tr>
<td>sparc</td>
<td>10774</td>
<td>49327</td>
<td>60101</td>
<td>118582</td>
<td>1004762</td>
<td>1823946</td>
</tr>
<tr>
<td>tile</td>
<td>10655</td>
<td>48748</td>
<td>59403</td>
<td>117195</td>
<td>990749</td>
<td>1798113</td>
</tr>
<tr>
<td>um</td>
<td>10606</td>
<td>48550</td>
<td>59156</td>
<td>116723</td>
<td>998908</td>
<td>1821791</td>
</tr>
<tr>
<td>unicore32</td>
<td>10753</td>
<td>49191</td>
<td>59944</td>
<td>118246</td>
<td>998333</td>
<td>1811566</td>
</tr>
<tr>
<td>x86</td>
<td>11135</td>
<td>51280</td>
<td>62415</td>
<td>123314</td>
<td>1051478</td>
<td>1913811</td>
</tr>
<tr>
<td>xtensa</td>
<td>10674</td>
<td>48786</td>
<td>59460</td>
<td>117278</td>
<td>993296</td>
<td>1802906</td>
</tr>
</tbody>
</table>

References


