Facing the Linux 8000 Feature Nightmare

Julio Sincero (PhD student), Reinhard Tartler (PhD student), Christoph Egger (student),
Wolfgang Schröder-Preikschat, Daniel Lohmann
{js,rt,siccegge,wosch,dl}@cs.fau.de
Friedrich-Alexander University Erlangen-Nuremberg

I. Introduction

System software, especially operating systems, tends to be highly configurable. The relatively small eCos operating system for embedded applications [1] already offers more than 750 features; current versions of Linux provide even more than 8000 configuration options – which technically are implemented and enforced in the code by means of preprocessor macros.

To assist the user on his path through this variability, both systems additionally employ a dedicated *variability model*, together with some interactive configuration tool (such as *qconf* for Linux), which not only organizes, lists, and explains these options, but also enforces a myriad of inter-feature constraints and sanity checks to guarantee the outcome of a sound configuration.

These two views onto the same system (variability *model* vs. variability *implementation*) are a nightmare with respect to maintenance and evolution. Without sophisticated tool support, they quickly lead to "dead" code and "zombie" features because of inconsistencies. Our initial analysis of just the *referential* integrity between both worlds in Linux has uncovered around 400 bugs – and we thereby have just touched the tip of the iceberg!

A. Variability in Linux

The Linux kernel employs its variation points at three different levels:

Model Level: The *Kconfig* tool set was especially written to support the modeling of features (also called *config options*) and interdependencies of the Linux kernel. In the version `2.6.30`, a total of 534 `Kconfig` files are employed, consisting of 88,112 lines of code that describe 8063 features and their dependencies. The user configures a Linux kernel by selecting features from this model. During the selection process, the *Kconfig* configuration utility implicitly enforces all dependencies and constraints, so that the outcome is always the description of a valid configuration. Technically, this description is given as a C-style header file that defines a `CONFIG_xxx` preprocessor macro for every selected feature.

Generation Level: Coarse-grained variability is implemented on the generation level. The compilation process in Linux is controlled by a set of custom scripts called *Kbuild* that interpret a subset of the `CONFIG_xxx` flags (generated with the *Kconfig* tool set) and drive the compilation process by selecting which compilation units should be compiled into the kernel, compiled as a loadable module, or not compiled at all.

Implementation Level: Fine-grained variability is implemented by conditional compilation using the C preprocessor. The source code is annotated with preprocessor directives (like `#ifdef CONFIG_xxx`), which are evaluated in the compilation process. The force inclusion mechanism (option `-include`) of the `gcc` compiler guarantees that every source file is compiled with the set of flags defined during configuration.

As of this writing, the Linux kernel has around 3 million lines of code, 20 thousand lines of code are changed per day, and there are around one thousand developers involved in each release, the daily changes include refactoring/addition/removal of features at model level and at source code level.

B. Problem Statement

This high volume of code changes makes keeping the consistency of the variability in the three levels very challenging.

We have identified problems that both users and developers may face during configuration or development. For example: (1) during configuration, the selection of a *config option* may be impossible, or may result in no effect in the generated kernel image. (2) for testing, the developer may not be able to find a valid configuration that enables the specific conditional block she wants to test.

Consider the patch below, which fixes a typo in a Linux configuration flag regarding the *hot CPU plugging* feature.

```
diff --git a/kernel/smp.c b/kernel/smp.c
index ad63d85..94188b8 100644
--- a/kernel/smp.c
+++ b/kernel/smp.c

-#ifdef CONFIG_CPU_HOTPLUG
+#ifdef CONFIG_HOTPLUG_CPU
```

The consequence of this typo (which remained undetected for six months) was that when the user selected this feature during configuration, the generated kernel image did not contain the required code – CPU plugging had not been working under some circumstances. Imagine the sysadmin who has to figure out the source of this bug on his 7/24 system.

II. The Life Approach

Our approach aims at closing the gap between the different views and related that control, define, or use *config options*. We have devised four consistency conditions in order to automatically detect configuration problems on the code base of the Linux kernel:

- Referential conditions:
  1) Every reference to a *config option* in the source code must be defined in the *Kconfig* model.
  2) Every *config option* defined in the *Kconfig* model must be referenced in the source base.
- Semantic conditions:
  3) Every single *config option* in the source code must be *selectable* in the configuration process provided by the *Kconfig* tools.
  4) Every `#if` branch that contains *config options* must be enabled by at least one valid configuration derived from the *Kconfig* tools.

Configuration problems either pollute the code base (in case of dead code blocks) or lead to actual bugs (the wrong selection of a conditional code block). Our approach is to use static analysis tools on the different variability models. Because our preliminary estimations show that currently these consistencies are largely unregarded, we believe that by consequently checking conditions the quality of the code base of the Linux kernel can be vastly improved.

We envision tool support to support Linux engineers to completely understand the *impact* of each feature on the code base. For example, answering questions like "*What are the consequences of deleting FEATURE_X from the Kconfig model?*" implies determining which compilation units would be disregarded and locating methods, functions, fields in structs, and so on that would be affected by a refactoring. Our infrastructure is designed to deal with such issues as part of the LINUX FEATURE EXPLORER– LIFE framework. We are optimistic that our approach avoids the current tedious, and – as our preliminary results show – error-prone process.

III. THE LIFE TOOL CHAIN

While the explorative functionality is still future work, we focus at this point on the detection of configuration problems. The LIFE tool chain aims at extracting the configuration-related information from the Linux kernel to a common (model) representation, so that the consistency among the involved parts can be evaluated in an automated fashion. This is done by translating these two kinds of variability models – the configuration model from *Kconfig* and the implementation model as implemented by `CPP` statements in the source – to propositional-logic statements. With these models, we use well studied theories like *Binary Decision Diagrams* (BDDs) and *Boolean Satisfiability Problems* (SAT) to reason about the satisfiability of our consistency conditions. Currently, our LIFE tool chain is comprised of the following tools:

source2rsf is our scanning tool that extracts all information about conditional compilation. The tool is built upon the preprocessor of the *sparse* static analysis tool, which was originally developed for the Linux's code base.

undertaker is the tool responsible for analyzing the generated `rsf` files, which implements our algorithm [2] for calculating the variability induced model by `CPP` directives.

KconfigExtractor is based on the parser of the *Kconfig* language and generates the boolean formula that represents the *Kconfig* model.

IV. PRELIMINARY RESULTS

By applying our UNDERTAKER tool to the Linux kernel, we invested around three weeks searching for configuration inconsistencies. Our reports about dead conditional blocks were well received by the Linux developers. In a nutshell, none of our findings turned out as false positives. So far, we have **14** patches accepted that are waiting for the 2.6.34 merge window; **five** have been accepted by Andrew Morton's `-mm` development tree. **Four** patches have been confirmed but fixed independently. **Three** patches have been acknowledged but are currently being discussed. **Nine** further patches have been submitted but await response. Interestingly, for **four** patches our findings have been confirmed, but the patches were rejected. For these cases it turned out that the actual development does happen in a separate development tree and it has been argued that keeping these dead blocks serve as "documentation" to help future merging.

The results are very encouraging. On the one hand they indicate that tool support is definitely needed to avoid these obvious inconsistencies. On the other hand, development styles like out-of-tree development make automatic reporting of these issues difficult. We therefore envision a semiautomatic approach in which the developer is assisted with sound information about the kinds of variability in the source artifacts he is working on.

V. CONCLUSION AND FUTURE WORK

The management of configuration-related information is crucial for system software. We have identified problems that may occur due to the lack of integration between the tools that maintain the configuration of the Linux kernel. Our approach makes use of well-studied formalisms in order to verify consistency between the several forms of variability of the Linux kernel. Our tool chain is integrated into the Linux build system in order to make it easy to use for (kernel) developers. Our preliminary results indicate that there are several very obvious inconsistencies in the code base that need to be fixed. The feedback from the Linux community has been very positive so far; also, our results seem to be accurate, as we have not yet received feedback for any false positive. We are currently investigating our results for the inconsistencies that are caused by *semantic* inconsistencies between *Kconfig* and `CPP` statements. Looking forward, we consider implementing our formalism using SAT solvers instead of BDDs to compare which realization techniques achieves the best performance. Moreover, we will carry out a detailed study of the four of consistency conditions violations found in the Linux kernel.

REFERENCES

[1] A. Massa, *Embedded Software Development with eCos*. New Riders, 2002.

[2] J. Sincero, R. Tartler, and D. Lohmann, "An algorithm for quantifying the program variability induced by conditional compilation." Friedrich-Alexander-Universität Erlangen-Nürnberg, Tech. Rep., 2010.

Facing the Linux 8000 Feature Nightmare

Julio Sincero, Reinhard Tartler, Christoph Egger, Wolfgang Schröder-Preikschat, Daniel Lohmann

Introduction

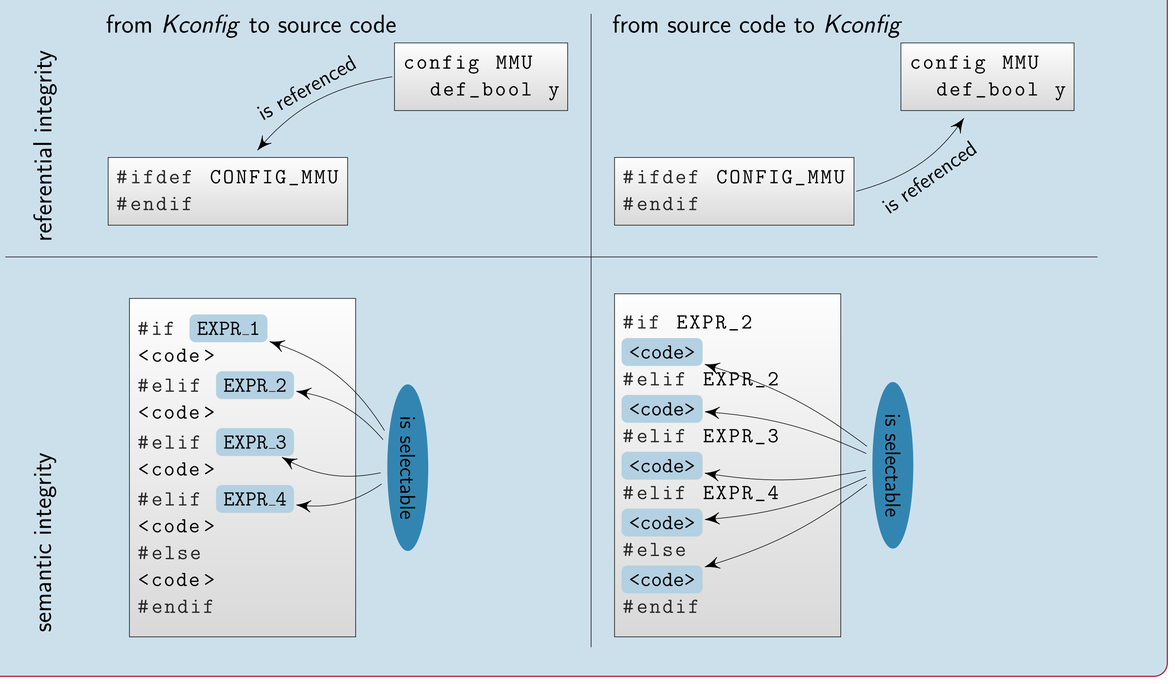
Operating systems are often highly configurable. Current versions of Linux provide even more than 8000 configuration options, which are implemented and enforced in the code by means of preprocessor annotations.

To assist the user with managing variability, Linux employs a dedicated **variability model** along with the interactive configuration tool **Kconfig**.

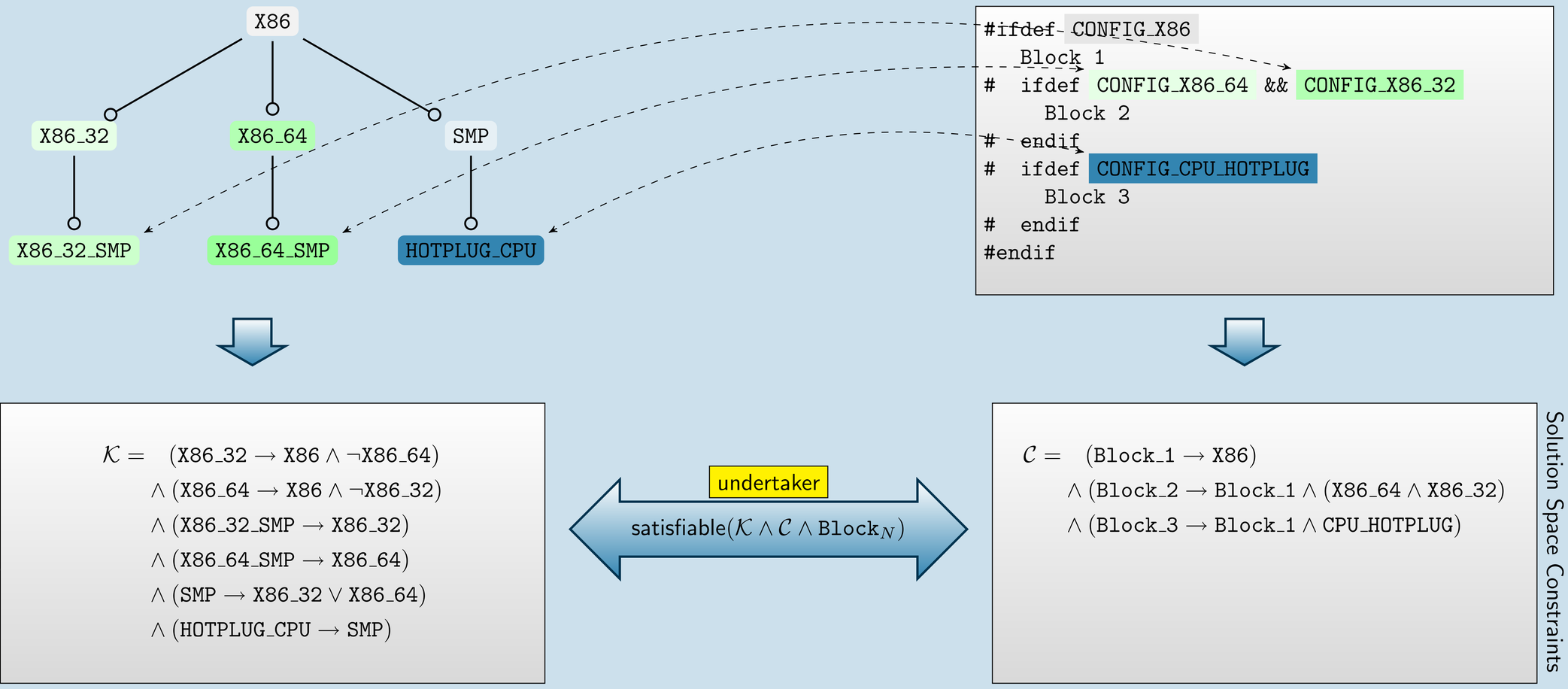
We have identified that currently developers fail to detect obvious inconsistencies between the **code base** and the **configuration options**. Therefore, we have devised a set of **consistency rules** and developed a tool chain that analyzes the source code and automatically detects any violation of such rules.

Sources of the Problem

The variability described in the Linux **Kconfig** files is conceptually interconnected with the **source code**. Both have to be kept consistent. While analyzing the Linux kernel we have discovered obvious inconsistencies, which can be classified in two dimensions:

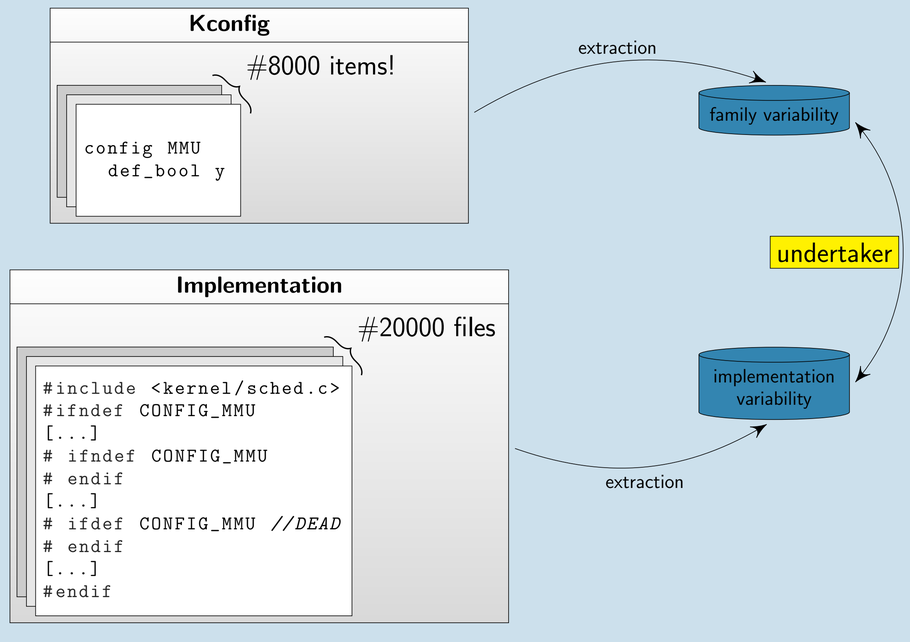


LIFE – The Linux Feature Explorer



Linux Feature Extraction and Analysis Framework

The LIFE tool chain scans the Linux code base and extracts information about the definition and use of configuration options. Using propositional logic, the undertaker transforms this into a common format such that violations of all four consistency categories can be detected reliably:



Detected Inconsistencies

So far we have detected approximately 90 consistency violations. We have been reporting them to the Linux community, and the feedback has been very positive. Many of our findings have turned out to be real defects, and the proposed fixes have already been merged into the mainline code base:

Linux	found	confirmed	merged	rejected
arch/	3	1	1	
drivers/	40	23	19	2
drivers/net/	13	4	4	
drivers/usb/	11	10	10	
net/	2	2	2	
fs/	10	5	3	1
sound/	5	4	1	3
kernel	1	0		
crypto/	1	1	1	

Friedrich-Alexander-Universität Erlangen-Nürnberg

Chair in Distributed Systems and Operating Systems