Configuration Lifting: Verification meets Software Configuration

Hendrik Post, Carsten Sinz
Research Group "Verification meets Algorithm Engineering"
Institute for Theoretical Computer Science, University of Karlsruhe, Germany
http://verialg.iti.uka.de {post,sinz}@ira.uka.de

Abstract

Configurable software is ubiquitous, and the term Software Product Line (SPL) has been coined for it lately. It remains a challenge, however, how such software can be verified over all variants. Enumerating all variants and analyzing them individually is inefficient, as knowledge cannot be shared between analysis runs. Instead of enumeration we present a new technique called lifting *that converts all variants into a meta-program, and thus facilitates the configuration-aware application of verification techniques like static analysis, model checking and deduction-based approaches. As a side-effect, lifting provides a technique for checking software feature models, which describe software variants, for consistency.*

We demonstrate the feasibility of our approach by checking configuration dependent hazards for the highly configurable Linux kernel which possesses several thousand of configurable features. Using our techniques, two novel bugs in the kernel configuration system were found.

1 Introduction

Formal software verification plays an increasingly important role in the software development process. It is gradually evolving into a tool with much more widespread use and is on the edge of becoming a natural companion to well-established testing methods. Microsoft's *Static Driver Verifier (SDV)* [1], for example, is a tool that automatically checks C code of Windows device drivers for conformance to the Windows Driver Model (WDM). However, complete software systems not only consist of a bunch of source code files, but also contain various dimensions of versioning. For low-level code this extra information comes in the form of makefiles, configuration files, or shell scripts, and these additional files are generally not considered during formal verification.

Software configuration provides another source of complexity for software testing and verification. A configuration space consisting of n features that may be turned on and off independently requires analysis of (up to) 2^n concrete configuration instances. And with conventional software testing already facing the problem of incomplete coverage of possible program executions, adding an exponential factor to the set of possible program instances may completely break the testing approach. Software verification techniques like Model Checking have the potential to cover all possible program executions but are currently incomplete—or even sound—because configurations are not taken into account.

The new technique, called *lifting*, integrates configuration information into a conventional verification process. In this paper, this novel approach is introduced, concentrating on applying it to build configurations and conditional compilation techniques. Lifting can be described as automatically transforming a configurable software system such that all configuration steps are performed at runtime of the program.

Lifting allows standard verification techniques to reason about the following typical defect areas in a generative [3] software build process:

D1. The feature model may be inconsistent.

D2. The feature model restrictions may not match the restrictions required by the concrete, implemented features.

D3. *Any* variant may violate software requirements that should hold for all variants, e.g., it may cause runtime errors.

As verification backend we have chosen CBMC [2], a SAT-based source code bounded model checker for C.

1.1 Lifting

In this paper we consider configurable programs (SPLs) consisting of the following parts:

- A feature model consisting of (propositional logic) rules over a finite domain language that describes valid software variants (e.g., Kconfig for the Linux kernel).

- Configurable source code (configuration is done, e.g., by preprocessor directives).
- A tool chain for configuring a variant consistent with the feature model as well as for building the correct software variant (makefiles, C preprocessor).

The tool chain is also called a *Software Variant Generation System (SVGS)*. The main idea of lifting now is as follows: We assume that a verification tool for checking individual variants is already available. To handle different software variants, we have to integrate relevant parts of the configuration and build process into our verification approach. This is done by converting the configuration and build parts into the source code language of the configurable program (written, e.g., in C), and modifying the original program accordingly. Then the assume/assert mechanism of the verification backend can be used to reason about multiple configurations as well as properties of the configuration system. In that way, we can check, e.g., that a property holds for *all* valid software variants. We will also denote the modified program containing the configuration and build parts (thus encompassing all valid variants) by *meta-program*.

The impact of variant generation on the source code level can be modelled by conditional code inclusion/exclusion[1]. Each inclusion/exclusion is then guarded by a condition that depends on the feature model and build tool chain. For example, a function definition may be included in a C program only if the corresponding feature is enabled, the makefile includes this file (this may also depend on the selected features) and all relevant preprocessor conditions (`#ifdefs`) evaluate to true. The challenge for configuration-aware verification is now twofold: the set of guards must be computed by analyzing the feature model, Makefiles, and preprocessor statements. Additionally, guards must be expressed in a way such that the verification backend can handle them. One way to achieve the latter is to encode guards as path conditions as will be illustrated in the next section.

Compared to checking each software variant independently, lifting is expected to work well if the variants share large parts of the project's source code, which is typically the case. The benefits of lifting are that configuration independent knowledge must be generated only once, and that program errors that occur in many software variants can be detected efficiently.

2 Case Study: Linux as a SPL

Sincero *et al.* state that Linux is in fact a software product line [4]. Variant generation in Linux is mainly based on conditional compilation of C code. The code, organized in thousands of modules, is conventionally configured using the following means: An *Architecture Definition*, a rule-based configuration system called *Kbuild* which encodes a feature model, a set of *Makefiles*, *Preprocessor Statements* that may exclude, include, or modify source code on any syntactical level and *Runtime Parameters* for modules and the kernel itself.

In this section we apply lifting to the configuration of the Linux kernel. This is done in several steps, which resemble the steps that are required to configure a kernel. At first, the Kbuild feature space and rules is translated into C. After this step, problems belonging to class D1 can already be analyzed. Next, Makefiles are lifted to express their semantics in C. Having done that, problems of class D2 can be expressed—e.g., to rule out that functions are used but not defined in certain configurations. Finally, the effects of arbitrary preprocessor statements are encoded into preprocessor free C code, which enables the use of verification tools like C software model checkers to gain full access to the configuration process as a whole. The latter transformation relates to problems belonging to class D3.

Lifting Kbuild. Kbuild allows feature models to be defined as a set of (boolean) features and rule-based restrictions. Features may depend on each other. For example, feature `A` may only be allowed if another feature `B` is also enabled. Such and other dependencies can be expressed by the Kbuild keywords `depends [on]` . Reverse dependencies can be defined using the `selects` attribute. Other features are omitted in this paper.

We now translate the Kbuild rules into a C-function. For all possible features a new, unconstrained variable is introduced. It is then checked programmatically whether values assigned to the variables comply with the set of propositional logic rules. For a valid configuration, the C-function calls all possible entrypoints (i.e., `main`) of the original source code. If the configuration does not comply, the program terminates gracefully to avoid errors resulting from variants executed under illegal configurations.

Example 1 *Kbuild feature model consisting of the sole rule* `A depends on B` *lifted into a C function.*

```
void feature_model() {
  _Bool invalid = false, feature_A, feature_B;
  invalid = invalid || (feature_A && ! feature_B);
  ...
  if (!invalid) main(...);
  exit(RETURN_CONFIG_INVALID);
}
```

Note that the above example encodes the dependencies between different feature models as a path constraint.

The above transformation is implemented by a script written in `awk` which translates the ruleset into C code. The C code can then be analyzed using the bounded source

[1] Code exclusion/inclusion may transform any program into any other program, hence no loss of expressive power occurs.

Table 1. Kbuild feature model analysis, i386 architecture.

Features	4675
Dependencies	3640
Selects	1830
Variables in SAT formula	142004
Clauses in SAT formula	280629
Size of program expression (assignments)	15233
Analysis runtime	108s

Table 2. Analysis of undefined functions, PowerPC architecture (for `drivers/base` and `drivers/macintosh`).

Source files	68
LOC	≈31000
Non-trivial assertions to be checked	50
Features	4495
Makefiles	952
Conditions in Makefiles	10489
Errors found	5
New errors	1
Makefile analysis runtime	25s
Source parsing, condition extraction runtime	30s
Create verification conditions	10s
Analysis runtime	≤1s

code model checker CBMC [2]. In our experiments with the Linux kernel we addressed the following problem (from class D1):

Problem 1 *May a feature be enabled, although its dependencies are not fulfilled?*

This problem arises from the Kbuild implementation of reverse dependencies[2]: Kbuild allows a feature `A` to be activated by another feature `B` without ensuring that `A`'s dependencies are met, as `selects` automatically enables all required features.

Modelling this problem can be done in finite domain logic, but state updates must also be modelled. C as a modelling language is of course expressive enough to handle this. The resulting C program that encodes the problem is similar to Example 1 except that two steps replace the invocation of a main function: (1) Apply all `selects` by updating configuration variables if the feature that contains the `select` is enabled. (2) Assert that the new configuration still complies with all dependency rules.

The above steps model the effect of reverse dependencies introduced by `select` statements: CBMC then checks whether the assertion holds or not, and, if the latter should be the case, a counter-example trace will be printed. The counter-example gives information about the first dependency that has been violated. The above analysis lead to the discovery of an error. Details about the performed analysis are given in Table 1.

[2]Reverse dependencies activate dependent features while normal dependencies cannot be enabled when their dependencies are not met.

Lifting Make. Makefiles direct the compilation process by determining which source files (and in which order) have to be compiled and linked together. In Linux, the set of files and directories which are included into a kernel build is determined by configuration flags. For each source file we compute all conditions that must be fulfilled such that `make` would compile and link it. This condition is usually encoded recursively: an item is included if its guard expression is true and its parent directory is also built. Hence, the resulting guard is a conjunction of guards for this file and all directories on the file's path.

Problems of class D2 are related to compile problems that arise for certain configurations. We decided to pick one type of problem that has been reported on the Linux kernel mailing list:

Problem 2 *A function may be used in configurations in which it is not defined.*

Whether a C function is defined may depend on configuration features. It may occur that calls to this configuration-dependent function do not have sufficient guards, e.g., calls to undefined functions may exist for some configurations. Let G_D denote the guard for the definition of a function, and G_C denote the guard under which any call to the function may occur. The system is safe with respect to the above problem if $G_C \Rightarrow G_D$.

The problem extends Problem 1 in so far, as information about makefile and preprocessor guards must be included. A function is defined or used if the file is included by make and the definition is not masked by `#ifdef` guards.

If $G_C \Rightarrow G_D$ holds for all valid configurations, then the function call can be considered safe. Using this technique, we were able to re-discover four known configuration errors. Moreover, we found another unsafe call/definition pair that has been fixed by the Linux kernel developers independently from our work.

Table 2 gives more details on the performed experiments.

Lifting Preprocessor Code. The C preprocessor implements code modifications. The preprocessor gets as input the macro declarations created by Kbuild. These and other information are then used to modify the source code.

In order to understand the dependencies between Kbuild rules, makefiles and preprocessor directives we introduce a new term called *preprocessor (code) region*. A preprocessor region is a maximal nonempty section of C source code

that does not contain any preprocessor statements like `#if`, `#ifdef`, `#else`, `#endif` and others.

For each region we compute the preprocessor condition (*guard*) that must hold for it to be active (i.e., the condition under which it is included in the compilation process). The relationship between features and region guards is that conditional compilation uses preprocessor variables that are defined only if certain features are included in the current configuration. Consider, as a simple example, a statement like `x = 5/y`, embraced by an `#ifdef DIVISION_ENABLED`. The division is only included in the source file if the preprocessor symbol `DIVISION_ENABLED` is defined. Kbuild sets `DIVISION_ENABLED`, which bridges the gap between feature model and preprocessor evaluation.

Commonly the following aspects of a C program are changed using configuration dependent preprocessing: code blocks or individual statements, definitions, declarations, types and initializers. We will now show how declarations can be lifted. Other configuration dependent elements are treated similarly: If a declaration is configuration dependent—i.e., it is located in a region which has a non-trivial guard—we modify its name by adding to it a new postfix. Any direct reference to the original variable is replaced by a conditional reference. Variant bound declarations can now be referenced in the meta-program. Removing all preprocessor effects enables configuration aware verification. The corresponding problem (from class D3) is straightforward:

Problem 3 *Does a set of variants contain runtime errors?*

As a micro case study, we transformed a real Linux device driver, `sound/oss/ad1848.c`, using our technique (we had to manually preprocess several lines of the original code). Note that afterwards the meta driver was still compilable using the custom kernel make process.

The driver contains five binary configuration features that are independent from each other, i.e., 32 (almost identical) variants exists. We started with the smallest variant and incrementally lifted features one after another. Thereby, we introduced 56 new `if` statements that guard configuration dependent operations or names. No new function calls or loops were introduced, hence the amount of code covered by CBMC was as high as before. Our conclusion is that, at least for this driver, lifting may provide a substantial speedup compared to naively checking each variant separately (Table 3). Note that the fastest runtime was achieved for the smallest variant that excludes PNP, SMP and timer support (ET). If the analysis of each possible variant takes at least as long as for the minimal variant, the total conventional analysis time would be $32 \times 71s \approx 37$min. Lifting can analyze all variants (cf. last line in Table 3) in 73s. The speedup is dependent on the fact that most lines of code are not influenced by configuration.

Table 3. Verif. of lifted code (`ad1848.c`).

Lifted Feature	Variants	Time	CBMC / SAT problem size #Vars	#Clauses
Minimal code	1	71.0s	$8 * 10^6$	$13 * 10^6$
+MODULES	2	74.6s	+1	+3
+DEBUGXL	4	75.6s	+25657	+81983
+ET	8	75.4s	+3	-295
+CONFIG_SMP	16	72.9s	+117704	+185367
+CONFIG_PNP	32	73.0s	+4	+13975

3 Discussion

Software configuration lifting, as presented in this paper, is a novel technique. We have shown that it can be applied to very complex software systems featuring many levels of configuration. Lifting facilitates the analysis of specifications in three domains: inconsistencies within a feature model, inconsistencies between feature model and feature implementations and even the coverage of runtime errors in all product variants. Though the presented technique is centered on configurable code using conditional compilation mechanisms, it should in principal be applicable to other SPL implementation mechanisms. The usage of the target language as modelling language for the generative process is expressive enough to model complex generation procedures. The three domains have been tested on a very large system including more than 4600 features. The computationally hardest part, model checking the meta-program, did provide a substantial speed-up compared to enumeration based analysis runs. We were able to expose new bugs attached to uncommon configurations. Finding those would potentially have required much more effort if all configurations were checked separately.

References

[1] T. Ball, E. Bounimova, B. Cook, V. Levin, J. Lichtenberg, C. McGarvey, B. Ondrusek, S. K. Rajamani, and A. Ustuner. Thorough static analysis of device drivers. In *EuroSys Conf., Proc.*, pages 73–85. ACM Press, 2006.

[2] E. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In *TACAS*, volume 2988 of *LNCS*, pages 168–176. Springer, 2004.

[3] K. Czarnecki and U. Eisenecker. *Generative Programming. Methods, Tools and Applications.* Addison Wesley, May 2000.

[4] J. Sincero, H. Schirmeier, W. Schröder-Preikschat, and O. Spinczyk. Is The Linux Kernel a Software Product Line? In *Intl. Workshop on Open Source Software and Product Lines (SPLC-OSSPL 2007), Proc.*, 2007.