A Kconfig Translation to Logic with One-Way Validation System

David Fernandez-Amoros
UNED
Madrid, Spain
david@issi.uned.es

Ruben Heradio
UNED
Madrid, Spain
rheradio@issi.uned.es

Christoph Mayr-Dorn
JKU Institute for Software Systems Engineering, Johannes Kepler University
Linz, Austria
christoph.mayr-dorn@jku.at

Alexander Egyed
JKU Institute for Software Systems Engineering, Johannes Kepler University
Linz, Austria
alexander.egyed@jku.at

ABSTRACT

Automated analysis of variability models is crucial for managing software system variants, customized for different market segments or contexts of use. As most approaches for automated analysis are built upon logic engines, they require having a Boolean logic translation of the variability models. However, the translation of some significant languages to Boolean logic is remarkably non-trivial. The contribution of this paper is twofold: first, a translation of the Kconfig language is presented; second, an approach to test the translation for any given model is provided. The proposed translation has been empirically tested with the introduced validation procedure on five open-source projects.

ACM Reference Format:
David Fernandez-Amoros, Ruben Heradio, Christoph Mayr-Dorn, and Alexander Egyed. 2019. A Kconfig Translation to Logic with One-Way Validation System. In *23rd International Systems and Software Product Line Conference - Volume A (SPLC '19), September 9–13, 2019, Paris, France.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3336294.3336313

1 INTRODUCTION

The creation of system variants is essential in software engineering paradigms: software product lines, software ecosystems, context-aware software, etc. In such paradigms, *Variability Models (VMs)* are frequently used to account for the configurable features of the variants.

Automated analysis of VMs has a long history[4, 18, 19]. For instance, to detect whether a model instance (also known as *product* or *configuration*) is *valid* [1–3, 10, 16, 20, 23, 24, 30, 36, 38, 40] (i.e. it does not violate any inter-feature constraints); to provide explanations about the causes that make an instance invalid in order to guide the user to solve the problem [1, 3, 8, 12, 13, 20, 30, 33, 34, 38, 42]; to detect *dead features* in the model (i.e. features that cannot be part of any valid instance) [8, 12, 13, 20, 25, 26, 33, 34, 39, 41, 42, 44, 45], etc.

A common approach for VM analysis is translating the model into a Boolean formula, which is then processed with a logic engine. For some VM languages, such as feature models, the translation to logic is straightforward [3]. However, the translation of the Kconfig language has not been adequately described. Several attempts have been made to translate Kconfig code to Boolean logic[5, 6, 14, 21, 27–29, 37, 43].

A Kconfig translation is challenging for two major reasons: first, a formal specification of the KConfig language does not exist, only the various versions of the `kconfig-language.txt`[1] file. Second, the translation requires a paradigm shift from imperative to logic: Using the *conf* application, an application engineer can configure the software project. The feature values are variables that can change from one value to another for as long as the configuration process lasts (e.g., a feature with a false value may change to true later when a select clause for another feature is evaluated). This is in contrast to propositional variables which can only have one value in a particular configuration. This paper provides the following two contributions to research on VM analysis:

(1) A translation from Kconfig to logic which is complete for Boolean configs. The translation supports language features that have been omitted in some of the related work, such as: *chaining*, *visibility conditions*, *user prompts*, *default values*, etc. Also, no artificial variables are added in the translation.
(2) Empirical validation of the translation on five real-world projects.

2 INTRODUCING KCONFIG

This section provides a walkthrough of the main features of the Kconfig language with an emphasis on their role in the translation process to Boolean logic.

Configs and symbols. Figure 1 provides an example of the configuration file for the embedded OS of a hypothetical multimedia device. Kconfig deals with *configs*. A config is a declaration of a *symbol* (e.g., HAVE_WIFI on line 1) with a *type* (e.g., bool on line 2) and other attributes. The type restricts the possible values available for a symbol. A symbol can be declared multiple times (usually with different attributes) in different configs, as long as the type is the

This work has been supported by (i) the Spanish Ministry of Education and Vocational Training under the projects with reference DPI2016-77677-P and CAS17/00022, (ii) the Austrian Science Fund (FWF): P29415-NBL funded by the Government of Upper Austria; and (iii) Pro2Future: FFG, Contract No. 854184.

SPLC '19, September 9–13, 2019, Paris, France
© 2019 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-7138-4/19/09…$15.00
https://doi.org/10.1145/3336294.3336313

[1]https://www.kernel.org/doc/Documentation/kbuild/kconfig-language.txt

same. In Figure 1, for example, USE_CUDA is declared three times (lines 60, 66 and 71).

```
1   config HAVE_WIFI
2    bool
3    default n
4
5   config STREAMING
6    bool "Streaming?"
7    select WIFI
8
9   config USB
10   bool "Use USB?"
11
12  config WIFI
13   bool "Use Wifi?"
14    if HAVE_WIFI
15   default n
16
17  config WPA
18   bool "Use WPA?"
19   depends on WIFI
20   default y
21
22  config WEP
23   bool
24   default n if WPA2
25   default y
26
27  menu "Compiler"
28   visible if EXPERT
29   depends on DEVEL
30    config GCC
31     bool "Use GCC?"
32
33    config CLANG
34     bool
35     default !GCC
36  endmenu
37
38
39  config PROFILE
40   bool "Profile?"
41
42  config TWEAK
43    bool "Tweak?"
44    default y
45    select PROFILE
46      if KEYBOARD
47
48  choice
49   depends on PROFILE
50   prompt "USER "
51   config STANDARD
52    bool "Standard"
53   config ADVANCED
54    bool "Advanced"
55   config EXPERT
56    bool "Expert"
57  endchoice
58
59  if EXPERT
60   config USE_CUDA
61    bool "USE CUDA?"
62  end−if
63
64  if ADVANCED &&
65     HAVE_CUDA
66   config USE_CUDA
67    bool "USE CUDA?"
68    default n
69  end−if
70
71  config USE_CUDA
72   bool
73   default n
74
75  if HAVE_WIFI
76   source "Config.in"
77  end−if
```

Figure 1: Example Kconfig file

Symbol types. There are four types of configs in the projects evaluated: *bool*, *string*, *int*, and *hex*. Bool configs may have *n* and *y* values to represent logic *false* and *true*. A common use of string configs is to hold the string value of a set of related Boolean configs. The types int and hex are essentially equal to string. For space limitations, this paper will consider only bool configs. In the projects evaluated, no string-like types were used inside the bool configs, which means that the translation is complete for these configs.

User input. Configs may specify a *prompt* to obtain a value from the user. On line 6, the text *Streaming?* is displayed and a value for STREAMING is requested from the user. The prompt may be guarded by a Boolean expression (e.g. lines 13-14). When the expression evaluates to false, the text is not shown and the user input is not requested. If HAVE_WIFI is false or unassigned, the default value of "n" will be assigned to WIFI.

Default values may be specified for a config. In combination with a prompt, the default value is shown as a suggestion to the user. Without a usable prompt (e.g., lines 22-25) the config takes the *default value* if one exists. If there is no default value, the symbol remains unassigned. A later redeclaration of the same symbol may provide a value for it, in the meantime, logical expressions containing an unassigned value evaluate to false. A default clause consists of a value and an optional Boolean expression (e.g., line 24). If there is more than one default clause, they are evaluated in order. Symbol WEP in line 25 has no prompt, so default clauses are employed. If WPA2 is false, the WEP is assigned true. Otherwise, it is assigned true.

Direct dependencies, menus, and *if* blocks. A config declaration may have *direct dependencies*; a dependency is a Boolean expression which must be satisfied in order for the config to be evaluated. If the dependency of a config does not hold, the rest of the declaration is ignored. For example, the config WPA is only evaluated when WIFI is true. A *menu* is a mechanism for grouping related configs (lines 27-36). It allows imposing dependencies to a group of configs at once. A config inside a menu *inherits* the dependencies of the menu. Menus may be nested and thus inherit the dependencies from their respective parents.

Visibility conditions further constraint menu evaluation. The visibility condition acts as menu-wide guard to prompts; instead of adding the guard to each individual prompt, it is specified at the menu level. In the "Compiler" menu, both GCC and CLANG depend on DEVEL. Also, the prompt for GCC is not shown unless EXPERT is true. Another way to add dependencies to a group of configs is through use of an *if-block*. It takes a condition and adds it as an additional dependency to the declarations (configs, menus, choices and other if-blocks) inside the block. In line 59, USE_CUDA is inside an if-block with EXPERT as condition, which is equivalent to having a "depends on EXPERT" clause.

Reverse dependencies. A Boolean config may declare *reverse dependencies* through the use of the *select* construction (e.g. line 7). If the config dependencies are met, the symbol (the selector, here STREAMING) logically implies another one (the selected, here WIFI, so that, if the selector is true, the selected has to be true as well. When the select clause is guarded by an expression (e.g. lines 45-46), the selector and the condition imply the selected. In our example, when TWEAK and KEYBOARD are true, PROFILE is assigned "y", regardless of the previous value.

Choices. This construct enables grouping a set of dependencies and config declarations out of which only exactly one config may become true while the rest become false (lines 48-57). If the dependencies do not hold (line 49), then all the choice members are set to false. The choice construction contains config declarations and may also include if-blocks. In the example, if PROFILE is true, the user is asked to choose a value among STANDARD, ADVANCED or EXPERT.

Importing Kconfig files. Finally, Kconfig enables including another Kconfig file via the *source* construction (e.g. line 76). If it is inside a menu or if-block, the source will inherit the corresponding dependencies. A source command with unmet dependencies is ignored during the configuration process, i.e. the source file is not included.

3 CLARIFYING KCONFIG SEMANTICS

The Kconfig interpreter requires configuration files with correct syntax, although developers often produce incorrect Kconfig code because of false assumptions [17].

Kconfig semantics, however, is primarily defined by what the interpreter accepts as correct. The `kconfig-language.txt` file is insufficient for translating Kconfig to Boolean logic. In the following subsection we describe aspects that have been incorrectly or incompletely described before, that have convoluted semantics, or that tend to lead to incorrect configurations.

3.1 Config chaining

Kconfig files typically declare some symbol several times. A new declaration of the same symbol (e.g., lines 59-73 in Fig. 1) does not *override* the previous one. The interpreter evaluates configurations of the same symbol in order. Every symbol starts outs unassigned. Configs of unassigned symbols are evaluated. A config will provide no value for the symbol if either a) its dependencies are not met or b) there is no productive prompt and no productive default (i.e. there are none or they are guarded by conditions that evaluate to false). Instead, the interpreter checks successive declarations of the same symbol as long as it has no value assigned. Once a symbol obtains an assignment (i.e. true or false) all remaining redeclarations are ignored. We coined the term *chaining* to account for this very usual behavior.

3.2 Decision overriding with Select

The dependencies of a selected symbol are not evaluated. It is thus possible to set to true a symbol whose dependencies do not hold. The use of the *select* construction is discouraged because it has the potential to produce wrong configurations, bypassing the dependency mechanism. This usage is referred to as *heavy-handed select*. Lines 39-46 in Figure 1 provide an example.

```
1  config H
2   bool "Prompt"
3   select A if !C
4  config J
5   bool
6   default y
7   select D if X
8  config J
9     bool "Prompt"
10    depends on X
11    select H
12 config ZA
13  bool "ZA?"
14  select ZB
15 config ZB
16  bool "ZB?"
17  depends on X
18  select ZC
19 config ZC
20  bool
```

Figure 2: Convoluted select clauses

3.3 Select in chained configs

The interpreter evaluates select statements even in configurations that are not evaluated. The configuration of symbol J in lines 4 to 7 in Figure 2 leads to an assignment of true. While the config of J in lines 8-11 is not evaluated because J is already assigned, the interpreter will nevertheless select H (line 1) if X is true (line 7). The reason is that the interpreter stores select clauses as reverse dependencies of the selected symbols.

3.4 Select transitiveness

The documentation is unclear w.r.t. transitivity of select clauses: if ZA selects symbol ZB and ZB selects symbol ZC in Figure 2, does this imply that if ZA is true, then ZC is also true? The answer depends on whether ZB's dependencies are met. The interpreter stores the selector, the dependencies of the config and the optional guard as attributes of the selected. This way, a select clause is only triggered if the dependencies of the selector are true.

4 RELATED WORK

One of the first studies of variability models extracted from the Linux kernel was carried out by She et al. [27–29]. The formalism employed uses denotational semantics. There is no translation to logic and no validation attempt is made. She et al. describe the formalism as incomplete since some "corner cases" are not modeled. The only hint as to what are considered corner cases is heavy-handed select. Config declarations are not checked for redeclarations, so chaining of configs is not even considered. This formalism was apparently the source of the translation used by Berger et al. in [5, 6] where a catalog of software system projects was assembled as defined in Kconfig and eCos files, dubbed "Variability Models in the Wild". Again no empirical validation of the translation seems to have been performed. There are no details of the translation, although configs are treated separately, so chaining does not seem to be implemented. The models make heavy use of artificial variables, such as those used in the Tseitin construction [35] to transform a formula to CNF.

Tartler et al. [31, 32] adapt the Linux Variability Analysis Tool (LVAT)[2] from She to extract variability models from both Kconfig and code artifacts with the purpose of detecting and repairing inconsistencies between them.

Zengler and Küchlin [43] go a step further and present an encoding into Boolean logic that they illustrate only for the Linux kernel. Their model is updated in [37]. The translation is incomplete; prompts, visibility conditions, and default values are not considered. No experimental validation of the translation is presented. The translation is used to find dead and core features; however chaining is not considered.

In [14], another translation is presented. It is only for Boolean configs and lacks both chaining and validation. The translation is used to build BDDs representing the model.

A thorough experimental revision of the semantics of the Kconfig language is presented in [11]. Nowhere is the issue of chaining mentioned. Neither is the need to add the dependencies of a selector to translate a select clause.

KconfigReader is a tool that translates Kconfig files to Boolean logic. To date, it is the translation covering the most of the Kconfig language. In [21], the validation approach of KconfigReader is described. The translation itself is not explained. The tools can also perform brute-force validation of very small snippets of code using the official conf interpreter capabilities. It provides a useful repository of test cases, some randomly generated and some taken from previous research. Two limitations are mentioned, though: Limitations regarding reverse dependencies and the inability to decide the validity of configurations using string values not mentioned in the kconfig code. The command-line interpreter conf allows checking configurations through command-line options. The process can be summarised as follows: If present, a file containing symbol values from a previous configuration is read, then the configuration process begins. User input is never requested, the values read from the file are used instead. These values may still change (e.g., due to select clauses). When the configuration is finished, the symbol values are written back to the file. Kästner uses this mechanism as an oracle: Every possible combination for the values is generated and written to a file. The interpreter is then invoked on the file. At the end, the file is checked for changes. If the file has not changed, the values are correct, otherwise the values are not valid. KconfigReader includes a repository of test cases, unfortunately, chaining is not even tested. The tool is complete and correct according to our evaluation, although it introduces artificial variables that obscure the meaning.

5 TRANSLATION TO BOOLEAN LOGIC

We extended the original parser to extract a set of additional attributes for each declaration: To propagate dependencies and visibility conditions, the parser keeps a stack of open commands (if block,

[2] https://github.com/matachi/linux-variability-analysis-tools.exconfig

menu, choice) so when a config is declared, the top of the stack is checked and the corresponding attributes inherited. The translation presented here relies heavily on the "if-then-else" construction. Of course, "if A then B else C" can still be considered as a shorthand for the more Boolean logic-looking $(\neg A \vee B) \wedge (A \vee C)$.

5.1 Config translation

For the translation of bool configs, one Boolean variable with the same name is used. To distinguish different configs of the same symbol a subindex may be used. During the parsing of the Kconfig files, a series of attributes for each declaration are compiled. For a config FOO_i a series of syntactic expressions from the parsing phase is derived: FOO_i.dep, FOO_i.vis, X_i.prompt, FOO_i.promptGuard, standing for the dependencies of config FOO_i, the visibility condition of FOO_i, a Boolean value indicating if there is a prompt in the declaration or not, and the expression guarding the prompt (true if there is none), respectively. The system also keeps track of which config selects which symbol for later use. The translation is composed of the following steps: The first piece to put together for each config is a logic formula for the default value. Consider this config:

config FOO_i
default V_{i1} if C_{i1}
default V_{i2} if C_{i2}
...
default V_{in_i} if C_{in_i}

and let us define:

FOO_i.default ≡
if C_{i1} then V_{i1} else
if C_{i2} then V_{i2} else
...
if C_{in_i} then $\text{V}in_i$

The second aspect to consider is whether the config is selected by another config or not. As mentioned in the previous section, a select clause requires that the selector dependencies are met to activate the selected symbol. So first, a list of selectors is needed, together with their respective dependencies and optional guarding expressions. For k ∈ $\{1..m_{\text{FOO}}\}$, let that be:
config SELECTOR_k
bool ...
select FOO if GUARD_k

Let us define a formula for later use:

FOO.selectCondition ≡
$\bigwedge_{k=1}^{m_{\text{FOO}}} \text{SELECTOR}_k.\text{dep} \wedge \text{SELECTOR}_k \wedge \text{GUARD}_k$

Let us now consider the user input. If it is going to be requested, any value is possible for the symbol and nothing else needs to be done. In contrast, if user input is not requested, then each configFOO_i is checked for usable default values until a value is computed for the symbol or there are no more configs. For a configFOO_i to request user input, the dependencies must hold, and there has to be a visible prompt meeting its guard. If just one declaration of FOO meets this criterion, user input is asked. So, let us define:
FOO.promptCondition ≡
$\bigvee_{i=1}^{n} (\text{FOO}_i.\text{dep} \wedge \text{FOO}_i.\text{prompt} \wedge \text{FOO}_i.\text{promptGuard})$
To put it all together, the translation for symbol FOO is:
Translation(FOO) ≡ if FOO.selectCondition then FOO
else if FOO.promptCondition then true
else if $\text{FOO}_1.\text{dep} \wedge \bigvee_{i=1}^{n_1} \text{C}_{1i}$ then FOO_1.default
else if $\text{FOO}_2.\text{dep} \wedge \bigvee_{i=1}^{n_2} \text{C}_{2i}$ then FOO_2.default
...
else if $\text{FOO}_r.\text{dep} \wedge \bigvee_{i=1}^{n_r} \text{C}_{ri}$ then FOO_r.default
else ¬FOO

5.2 Choice translation

Let us consider a choice involving n symbols, $\text{X}_1, \text{X}_2, \ldots, \text{X}_n$. The symbols inside the choice block, also called choice members, are assumed not to be declared anywhere else[3] . The semantics of the Boolean choice is fairly straightforward: if the dependencies hold and the choice is visible, exactly one of the choice members is true. Otherwise, all the choice members must be false. A member of a choice must have a usable prompt to be considered. A choice with no member translates to true. Default values for the choice and selects to choice members are ignored. For each member, X_i, let us define the auxiliary formula $\text{B}_i \equiv \text{X}_i \wedge \text{X}_i.\text{prompt} \wedge \text{X}_i.\text{promptGuard}$. The translation for a choice C, would then be:

translation(C) ≡ if $\text{C.dep} \wedge \text{C.vis} \wedge \text{C.prompt} \wedge \text{C.promptGuard}$ $\wedge \bigvee_{i=1}^{n} (\text{X}_i.\text{prompt} \wedge \text{X}_i.\text{promptGuard})$ then
$\bigvee_{i=1}^{n} \text{B}_i \wedge \bigwedge_{\substack{i=1 \\ j>i}}^{n} \neg(\text{B}_i \wedge \text{B}_j)$ else $\bigwedge_{i=1}^{n} \neg \text{B}_i$. The result of the translation process is a set of formulas whose conjunction provides a logical model for the corresponding Kconfig files.

6 ONE-WAY VALIDATION OF THE PROPOSED TRANSLATION

Figure 3 summarizes the one-way validation approach. For a Kconfig translation to Boolean logic, S, a random sample of valid products $p_1, p_2, \ldots, p_n$ is generated using the *conf* program. Then, n BDDs are built, one for each conjunction $S \wedge p_1, S \wedge p_2, \ldots, S \wedge p_n$. Each BDD is then checked to see if there is only one solution, namely that of the corresponding product. Otherwise, the logic translation is invalid. BDDs are described in detail in [9, 22]. There are two

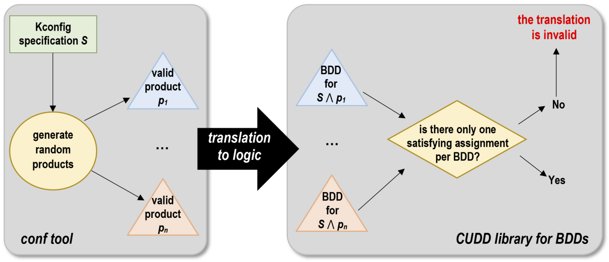

Figure 3: Schema of the one-way validation approach

good reasons to use BDD technology for our validation approach:

[3]The interpreter would otherwise issue a warning.

1) **BDDs do not require CNF-formulas**. A BDD can be built by feeding it formulas which do not need to be in *Conjunctive Normal Form (CNF)*. This is unlike SAT-solvers [7], which traditionally require them. Translation to CNF can produce a formula exponentially longer than the original one (e.g. the translation for freetz in CNF would requiere billions of clauses). Alternative approaches to get a CNF formula, such as the Tseitin's construction of an equisatisfiable formula [35], require the introduction of a great deal of artificial variables which impede efficiency and obscure the meaning.

2) **BDD-based validation approach helps debugging the translation**. Getting the translation of a complex VM language to logic right usually requires several iterations. When a generated product does not validate, there are two possibilities: a) There are zero solutions. b) There is more than one solution. If there are no solutions for the BDD, there is a contradiction. In that case, the building process can be retraced to the point in which the BDD became *false*, because the number of nonterminal nodes in the BDD drops to zero. The offending formula points to the symbol(s) or choice that is causing the problem. If there is more than one satisfying assignment, BDDs are more useful than SAT-solvers: In a valid product, there is only one correct value for each variable. For that reason, there is also only one node per variable in the BDD and for every node, one of the children has to point to false. There also no empty levels. It suffices to traverse the BDD, find out which nodes violate that property, and associate them with the variables. This way, a set of symbols whose translation is wrong is obtained, which can be used to troubleshoot the translation.

The biggest disadvantage of BDDs (the potential for memory exhaustion) is mitigated because instead of facing the problem head-on and making a BDD of the full logical model, an alternative route is taken to always build the BDD corresponding to the model and a product to be tested. Such a BDD is known to have only one node per variable, assuming the model is correct.

To validate the translation described in Section 5, the open-source projects in Table 1 have been used as test-bed. These projects are heterogeneous in the number of configs and concerning how they use the Kconfig language, thus being a representative sample to account for the diversity of the population of Kconfig projects. The table also shows that the vast majority of configs were of type bool. A thousand products for each project were generated and all of them validated for the translation presented in the paper (i.e. the conjunction of the model translation plus each individual product has exactly one satisfying assignment). The debugging process allowed us to discover the chaining mechanism and also the need to include a selector's dependencies in the translation. The code and instructions to replicate the results have been made available in a repository[4]. We also evaluated KconfigReader [21] under the same conditions (i.e. only bool configs). KconfigReader translates all the symbol types as opposed to only bool, but failed to build models for busybox and uClibc due to syntax errors. For the other projects the translation process completed successfully and the models validated without any problems, although the translation involved adding additional variables not corresponding to config definitions, which goes in the way of later product line analysis. Interestingly, the short snippet validation approach in [21] carried over to the big

Name	#Feat.	% bool	Kconfig2Logic**		KconfigReader	
			Time*	Valid	Time*	Valid
toybox	12	83.33	0.04	100%	0.03	100%
axtls	64	67.36	0.05	100%	0.05	100%
uclibc	303	87.82	1.26	100%	error	error
busybox	604	94.52	2.40	100%	error	error
freetz	6492	98.43	69.3	100%	323.8	100%

*Avg time taken to generate and test one configuration (seconds).
**Our approach

Table 1: Project validation for 1000 sample products

projects. The running times comparison shows that our translation scales better that KconfigReader. The reason seems to be that KconfigReader produces a very long translation, while our approch only produces at most one constraint per config. For instance, for the freetz project, KconfigReader produced 81780 contraints vs. 5373 for our approach.

To build the BDDs, the CUDD package by Fabio Somenzi was used[5]. The experiments were performed on an Intel Xeon@2Ghz running Linux. The memory requirements are very small, less than 250MB per run with the biggest model.

7 THREATS TO VALIDITY AND LIMITATIONS

The primary threat to validity of our translation is the modification of the *conf* program to produce random valid products. This modification improves the reliability of our validation procedure: while the original *conf* program has an option to create random configurations, it uses different code for generating configurations than for configuring them. This could give rise to incompatible results. To mitigate this problem, *conf* was modified with care not to alter its intended behavior. The interpreter is designed to show the user a list of acceptable values for boolean and tristate types. We reprogrammed the function waiting for user input to return a randomly chosen value among the alternatives to ensure that generating and configuring work the same way. A limitation of the validation is that it goes is one-way only. We make no assurances that the translation is correct the other way around.

8 CONCLUSIONS

The translation of a VM to logic is a necessity for most automated analysis approaches because they are built upon logic engines. This paper has described how to translate Kconfig to logic for bool configs, including some of its most obscure constructions, such as config chaining and convoluted selects. We have also checked if valid products according to a Kconfig specification remain valid according to its logical translation for five different projects. Both our translation and KconfigReader passed the tests. KconfigReader is able to translate string types but our approach is more compact, more scalable, more compatible with Kconfig varieties, and overall clearer, since there is a one-to-one correspondence between configs and logical variables. Future work will explain the translation of string and tristate types.

[4]https://figshare.com/s/df2b0e4bc889a701f3f3

[5]http://vlsi.colorado.edu/~fabio/

REFERENCES

[1] Hai H. Wang A, Yuan Fang Li B, Jing Sun C, Hongyu Zhang D, and Jeff Pan E. 2007. Verifying feature models using OWL. *Journal of Web Semantics* 5 (2007), 117–129.

[2] Randall C. Bachmeyer and Harry S. Delugach. 2007. A Conceptual Graph Approach to Feature Modeling. In *Conceptual Structures: Knowledge Architectures for Smart Applications, 15th International Conference on Conceptual Structures, (ICCS)*. Springer, 179–191. https://doi.org/10.1007/978-3-540-73681-3_14

[3] Don Batory. 2005. Feature models, grammars, and propositional formulas. In *9th international conference on Software Product Lines*. Springer-Verlag, Rennes, France, 7–20.

[4] David Benavides, Sergio Segura, and Antonio Ruiz-Cortés. 2010. Automated Analysis of Feature Models 20 years Later: a Literature Review. *Information Systems* 35, 6 (2010), 615–636.

[5] Thorsten Berger, Steven She, Rafael Lotufo, Andrzej Wąsowski, and Krzysztof Czarnecki. 2010. Variability modeling in the real: a perspective from the operating systems domain. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*. ACM, IEEE, Lawrence, KS, USA, 73–82.

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

[7] Armin Biere, Marijn J.H. Heule, Hans van Maaren, Toby, and Walsh. 2009. *Handbook of Satisfiability*. Frontiers in Artificial Intelligence and Applications, Vol. 185. IOS Press, Amsterdam, The Netherlands, The Netherlands. 697–698 pages.

[8] Pim Van Den Broek and Ismênia Galvão. 2009. Analysis of Feature Models using Generalised Feature Trees. In *Third International Workshop on Variability Modelling of Software-Intensive Systems, Seville, Spain, January 28-30, 2009. Proceedings (ICB Research Report)*, David Benavides, Andreas Metzger, and Ulrich W. Eisenecker (Eds.), Vol. 29. Universität Duisburg-Essen, 29–35. http://www.vamos-workshop.net/proceedings/VaMoS_2009_Proceedings.pdf

[9] Randal E. Bryant. 1986. Graph-based algorithms for Boolean function manipulation. *IEEE Trans. Comput.* 8, C-35 (1986), 677–691.

[10] Krzysztof Czarnecki and Peter Chang. 2005. Cardinality-based feature modeling and constraints: A progress report. In *International Workshop on Software Factories*. ACM, 16–20.

[11] Sascha El-Sharkawy, Adam Krafczyk, and Klaus Schmid. 2015. Analysing the Kconfig Semantics and Its Analysis Tools. In *Proceedings of the ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE 2015)*. ACM, New York, NY, USA, 45–54. https://doi.org/10.1145/2814204.2814222

[12] Abdelrahman Osman Elfaki, Somnuk Phon-Amnuaisuk, and Chin Kuan Ho. 2008. Knowledge Based Method to Validate Feature Models. In *12th Software Product Line Conference (SPLC)*, Vol. 2. IEEE Computer Society, Los Alamitos, CA, USA, 217–225.

[13] Abdelrahman Osman Elfaki, Somnuk Phon-Amnuaisuk, and Chin Kuan Ho. 2009. Using First Order Logic to Validate Feature Model. In *Third International Workshop on Variability Modelling of Software-Intensive Systems, Seville, Spain, January 28-30, 2009. Proceedings (ICB Research Report)*, David Benavides, Andreas Metzger, and Ulrich W. Eisenecker (Eds.), Vol. 29. Universität Duisburg-Essen, 169–172. http://www.vamos-workshop.net/proceedings/VaMoS_2009_Proceedings.pdf

[14] David Fernandez-Amoros, Ruben Heradio, Carlos Cerrada, Enrique Herrera-Viedma, and J Cobo Manuel. 2017. Towards Taming Variability Models in the Wild. In *New Trends in Intelligent Software Methodologies, Tools and Techniques: Proceedings of the 16th International Conference SoMeT_17*, Vol. 297. IOS Press, Amsterdam, The Netherlands, 454.

[15] David Fernandez-Amoros, Ruben Heradio, and Jose Antonio Cerrada. 2009. Inferring Information from Feature Diagrams to Product Line Economic Models. In *Proceedings of the 13th International Conference on Software Product Lines*. Carnegie Mellon University, Pittsburgh, PA, USA, 41–50.

[16] Rohit Gheyi, Tiago Massoni, and Paulo Borba. 2006. A theory for feature models in alloy. In *Proceedings of the ACM SIGSOFY First Alloy Workshop*. ACM, 71–80.

[17] Stefan Hengelein and Daniel Lohmann. 2015. *Analyzing the Internal Consistency of the Linux KConfig Model*. Master's thesis. University of Erlangen, Dept. of Computer Science, 2015.

[18] Ruben Heradio, David Fernandez-Amoros, Jose A. Cerrada, and Ismael Abad. 2013. A Literature Review on Feature Diagram Product Counting and Its Usage in Software Product Line Economic Models. *International Journal of Software Engineering and Knowledge Engineering* 23, 08 (2013), 1177–1204.

[19] Ruben Heradio, Hector Perez-Morago, David Fernandez-Amoros, Francisco Javier Cabrerizo, and Enrique Herrera-Viedma. 2016. A bibliometric analysis of 20 years of research on software product lines. *Information and Software Technology* 72 (2016), 1 – 15.

[20] Kyo Kang, Sholom Cohen, James Hess, William Novak, and Spencer Peterson. 1990. *Feature-Oriented Domain Analysis (FODA) Feasibility Study*. Technical Report CMU/SEI-90-TR-21. Software Engineering Institute.

[21] Christian Kästner. 2017. Differential Testing for Variational Analyses: Experience from Developing KConfigReader. *CoRR* abs/1706.09357 (2017). arXiv:1706.09357

[22] Donald Knuth. 2009. *The Art of Computer Programming, Volume 4, Bitwise Tricks & Techniques; Binary Decision Diagrams*. Pearson Education, Reading, Massachusetts.

[23] Mike Mannion. 2002. Using First-Order Logic for Product Line Model Validation. In *2nd International Conference on Software Product Lines*. Springer-Verlag, London, UK, 176–187.

[24] Mike Mannion and Javier Camara. 2004. Theorem proving for product line model verification. *Lecture Notes in Computer Science* 3014 (2004), 211–224. https://doi.org/10.1007/978-3-540-24667-1_16

[25] Marcilio Mendonca, Andrzej Wasowski, and Krzysztof Czarnecki. 2009. SAT-based analysis of feature models is easy. In *Proceedings of the 13th International Software Product Line Conference*. Carnegie Mellon University, 231–240.

[26] Camille Salinesi, Colette Roll, and Raúl Mazo. 2009. Vmware: Tool support for automatic verification of structural and semantic correct- ness in product line models. (2009), 173–176 pages. http://www.vamos-workshop.net/proceedings/VaMoS_2009_Proceedings.pdf

[27] Steven She. 2008. *Feature model mining*. Master's thesis. University of Waterloo.

[28] Steven She. 2013. *Feature model synthesis*. Ph.D. Dissertation. University of Waterloo.

[29] Steven She and Thorsten Berger. 2010. *Formal semantics of the Kconfig language*. Technical Report. University of Waterloo.

[30] Jing Sun and Yuan Fang Li. 2005. *Formal Semantics and Verification for Feature Modeling*. Technical Report.

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

[32] Reinhard Tartler, Julio Sincero, Christian Dietrich, Wolfgang Schröder-Preikschat, and Daniel Lohmann. 2012. Revealing and repairing configuration inconsistencies in large-scale system software. *International Journal on Software Tools for Technology Transfer* 14, 5 (2012), 531–551.

[33] P. Trinidad, D. Benavides, A. Durán, A. Ruiz-Cortés, and M. Toro. 2008. Automated error analysis for the agilization of feature modeling. *Journal of Systems and Software* 81, 6 (jun 2008), 883–896. https://doi.org/10.1016/j.jss.2007.10.030

[34] P Trinidad, D Benavides, and A Ruiz-Cortés. 2006. A first step detecting inconsistencies in feature models. In *CAiSE Short Paper Proceedings*.

[35] G. S. Tseitin. 1983. *On the Complexity of Derivation in Propositional Calculus*. Springer Berlin Heidelberg, Berlin, Heidelberg, 466–483.

[36] Thomas von der Massen and Horst Lichter. 2004. RequiLine: A Requirements Engineering Tool for Software Product Lines. *Lecture Notes in Computer Science* 3014 (2004), 168–180. https://doi.org/10.1007/978-3-540-24667-1_13

[37] Martin Walch, Rouven Walter, and Wolfgang Küchlin. 2015. Formal Analysis of the Linux Kernel Configuration with SAT Solving. In *17th International Configuration Workshop*. University of Helsinki, Helsinki, Finland, 131–137.

[38] Hai Wang, Yuan Fang Li, Jing Sun, Hongyu Zhang, and Jeff Pan. 2005. A semantic web approach to feature modeling and verification. In *In Workshop on Semantic Web Enabled Software Engineering*. 44.

[39] Haiyan Zhao Wei Zhang, Hua Yan and Zhi Jin. 2008. A bdd-based approach to verifying clone-enabled feature models' constraints and customization. *Lecture Notes in Computer Science* 5030 (2008), 186–199.

[40] J. White, D. C. Schmidt, D. Benavides, P. Trinidad, and A. Ruiz-Cortés. 2008. Automated Diagnosis of Product-Line Configuration Errors in Feature Models. In *Proc. 12th Int. Software Product Line Conf.* IEEE, 225–234. https://doi.org/10.1109/SPLC.2008.16

[41] Hua Yan, Wei Zhang, Haiyan Zhao, and Hong Mei. 2009. An Optimization Strategy to Feature Models' Verification by Eliminating Verification-Irrelevant Features and Constraints. In *Formal Foundations of Reuse and Domain Engineering*, Stephen H. Edwards and Gregory Kulczycki (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 65–75.

[42] Lamia Abo Zaid, Frederic Kleinermann, and Olga De Troyer. 2009. Applying semantic web technology to feature modeling. In *Proceedings of the 2009 ACM symposium on Applied Computing*. ACM, 1252–1256.

[43] Christoph Zengler and Wolfgang Küchlin. 2010. Encoding the Linux kernel configuration in propositional logic. In *Proceedings of the 19th European Conference on Artificial Intelligence (ECAI 2010) Workshop on Configuration*, Vol. 2010. IOS Press, Amsterdam, The Netherlands, 51–56.

[44] Wei Zhang, Hong Mei, and Haiyan Zhao. 2006. Feature-driven requirement dependency analysis and high-level software design. *Requirements Engineering* 11 (2006), 205–220. https://doi.org/10.1007/s00766-006-0033-x

[45] Wei Zhang, Haiyan Zhao, and Hong Mei. 2004. A propositional logic-based method for verification of feature models. In *International Conference on Formal Methods and Software Engineering*. Springer, 115–130.