A Robust and Portable Approach for Extracting Build-System Variability

Bachelorarbeit im Fach Informatik

von

Christian Dietrich

geboren am 05.12.1989 in Rothenburg o.d.T.

Lehrstuhl für Informatik 4
Friedrich-Alexander Universität Erlangen-Nürnberg

Betreut durch:

Dipl.-Ing. Reinhard Tartler
Dr.-Ing. Daniel Lohmann
Prof. Dr.-Ing. habil. Wolfgang Schröder-Preikschat

Beginn der Arbeit: 1. April 2011
Ende der Arbeit: 30. Juni 2011

Hiermit versichere ich, dass ich die Arbeit selbstständig und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Stellen, die dem Wortlaut oder dem Sinn nach anderen Werken entnommen sind, habe ich durch Angabe der Quelle als Entlehnung kenntlich gemacht.

Erlangen, den 9. Juli 2012

Kurzzusammenfassung

Ein Build System spielt, neben dem tatsächlichen Quellcode, eine wichtige Rolle in einer Software Produkt Linie und deren Entwicklung. Es kontrolliert den Prozess des Übersetzens und Fertigstellens eines Software Produkts und implementiert einen Teil der statischen Variabilität zur Übersetzungszeit in einem statisch konfigurierbaren System. Mit mehr als 11000 optionalen und alternativen Merkmalen, ist der Linux Kernel ein solches Stück statisch konfigurierbarer Software. Das Build System des Linux Kernels, genannt KBUILD, verwaltet den Einfluss von beinahe 8000 dieser Merkmale auf den Übersetzungsprozess. Daher müssen variabilitätsgewahre Ansätze zur statischen Analyse diesen Einfluss beachten. Allerdings ist es schwierig die Variabilitätsinformationen aus solch einem Build System zu extrahieren, da sich bisher kein Standard für solche Systeme entwickelt hat. Selbst für ein spezifischen Build System ist der Extraktionsprozess schwierig, da häufig deklarative und Turing-vollständige Sprachen, wie die MAKE Sprache in KBUILD, verwendet werden. Das Problem wird noch drastischer, wenn mehr als eine Version des Build Systems betrachtet werden soll. Nachdem, in dieser Arbeit,ein robuster Ansatz zur Extraktion von Variabilitätsinformation aus dem Linux Build System beschrieben wird, wird ein verallgemeinerte Version der verwendeten Ideen hergenommen, um solche Variabilitätsinformationen auch aus anderen Build Systemen zu extrahieren. Wie die Ergebnisse zeigen, ist der Ansatz robust, in Hinsicht auf die Entwicklungsgeschichte von Linux und einfach portierbar auf andere Build Systeme.

Abstract

Build systems play, besides the actual source code itself, an important role in a software product and its development. They control the build process and implement a part of the compile-time variability within a static configurable system. With more than 11,000 optional and alternative features, the Linux kernel is such a static configurable piece of software. Its build system KBUILD manages the influence of nearly 8,000 of these features on the build process. Hence, variability-aware static analysis tools have to take the influence of the build system into account. But extracting the variability information from a build system is hard, since no real standard for such systems has emerged yet. And even for a specific build system the extraction is challenging, because often declarative and turing-complete languages, like the MAKE-language in KBUILD, are used. The problem dramatically increases when more than one version of the build system should be taken into consideration. After describing a robust approach for extracting implementation variability from the Linux build system, a more abstract version of the used ideas is presented and the approach is ported to different build systems. As the results show, the approach is robust in respect to the development history of Linux and easily portable to other build systems.

Contents

1 Introduction

At the dawn of computer science, most programs were relatively small in regard to code size, implemented features and complexity. The resources were limited and the program was hand-optimized for a specific problem. There was no operating system, because there was no need for it and all communication with the outside world was done within the manufactured program.

Over the years and with growing amounts of memory and processing power available, the programs became bigger and more complex. They solved more complex problems or even more than one problem. Code libraries – collections of useful subroutines and functions – and operating systems, which were libraries in the beginning, emerged.

The complexity, which came hand in hand with the now bigger programs, was faced by improved software engineering paradigms, like higher-level programming languages and modularization. One very essential meta-paradigm in the development of software is the wish to reuse as much code as possible, since developer hours are expensive. This leads to the phenomena that a new software product is not always written from scratch, but implemented as an additional and optional feature to an already existing software product. The new feature can use the intra-program infrastructure and, when designed properly, does also enhance the base-line software product. By augmenting the base software a software product line is born:

> A Software Product Line (SPL) is a set of software-intensive systems that share a common, managed set of features satisfying the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way.[1]

From the definition of the term *Software Product Line* we conclude that a SPL consists of a software which is partitioned into features. For a specific software product the SPL is tailored to the specific needs of the use case. In the configuration process a subset of all available features is selected. From this feature selection a concrete software product is built and shipped to the customer. The finished software product is a variant of the SPL.

1.1 Variability at different places

During the configuration process, the user of the SPL, who might be a domain expert in the application field of the SPL, has to select from the provided features. Those features might have connections between each other, like dependencies or cardinality constraints. After the selection process, the tailoring process of the SPL is started to produce the specific variant. In case of *static variability*, like compile-time variability, the build process is driven by the feature selection.

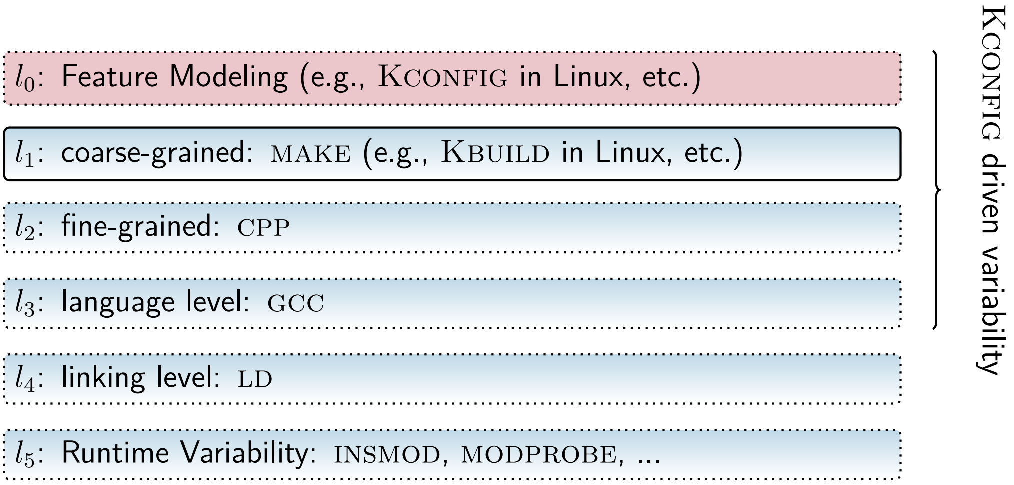

l_0: Feature Modeling (e.g., KCONFIG in Linux, etc.)
l_1: coarse-grained: MAKE (e.g., KBUILD in Linux, etc.)
l_2: fine-grained: CPP
l_3: language level: GCC
l_4: linking level: LD
l_5: Runtime Variability: INSMOD, MODPROBE, ...
KCONFIG driven variability

Figure 1: Abstract overview over the dominance hierarchy of variability implementations (taken from [2])

The selected features have influence on the inclusion of source-code files and the operation of preprocessor, compiler and linker.

Two aspects of feature handling in SPLs can be distinguished. The first aspect is the *declaration of features*. This declaration might be implicit or explicit. Implicit feature declarations are for example preprocessor macros in a header file. In contrast is a feature declaration language, like KCONFIG, which is interpreted by a configuration tool, an explicit declaration. Whether the declaration is implicit or explicit, it states the *intentional feature model*.

The second aspect of feature handling is the *extensional feature model*. The extensional feature model is formed by all points in the SPL where a specific feature selection influences the produced software product. Such a point will be called a *variation point (VP)*. A VP is influenced by a subset of all declared features and their selection state.

VPs may appear at different grained levels [2]. It may control, on a *coarse-grained level*, whether a source-code file is compiled into the product. But it might also control, on a *fine-grained level*, whether a specific source-code statement is processed by the compiler.

1.2 Variability in Linux

To illustrate these different levels of variability, and how the extensional side is controlled by the intentional side, I give a short overview over the variability in the Linux kernel: Linux is a SPL with a big amount of static variability. The features are declared on level l_0 with a domain specific language, which is parsed and interpreted by KCONFIG. KCONFIG, and its various user frontends, is also the name of the configuration tool, which is used by the domain expert to select the feature subset for the specific variant of Linux.

Static VPs are present on the coarse-grained level l_1 in the KBUILD build system and on the fine-grained level l_2, encoded with the C Preprocessor (CPP) [2]. On the coarse-grained level the inclusion of whole files into the software variant is influenced by the feature selection. On the fine-grained level the inclusion of single preprocessor blocks, which may boil down to the sub-statement language-level, is controlled. The other layers ($l_3 - l_5$) are only partially, or indirect, influenced by the static selection of features and represent the run-time variability (or dynamic variability) of a compiled Linux kernel.

The different variability levels in Linux and their interaction has been observed as a source of bugs [3, 4]. Inconsistencies between the intentional and the extensional side may lead to a VP that cannot be enabled or disabled and is therefore no real *variability* point. The different levels of variability make it also hard for a static analysis tools to cover all possible code paths [5]. Efforts were done towards variability-aware parsing of the fine-grained level and variability-aware type checking [6, 7].

A variability model is a collection of all VPs in a given part of the declaration or implementation and contains all necessary constraints between the VPs. Therefore a variability model, as well for the intentional as for the extensional side, is necessary to find inconsistencies and achieve full configuration coverage. Even for the variability-aware parsing approach, which handles only l_2, variability models for l_0 and l_1 are needed. Variability models for the intentional side was done by She and Berger [8]. On the extensional side it was surprising, that about 60% of the VPs in Linux are located on the coarse-grained level l_1 [2], and not as assumed before on the fine-grained level l_2. This observation leads to the conclusion that a proper build-system model is essential for further analyses.

1.3 Build-system Variability-Models

The Linux build system is written as a collection of MAKE [9] scripts, which are known under the name KBUILD. They provide a simple interface for the programmer to make a source file dependent on a certain feature. But since the fragments that encode these VPs, are also MAKE scripts, the whole functionality of MAKE might be – and is – used.

In the case of this build system a source-code file and the constraints under which the file is considered during the build process represent a VP. A few methods were proposed to extract the variability model from the MAKE scripts. These approaches are based on parsing the MAKE scripts directly. But since the MAKE syntax is very complex and the language turing-complete – it does even support execution of arbitrary UNIX-commands – the parsing is arbitrarily hard and tailored to the specific idioms that are used within KBUILD at the time of writing the parser.

In the following chapters of this work, a non parsing based approach for extracting variability models from build systems is presented. It exploits the build system itself, is robust in respect to the development cycle of Linux over many years, and is easy to adapt to other systems that encode VPs, when they are structured in a similar manner to KBUILD.

The thesis is structured as following: Section 2 is a paper that describes the non parsing approach and applications for it on the Linux build system. Section 3 gives an abstract build-system model and elaborates on a common version of the non parsing approach. Section 4 add various extensions to the common version of the approach. Section 5 presents three case studies of the approach. Section 6 collects further related work, additionally to the related work that is already mentioned in Section 2. Section 7 concludes the thesis.

2 A Robust Approach for Variability Extraction from the Linux Build System

I have described the non-parsing based approach together with Tartler, Schröder-Preikschat and Lohmann [10] in a paper that was accepted to the *Software Product Line Conference 2012*. Writing this conference paper was part of this thesis. It was directly accepted with four peer reviews.

In the paper i describe a probing-based approach, that exploits the build system itself to calculate a variability model for the extensional VPs in the Linux build system KBUILD. Afterward the result is compared to two parsing-based approaches (Nadi and Holt [11] and Berger et al. [12]) and it is shown that the probing-based alternative is much more robust in respect to the development history of Linux over many years. Additionally two applications for a build-system variability-model is given, and it could be shown that these models highly improve the results of these applications.

The paper is concentrated on the Linux build system, but the idea is easily applicable to other systems that encode variability similar to KBUILD. This application is done in Section 3. The rest of the section is filled by a accurate copy of the paper.

A Robust Approach for Variability Extraction from the Linux Build System

Christian Dietrich Reinhard Tartler
Wolfgang Schröder-Preikschat Daniel Lohmann
{dietrich, tartler, wosch, lohmann}@cs.fau.de
Friedrich-Alexander University Erlangen-Nuremberg, Germany

ABSTRACT

With more than 11,000 optional and alternative features, the Linux kernel is a highly configurable piece of software. Linux is generally perceived as a textbook example for preprocessor-based product derivation, but more than 65 percent of all features are actually handled by the build system. Hence, variability-aware static analysis tools have to take the build system into account.

However, extracting variability information from the build system is difficult due to the declarative and turing-complete MAKE language. Existing approaches based on text processing do not cover this challenges and tend to be tailored to a specific Linux version and architecture. This renders them practically unusable as a basis for variability-aware tool support – Linux is a moving target!

We describe a *robust* approach for extracting implementation variability from the Linux build system. Instead of extracting the variability information by a text-based analysis of all build scripts, our approach exploits the build system itself to produce this information. As our results show, our approach is robust and works for all versions and architectures from the (git-)history of Linux.

Categories and Subject Descriptors

D.4.7 [**Operating Systems**]: Organization and Design; D.2.9 [**Management**]: Software configuration management

General Terms

Design, Experimentation, Management

Keywords

Configurability, Maintenance, Linux, Build Systems, Kbuild, Static Analysis, VAMOS

1. INTRODUCTION

System-software product lines usually employ compile-time configuration as a simple and widely used technique for tailoring with respect to a broad range of supported hardware architectures and application domains. A prominent example is the Linux kernel. The Linux KCONFIG feature model provides more than 11,000 configurable features in Linux v3.2. The thereby described *intended* variability is implemented by 28,000 source files containing 84,000 `#ifdef`-blocks.

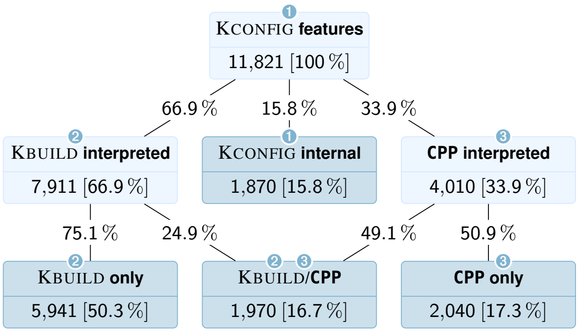


Figure 1: Statistic how the features, declared in KCONFIG, are referenced by source-code and Makefiles in Linux v3.2

In previous work, we could show that intended and actually implemented variability (i.e., the KCONFIG feature model and the variability points in the code) do not necessarily match. However, many configurability-related defects, such as dead `#ifdef`-code, and bugs, can be found upfront by better tool support [29]. This eventually has led to (accepted) fixes for twenty new bugs and the removal of 5,000 superfluous lines of `#ifdef`-code in Linux v2.6.36. However, these numbers are just the tip of an iceberg. The lesson to be learned from this is: Variability has to be understood, analyzed, and tested as a system property in its own respect. For a system-software product line at the size of Linux, this requires profound and robust tool support.

1.1 The Role of the Build System

A crucial building block for variability-aware static checking tools are reliable extractors that transform the *actually implemented* variability from their various sources into a formal model. Existing studies (including our own) have mostly focused on the C Preprocessor (CPP) as a means to implement features in Linux [13, 14, 25, 28, 29]; however, in Linux, variability is mostly implemented in a more coarse-grained manner (Figure 1): Only a third (33.9%) of all features do affect the work of the CPP, that is, have an effect on the sub-file level. On the other hand, two third (66.9%) of all features are referenced in the build system (KBUILD). These features have an effect on the selection of whole files into the build process. Hence, we need robust tools to extract the implementation variability from the Linux build system.

SPLC '12, September 02 – 07 2012, Salvador, Brazil
Copyright 2012 ACM 978-1-4503-1094-9/12/09 ...$15.00.

1.2 Related Work in a Nutshell

Approaches to extract implementation variability from KBUILD have previously been published by Berger et al. [4] and Nadi and Holt [18]. A common characteristic of both approaches is that they rely on *text processing* of makefiles, that is, they employ parsing (Berger et al.) or clever regular expressions (Nadi and Holt) to extract the presence implications for Linux source files from the build scripts. However, the underlying MAKE language is a declarative and turing-complete language; its advanced features, such as the `$(eval)`, `$(shell)`, or `$(wildcard)` functions, make it notoriously difficult to analyze. If these features are used, a text-processing–based approach quickly hits its limits, since the enclosed fragments may be for instance arbitrary shell command.

Even worse from a practical point of view is, however, that the existing approaches are brittle with respect to evolutionary changes in the KBUILD system itself: To achieve good results, they have to provide explicit support for many corner cases of KBUILD analysis, which effectively tailors them for a specific Linux version and architecture. While this might be perfectly acceptable if the goal is to analyze a certain Linux version, it renders them as practically unusable as a basis for general variability-aware tool support – Linux is a moving target.

1.3 About this Paper

The contribution of this paper is a *robust* approach for extracting implementation variability from the Linux KBUILD system. Instead of text processing, our approach exploits the build system itself to produce this information. Thereby, our approach is not only simple to implement, but also robust with respect to evolutionary changes and the usage of advanced MAKE features. Our evaluation results show that our approach works for all versions and architectures from the (git-)history of Linux and reliably extracts presence conditions for more than 93% percent of all source code files. In two applications, we show that the presented implementation significantly improves our previous results on configuration defects [29] and configuration coverage (CC) [28].

The context of this work is the VAMOS [1] project, funded by the German Research Council (DFG). The goal is to provide practical tools for analysis and management of variability in system software. So far the produced tool hav produced over 100 patches that have been integrated into the Linux mainline kernel.

The remainder of this paper is structured as following: In Section 2, we introduce the background and technical context and analyze the challenges in build-system analysis. This is followed by the description of the basics of our approach in Section 3. Then, we analyze the results in Section 4, followed by two applications in Section 5. After discussing the results in Section 6 and an overview over further related work in Section 7, the paper concludes with Section 8.

2. VARIABILITY IN LINUX

The scattered nature of variability and variability implementation in Linux makes holistic reasoning challenging. In practice, the analysis of the different models, languages and representations of variability requires very specialized and sophisticated extraction tools. A solid understanding of how the Linux build system KBUILD and the configuration tool KCONFIG play together is instrumental to correctly relate variability implementations from different extraction tools. This subsection analyzes the mechanics of KBUILD and identifies the challenges for an automated extraction of variability.

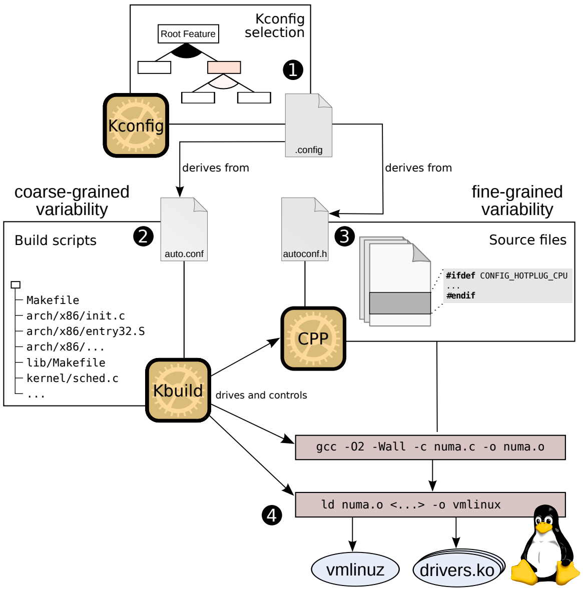


Figure 2: Overview of the technical realization of software variability in Linux. The coarse-grained varibility implemented in makefile dominates fine-grained varibility in CPP code.

2.1 Levels of Variability

In a nutshell, static configurability is specified and implemented in Linux top-down on three major levels, for which Figure 1 illustrates their quantitative relevance:

❶ The configuration system (KCONFIG) defines the available features and their constraints (*intended variability*) and provides an interface to specify and manage a concrete (product) **configuration**.

❷ The build system (KBUILD) implements *coarse-grained variability* in the code by inclusion and exclusion of complete translation units in the build process. The produced **build products** include object files, the bootable kernel image and loadable kernel modules (LKMs).

❸ The CPP implements *fine-grained variability* by inclusion or exclusion of `#ifdef`-blocks within the files selected by KBUILD.

Figure 2 describes the Linux toolchain that drives the compilation process. At the top, the Linux feature model defines the (intentional) **product line variability** [16, 20]. Here, the user selects a concrete product configuration with the KCONFIG tool and saves his selection to a file named `.config`. The Linux build system KBUILD transforms the thereby encoded feature selection into two further representations: An `auto.conf` file in MAKE-syntax and an `autoconf.h` file in CPP syntax (Figure 3). Technically, these representations control the (extensional) **software variability** [16, 27] in makefiles and C source code during the compilation process.

For the CPP representation, an additional normalization step is applied for tristate features: Many features, especially device drivers, can be configured as *compiled into kernel*, *compiled as loadable kernel module* or *disabled*. To ease the use in `#ifdef` statements, KCONFIG maps this to boolean flags by inserting an additional CPP variable with the `_MODULE`, suffix into `autoconf.h` for each tristate feature (Figure 3).

[1] **V**ariability **M**anagement in **O**perating **S**ystems

(a) KCONFIG output: .config

```
SMP=n
PM=y
APM=m
```

(b) MAKE representation: auto.conf

```
CONFIG_SMP    := n
CONFIG_PM     := y
CONFIG_APM    := m
```

(c) CPP representation: autoconf.h

```
#undef  CONFIG_SMP
#define CONFIG_PM           1
#undef  CONFIG_APM
#define CONFIG_APM_MODULE   1
```

Figure 3: Representation of a feature selection

In Step ❷, the MAKE representation of the current feature selection is then used by KBUILD to implement the coarse-grained variability on a per-file basis. All thereby included translation units are passed to the compiler, which in turn uses the CPP representation during preprocessing (Step ❸) to implement the *fine-grained* configurability. The invocation of the compiler and linker is, however, controlled by KBUILD,[2] which again uses the MAKE representation of the current feature selection to construct compiler and linker options used in Step ❹ for creating the build goals: The `vmlinuz` kernel image and the library of loadable driver objects.

2.2 Variability Implementation in Kbuild

As detailed in the previous section, KBUILD gets a file `auto.conf` that describes all selected features and their values in MAKE syntax. KBUILD then resolves which file implements what feature, determines the set of translation units that are relevant for a given configuration selection, and invokes the compiler for each translation unit with potentially configuration-dependent settings and compilation flags. Internally, KBUILD employs GNU MAKE [26] to control the actual build process; in Linux v3.2 the mapping from features to translation units is encoded in 1,568 makefiles that are spread across the source tree. However, Linux makefiles look quite different from typical text-book makefiles as they employ KBUILD-specific idioms to implement Linux-specific (variability) requirements, such as [11]:

- **Optional features:** Many features, such as drivers, are present (or absent) by deciding about the inclusion of their respective implementation files.
- **Tristate features:** Linux allows most drivers to be compiled either statically into the kernel or as LKM.
- **Loose coupling:** The decision about what set of files is used for a given configuration can be specified at various levels of granularity (such as disabling a complete subsystem by not descending a subdirectory).

In the following, we provide further details on these idioms, as they are relevant for this paper.

2.2.1 *Optional and Tristate Features*

In all makefile fragments, we can find two variables that collect selected and unselected object files: The make variable `obj-y` contains the list of all files that are to be statically compiled into the kernel. Similarly, the variable `obj-m` collects all object files that will be compiled as LKM. Object files in the make variable `obj-n` are not considered for compilation. The suffixes `{y,m,n}` are added by the expansion of variables from `auto.conf` (Figure 3).[3] This pattern for managing variability with KBUILD is best illustrated by a concrete example:

```
1  obj-y              += fork.o
2  obj-$(CONFIG_SMP) += spinlock.o
3  obj-$(CONFIG_APM) += apm.o
```

In line 1, the target `fork.o` is unconditionally added to the list `obj-y`, which instructs KBUILD to compile and link the file directly into the kernel. In line 2, the variable `CONFIG_SMP`, which is taken from the KCONFIG selection, controls the compilation of the target `spinlock.o`. The variable derives from the feature `SMP`, which is declared as `boolean`. Therefore, `spinlock.o` cannot be compiled as LKM. When the feature selection from Figure 3 (b) is applied, `CONFIG_SMP` has the value `n`, `spinlock.o` is added to `obj-n` and therefore not compiled. In line 3 the file `apm.o` is handled in a similar way to `spinlock.o`. Because the enabling feature `APM` is declared as *tristate*, it might take value `m`. With the feature selection from Figure 3 (b), `APM` has the value `m`, therefore `apm.o` is added to `obj-m` and compiled as LKM.

Note that instead of mentioning the source files, the makefile rules reference only the resulting build products. The mapping to source files is implemented by *implicit rules* (for details, cf. [26, Chapter 10]). This mapping has to be considered for any kind of makefile variability analysis.

2.2.2 *Loose Coupling*

Programmers specify in KBUILD makefiles the conditions that lead to the inclusion of source files in the compilation process. As shown above, this commonly happens by mentioning the respective build products in the special targets `obj-y` and `obj-m`. This works for the majority of cases, where a feature is implemented by a single implementation file. However, in order to control complete subsystems, which generally consist of several implementation files, the programmer can also include subdirectories:

```
obj-$(CONFIG_PM)  += power/
```

This line adds the subdirectory `power` conditionally, depending on the selection of the feature `PM` (power management). For each listed subdirectory, its containing Makefile is evaluated during the build process. This allows a more coarse-grained control of source file compilation with KCONFIG configuration options. As we will show later in this paper, the inclusion of most source files in Linux is controlled by enabling a single configuration option.

2.3 Challenges in Build-System Analysis

While the selection process described in Section 2.2 is conceptually simple, an automated analysis is challenging because of engineering reasons. Since KBUILD is implemented with the MAKE tool, the kernel developer has many possibilities to express constraints. Not only is MAKE a full-blown programming language that supports a wide range of operations, including string modifications, conditionals, and meta-programming, it also allows the execution of arbitrary further programs ("shell escapes"). The Linux coding guidelines do not pose any restrictions on what MAKE features should be used

[2] The exact mechanisms are fairly technical and have already been discussed elsewhere (e.g., [11, 17]).

[3] The idea of this pattern dates back to 1997 and was proposed by Micheal Elizabeth Castain under the working title "Dancing Makefiles" (`https://lkml.org/lkml/1997/1/29/1`). It was globally integrated into the kernel makefiles by Linus Torvalds shortly before the release of Linux v2.4.

in KBUILD. This subsection presents a few selected examples of constructs that are present in the build system of Linux and are far more expressive than the standard constructs.

The following example is taken from `arch/x86/kvm/Makefile` and uses the function `addprefix`:

```
obj-$(CONFIG_KVM_ASYNC_PF)  += \
  $(addprefix ../../../virt/kvm/, async_pf.o)
```

The `addprefix` function takes an arbitrary amount of arguments, prepends its first argument to the remaining ones, and returns them. In this case using `addprefix` is not really necessary, because there is only one additional argument and the whole expression is equal to `../../../virt/kvm/async_pf.o`. Nevertheless, this case requires special handling with a text-processing–based approach.

In KBUILD, programmers also use generative programming techniques and loop constructs, like in this excerpt taken from `arch/ia64/kernel/Makefile`:

```
ASM_PARAVIRT_OBJS = ivt.o entry.o fsys.o
define paravirtualized_native
AFLAGS_$(1) += -D__IA64_ASM_PARAVIRTUALIZED_NATIVE
[...]
extra-y += pvchk-$(1)
endef
$(foreach obj,$(ASM_PARAVIRT_OBJS),$(eval $(call
     paravirtualized_native,$(obj))))
```

Here, a list of implementation files (`ivt.S`, `entry.S` and `fsys.S`) not only need to be included, but also require special compilation flags. In this example, the macro `paravirtualized_native` is evaluated for all three implementation files by the MAKE tool at compilation-time. Again, for a text-processing–based approach, this corner case is challenging to implement in a general manner.

Even worse is the `shell` function, which makes it possible to spawn an arbitrary external program to let it control (parts of) the compilation process.

The text-processing–based approaches [4, 18] both fail on the examples shown above. Luckily – and this comes to their rescue – these MAKE language features are currently not used very frequently in KBUILD. However, they *are* used[4] and their usage is not discouraged by Linux coding guidelines. On the longer term, this implies a danger regarding the robustness of text processing as a means to extract variability information from the Linux build system. In the following, we therefore devise a pragmatic approach that, conceptually and practically, is robust with respect to these challenges.

3. EXPERIMENTAL PROBING FOR BUILD-SYSTEM VARIABILITY

In order to enable variability analyses, such as consistency checks with the KCONFIG feature model [29] or variability aware static analysis with existing tools [28], the results of the variability extractors may require a normalization step. Literature proposes propositional formulas as lingua franca for combining the different sources of variability (e.g., [4, 13, 15, 18]). Similar to [4, 18] , we extract propositional formulas that model the behavior of KBUILD, similar as we did in previous work for the CPP [24].

The set of files that KBUILD produces during the compilation process depends on selection of features done by KCONFIG. The basic idea of our approach is to (partially) execute KBUILD with different feature selections and observe the behavioral changes. This allows to correlate variability points in the feature model with the produced build products.

The presence implication of a source file is determined by the feature selections that include the file in the compilation process. Therefore, in order to extract the presence implication for a specific source file, all feature selections that enable this file need to be recorded. Our approach exploits this observation and determines for each file all selections that include the file during the compilation process.

Instead of parsing the makefile, our approach is based on "clever probing": Basically, we "ask" the build system for each feature which files it *would* built. The basic idea is to investigate a feature selection S_{base}, which uses the set F_{base} during the compilation process. Now we add one additional feature f_1 to it. The new feature selection $S_1 := S_{\text{base}} + \{f_1\}$ now compiles the set of files F_1. For every file that is in F_1 but not in F_{base} we have found a feature selection that enables this particular file.

3.1 Subdirectories

As discussed in Section 2.2.2, not necessarily all subdirectories in the Linux source tree are traversed at compilation time. Subdirectories are therefore not only used to organize files for the programmer, but also for implementing build-system variability. We address this in our approach by treating subdirectories that appear in the file sets F_{n+1} in a special way: For each subdirectory we determine the condition under which the compilation process traverses it. If the condition is non-trivial, then it is taken as precondition (the "base expression") to all presence condition of its included files. After processing all files in the file set F_n, each of the included subdirectories is processed recursively.

3.2 From Feature Selections to File Sets

Our approach relies on the following primitive operation to find the file set and all considered subdirectories that are associated to a feature selection:

$$\textit{list} : \text{Selection} \longmapsto (\text{FileSet},\ \text{SubDirs}) \qquad (1)$$

This primitive is essential for any build system that implements variability. There are several options how this can be implemented for a given build system. As a last resort, the mapping could be extracted from build traces of a real build process [cf. 2]. However, in order to avoid unnecessary compilation steps, an efficient extraction of this mapping is essential.

For KBUILD, our implementation traverses the source tree in the same way the regular compilation process. Hereby, MAKE collects all selected files and the visited subdirectories into lists (technically MAKE variables), which are used internally to drive the compilation. We make use of these implementation internals and therefore exploit the built-in KBUILD functionality to ensure an accurate operation of the *list* primitive. The full implementation is available for download from the VAMOS website [30].

As an additional optimization, our implementation ignores logical constraints that stem from KCONFIG declarations, which allows us to reduce the number of necessary probing steps. This optimization would not have been possible to implement using build traces, which (successfully) compiles and links only valid configurations.

3.3 Base Selection and Added Features

The algorithm starts with the empty selection $S_\emptyset$ as starting point for the recursion, which serves as base point for the file set and subdirectory differences. The empty selection contains no selected feature at all; it is therefore not a valid configuration according to the KCONFIG model. This base file set only includes files that

[4] In Linux v3.2, we count for `shell`: 127, `foreach`: 16, `eval`: 3, and `addprefix`: 88 occurrences.

are included in every configuration. One example of such a file is `kernel/fork.c`, which is essential for the process creation and therefore needed in every configuration.

$$(F_{\text{base}}, D_{\text{base}}) = \mathit{list}(S_\emptyset) \tag{2}$$

The files F_{base} selected by $S_\emptyset$ are unconditionally compiled into the kernel. In Linux v3.2 `arch-x86`, our implementation detects 334 such unconditional files. $S_\emptyset$ also selects the subdirectories D_{base}, which are the starting point for the build system during the source tree traversal. The presented approach uses F_{base} and D_{base} in the same manner as starting point.

In the process of adding single features to the base selection, it is necessary to know which variables have to be considered. We exploit the fact that the Linux source tree is organized hierarchically: Each conditional subdirectory carries, in addition to the base selection, a base directory d_{base}. All features referenced in the makefile of a base directory are added to the list of features to probe.

$$\mathit{features\_in\_dir} : \text{Directory} \longmapsto \text{FeatureSet} \tag{3}$$

For KBUILD, the *features_in_dir* function is straight-forward to implement with regular expressions that extract all referenced variables in the Makefile that start with `CONFIG_`. This is also some sort of text processing, but in contrast to the competing approaches [4, 18], we just extract the feature identifiers and not their (context-dependent) semantics. Therefore, the *features_in_dir* function also detects referential KCONFIG ↔ KBUILD defects, similar as described by Nadi and Holt in [18]. By excluding undeclared configuration variables from the `FeatureSet`, we reduce the number of necessary probing steps.

3.4 Build-System Probing

```
 1: function KBUILDPROBE
 2:     vDirs ← empty set                                  ▷ set of visited dirs
 3:     filePC ← empty map [ File → list [ Selection ]]
 4:     $(F_{\text{base}}, D_{\text{base}}) = \mathit{list}(S_\emptyset)$
 5:     for all $d_{\text{base}}$ in $D_{\text{base}}$ do
 6:         KBuildProbeRecursion($d_{\text{base}}$, $S_\emptyset$, $F_{\text{base}}$)
 7:     end for
 8:     for all (file, selections) in filePC do
 9:         toPC(file, selections)
10:     end for
11: end function
```

Figure 4: Starting-Point for the Build-System Probing

Figure 4 shows the recursion step over the source tree for probing the file presence implications. The recursion is done only once for each directory. In line 2, a set of already visited directories is initialized. The resulting selections for each file is stored in filePC, which holds a list of selections for each file (line 3). For each directory that is considered by the empty selection $S_\emptyset$, we start the recursion in line 6 to dig into the source tree beginning at the directory. After all file sets have been calculated, the presence implications for all source files are calculated by the helper function *toPC* in line 9.

In Figure 5, the recursion step, which is executed for every subdirectory that may be considered by KBUILD, is shown. The function *KBuildProbeRecursion* takes three arguments: The first argument d_{base} is the directory this function call should focus on. S_{base} contains the features that are necessary to visit d_{base} in the first place. The third argument is the file set associated with S_{base}. Our implementation avoids unnecessary recalculation of F_{base} by caching the result.

```
 1: function KBUILDPROBERECURSION($d_{\text{base}}$, $S_{\text{base}}$, $F_{\text{base}}$)
 2:     if $d_{\text{base}} \in$ vDirs then                          ▷ already visited
 3:         return
 4:     end if
 5:     vDirs ← vDirs ∪ $\{d_{\text{base}}\}$                        ▷ mark as visited
 6:     features ← features_in_dir($d_{\text{base}}$)
 7:     for all $f$ in features do
 8:         $S_{new} \leftarrow S_{\text{base}} \cup \{f\}$                  ▷ add one feature
 9:         $(F_{new}, D_{new}) \leftarrow \mathit{list}(S_{new})$
10:         for all file in $(F_{new} - F_{\text{base}})$ do
11:             filePC[file].append($S_{new}$)                ▷ new files found
12:         end for
13:         for all dir in $(D_{new} - D_{\text{base}})$ do
14:             KBuildProbeRecursion(dir, $S_{new}$, $F_{new}$)
15:         end for
16:     end for
17: end function
```

Figure 5: Recursion step in the Build-System Probing.

Lines 2 to 5 ensure that each directory is only visited once and the recursion terminates in finite steps. The function *features_in_dir* is called in line 6 to determine all features that are used in the directory's makefile. These features will be probed together with the base selection against this Makefile. A new feature selection S_{new} is created (line 8) as an extention to S_{base} for each of these features. For this feature selection, all considered files and subdirectories are collected by a call to *list* in line 9. The difference between the new file set and the old file set are all files that are additionally enabled by the current feature. The feature selection is added in line 11 to all additionally enabled files. Similar to this, we recurse into the file system hierarchy for each newly detected directory in line 14. The newly detected directory is used as base directory and S_{new}, with the associated file set, as base selection. The conversion from the feature selections to the presence implications for a file is straightforward:

$$\mathit{toPC}(\text{File}, \text{Selection}) = \text{File} \rightarrow \bigvee_{S \in \text{Sels}} \left(\bigwedge_{f \in S} f \right) \tag{4}$$

Each selection is a conjunction of the set features that must be enabled in order satisfy the developer-specified KBUILD constraint to compile the file. Multiple selections occur when there are multiple rules that require the source file to be compiled. In case of multiple selections, all selections are disjuncted, because any of these disjunction leads to the inclusion of the file in the compilation process.

The resulting propositional formula can by simplified, for instance by removing selections that are a full subset of another selection.

4. EVALUATION

In the following, we evaluate our approach and compare it to the existing approaches. We start with a general description and compare the three respective implementations, which is followed by analyses regarding run time, robustness and coverage.

4.1 Implementation Overview

The GOLEM tool

We have implemented the algorithms from Section 3 into the GOLEM tool which is part of our VAMOS [30] toolchain [28, 29]. The implementation encompasses about $1{,}000$ lines of Python code.

The KBUILD specific probing primitives are implemented in two additional "front-end" makefiles (about 120 lines of MAKE code), so that not a single line in Linux had to be changed for the analysis. The tools are freely available on the project website.

KBUILDMINER

KBUILDMINER by Berger and She [3] has been presented on a poster at SPLC '10 [5] and is further detailed in a technical report [4]: A fuzzy parser transforms KBUILD makefiles into an abstract syntax tree (AST), which is then transformed into presence conditions. The implementation consists of about 1,400 lines of Scala code and 450 lines of Java code. The tool, as well as a result set for Linux v2.6.33.3, have been downloaded from [3]. Because this tool requires manual modification of existing makefiles (the technical report states that for Linux v2.6.28.6, 28 makefiles were adapted manually [4]), it is not easily possible to apply it to arbitrary versions of Linux.

The UNDERTAKER Extension by Nadi

Nadi and Holt [18] have implemented their KBUILD extractor independently from us. Similar to our GOLEM tool, this extractor calculates logical constraints that our UNDERTAKER tool [29] can use directly. Their implementation employs pattern matching in Linux makefiles to identify variability in KBUILD. It consists of about 750 lines of Java code. While not (yet) publicly available, the authors have kindly provided us with the version that has been used in [18].

4.2 Runtime

All parsing-based approaches are (persumably) much faster than the GOLEM implementation presented in this paper. For KBUILDMINER [4], no run-time data is available. The parser by Nadi and Holt [18] processes a architecture in under 30 seconds. The current GOLEM implemenation takes approximately 90 minutes per architecture. The obvious bottleneck is the run-time and the amount of probing steps, which have been described in Figure 4. For Linux v3.2 `arch-x86`, the *list* operation takes about a second (depending on the selected features and filesystem cache state) and was executed 7,073 times.

However, the *list* function does neither modify the analyzed source tree, nor exhibit other side effects. We therefore see a great potential in improving the performance by running several probing steps in parallel. For practical applications, the large runtime overhead has little big impact on the usability of the approach, because for many applications, such as the applications in Section 5, the variability extraction has to be done only once per version and architecture.

4.3 Robustness

As Linux is a moving target, variability identification and extraction approaches need to be both conceptually as well as implementation-wise robust. In order to evaluate the property of robustness for future versions of Linux, we test on a wide-ranged number of Linux versions have been retrieved from the git history. We choose five Linux releases with one year distance that cover 4 years of the Linux development (2008-2012). In order to keep the results for the various implementations comparable, we refrain from analyzing earlier versions than Linux v2.6.25, because the `arch-x86` architecture was introduced in v2.6.24 by merging the 32bit and 64bit variants, which were previously maintained separately. Table 1 summarizes the results of this analysis.

In general we found it challenging to apply the parsing-based approaches to Linux versions for which they have not been tailored to.

Table 1: Direct quantitative comparsion over Linux versions over the last 5 years. The Kernel versions are roughly equidistant over the time and include all version for which dataset are available for KBUILDMINER and the Nadi Parser.

All source files for v2.6.25 (w/o `#included` files)	6,826	(127)
Files hit by KBUILDMINER	*data not available*	
Files hit by GOLEM	6,274	(93.7%)
Files hit by Nadi parser	*tool crashes*	
All source files for v2.6.28.6 (w/o `#included` files)	7,665	(153)
Files hit by KBUILDMINER	7,243	(96.4%)
Files hit by GOLEM tool	7,032	(93.6%)
Files hit by Nadi parser	*tool crashes*	
All source files for v2.6.33.3 (w/o `#included` files)	9,827	(261)
Files hit by KBUILDMINER	9,090	(95%)
Files hit by GOLEM	9,079	(94.9%)
Files hit by Nadi parser	7,154	(74.8%)
All source files for v2.6.37 (w/o `#included` files)	10,958	(292)
Files hit by KBUILDMINER	*data not available*	
Files hit by GOLEM	10,145	(95.1%)
Files hit by Nadi parser	7,916	(74.2%)
All source files for v3.2 (w/o `#included` files)	11,862	(276)
Files hit by KBUILDMINER	*data not available*	
Files hit by GOLEM	11,050	(95.4%)
Files hit by Nadi parser	8,592	(74.2%)

For the fuzzy-parsing approach presented by Berger et al. [4], there are only data sets for Linux version v2.6.28.6 [4] and v2.6.33.3 [3] available. For all other versions we were unable to produce any results, because of the necessary (but undocumented) changes of the Linux makefiles. These modifications include the disabling parts of `arch/x86/Makefile` in a way that break a regular compilation. The technical report leaves it open what effects these changes have on the extracted logical constraints.

The parsing approach presented by Nadi and Holt [18] does not require any modifications to existing Makefiles. We were able to produce presence implications for two additional versions. Unfortunately, the tool crashes with an endless recursion and a stack overflow on Linux v2.6.28.6 and earlier, so that no logical constraints could be obtained.

The presented approach and implementation in this article produces presence implications on all selected versions without requiring any source code modification or version specific adaptations. Also, the extraction process for the 22 other architectures in Linux v3.2 did not require any further modification.

As shown in this section, both parsing-based approaches have difficulties to achieve a robust operation on a wide range of versions. Since the Linux build system is still in active development and difficulties like those described in Section 2.3 may appear with every new version, every new introduced MAKE idiom requires manual (and thus error-prone) additional engineering in order to keep up with the Linux development. In contrast to to that, our approach works in a robust manner with stable results for each version without any further adaptations.

4.4 Coverage

This subsection compares the results of the three KBUILD variability extractors quantitatively. We do this by analyzing for how many source files the respective approach produces a logical formula as metric for their coverage in the Linux v2.6.33.3 source tree for `arch-x86`. We choose this source tree because it is the most recent version of Linux for which results of all tools are available.

For that version, KBUILD handles a total of 9,827 source files. As pointed out by Nadi and Holt [17], 276 of these source files (2.8%) are referenced by `#include`-statements in other implementation

Table 2: Configuration Defect Analysis Results with Linux v3.2

Configuration Defects without file constraints	
Code defects	1835
Referential defects	415
Logical defects	83
Total:	Σ 2333
Configuration Defects with file constraints	
Code defects	1835
Referential defects	439
Logical defects	299
Total:	Σ 2573

Table 3: CC-Analysis Results with Linux v3.2, `arch-x86`

Analyzed files	10,383
Number of variation points (files + `#ifdef` blocks)	25,369
1. Comparison with 'allyesconfig'	
Number of compiler (tool) invocations	10,383
Rate of skipped invocations	18.5%
Configuration Coverage	67.2%
2. Expansion without file constraints	
Number of partial configurations	14,169
Rate of skipped tool invocations (partial configurations)	83.6%
Configuration Coverage	37.4%
3. Expansion with file constraints	
Number of partial configurations	12,388
Rate of skipped tool invocations (partial configurations)	18.2%
Configuration Coverage	78.6%

source files rather than Kbuild rules in Kbuild.

The Undertaker extension by Nadi and Holt [18] approach identifies presence implications for 7,154 out of all source files (74.8%). For 2,412 source files, no logical implication was found. A quick analysis of the data indicates that deficiencies in the mapping from build products to source files (cf. Section 2.2.1) are part of the problem for this relatively high number.

An analysis of the data provided for KbuildMiner [3] on the tool's website for `arch-x86` shows that the tool produces presence implications for 9,090 out of all source files (95%) on Linux v2.6.33.3, `arch-x86`. This data is consistent to the technical report [4], which states a coverage of 94 percent on Linux v2.6.28.6, `arch-x86`.

The current implementation of our Golem tool calculates presence implications for 9,079 out of the 9,566 source files on Linux v2.6.33.3 (94.9%) on `arch-x86`.

5. APPLICATIONS

As part of the VAMOS project [30], we aim at providing (Linux) developers tool support for managing and maintaining variability. This goal includes finding configuration defects [29] and making existing tools for static analysis variability-aware [28]. The remainder of this section demonstrates the improvements of considering the build system in these tools.

5.1 Configuration Defect Analysis

In earlier work [29], we have discussed and analyzed configuration-derived defects in the variability implementation on an earlier version of Linux v2.6.35. Such defects are inconsistencies in the variability implementation, such as `#ifdef` blocks that either cannot be selected under any configuration selection (a *dead* block), or there is provably no configuration that deselects a `CPP` block (an *undead* block). Our Undertaker tool creates for each `#ifdef` block a set of propositional formulas and checks their satisfiability with a SAT Checker. The first formula includes only the constraints that are found in the structure of the `CPP` statements [cf. 24]. If this formula is unsatisfiable, then the block is classified as a *code defect*. If it is satisfiable, logical constraints that derive from the Kconfig feature model are added as further conjunctions to the formula. If the enriched formula is unsatisfiable, the Undertaker tool classifies the `CPP` block as a *logical defect*. This formula may (still) contain configuration variables that are not declared in the configuration model for this architecture (e.g., `CONFIG_ARM` is not present on `arch-x86`, etc.). The third formula therefore adds contraints to set such absent variables to false, and checks for satisfiability again. If this enriched formula is now unsatisfiable, then the Undertaker tool classifies the `CPP` block as *referential defect*.

For this kind of analysis, our tools, which (now) include the extracted variability from Kbuild, do not only need to be robust regarding the Linux version, but also the analyzed architecture. A more detailed explanation of this experiment can be found in [29]. That work has yielded 1,776 configurability issues, for which 123 patches has been proposed (49 merged, 8 accepted, 15 acknowledged), which in total have fixed 364 of these issues (among them 20 confirmed new bugs).

Table 2 compares the impact of the inclusion of the extracted source file constraints by our Golem tool on the results produced by the approach as presented in [29]. In this experiment, source file constraints from all 23 architectures in Linux v3.2 have been used to enrich the variability models. Every defect is tested against each architecture individually (where applicable) and classified as such.

In this work, we define as **variation point** every `CPP` block and source file that Kconfig allows to include or exclude in the resulting build products. This simplifaction is valid, because the coarse-grained selection of source files by Make could also be implemented by `CPP` by introducing additional `#ifdef` blocks that contain the whole file.

We did not find any *dead* source files, that is, files that will never be compiled due to the constraints from Kbuild. We can therefore confirm that the contributions of Nadi and Holt [18] have fixed all these "dead files". Nevertheless, by considering Kbuild-derived constraints, the Undertaker tool detects 216 additional (+260.2%) logical defects in `#ifdef`-blocks. The number of configuration defects increases by 10.3 percent. This shows that the source-file constraints have an considerable improvement on the results.

5.2 Configuration Coverage

This subsection investigates the effects of the extracted source-file constraints on the configuration coverage (CC) [28]: We define CC as the fraction of selected variation points (`#ifdef`-blocks and source files as defined in Section 2.1) divided by all possible variation points. However, one has to be careful with calculating the "possible" variation points on a specific architecture, because architecture-specific drivers or `#ifdef` blocks that test for a specific other architecture must not be counted. In order to get a fair comparison, we use our Undertaker tool to detect such unselectable variation points in the 11,862 source files considered by Kbuild on `arch-x86` and exclude them from all results in this subsection.

We calculate a set of configurations which, when combined (i.e., compile each configuration individually), maximize the CC. This allows "traditional" tools for static analysis to uncover additional defects that are hidden in seldomly selected `#ifdef`-blocks. Table 3 summarizes the results. Since the analyzed source files only reference a subset of all available Kconfig features, the produced configuration are “incomplete” in the sense that they define only referenced features. Such a *partial configuration* sets only variation points from the extracted software variability [27] of a given source file. The remaining, unreferenced features need to be set in a way that they do not conflict in order to obtain a concrete product configuration, upon which traditional tools for static analysis can be employed. We use the KCONFIG tool to *expand* such partial to *full* configurations.

For comparison purposes, we first calculate the CC for the KCONFIG provided configuration preset `allyesconfig`. Interestingly, `allyesconfig` is way off from a "full" configuration, as 1,917 (18.5%) of all source files for `arch-x86` are **not** compiled. This, and the fact that every file with `#else` and `#elif` statements require more than one configuration to select all lines of code, account for the missing 32.8% CC.

In previous work [28], we have calculated partial configurations on all source files, and applied the KCONFIG infrastructure to expand each partial configuration to a full configuration. In this work, we consider both, KCONFIG-controlled `#ifdef` blocks (i.e., `#ifdef` blocks with a logical expression that contains at least one reference to a variable that starts with `CONFIG_`), as well as the inclusion of a source file into the compilation process, as a variation point. Therefore, the numbers of the calculated CC are hard to compare to those in our previous work [28].

Table 3 shows that the number of calculated configurations is not much higher than the number of analyzed source files (about 19.3% more configurations than source files). This number is surprisingly low because most files in Linux do not contain `#ifdef` blocks, but are controlled by at most a single MAKE variable (cf. Section 2.2). This means the majority of files in Linux require only a single configuration to achieve full CC.

For each partial configuration, we check if the respective expanded configuration would actually let KBUILD include the file in the build process. Because of uncovered source file constraints in the GOLEM implementation and incompleteness of our KCONFIG variability model, this is not always the case. We do not count variation points of a partial configuration that does not include its corresponding file, because this configuration does not practically cover any variation point.

When calculating the CC without considering source file constraints (the second experiment in Table 3), we notice a coverage of only 9,492 out of 25,369 (37.4%) possible variation points. The reason for this alarmingly low rate is that 11,844 out of 14,169 (83.6%) variation points have not been considered, because the calculated configuration did not compile the source file for which it has been calculated.

When calculating the CC with considering the file constraints (the third experiment in Table 3), we observe a CC of 19,938 out of 25,369 (78.6%) variation points. The reason for this improvement is that the rate of skipped configurations decreases dramatically to 16.4 percent. This number is still considerable. Since each skipped configuration provably contains skipped variation points, we expect that additional engineering (cf. Section 6.1) will considerably increase the CC even further. Additionally, a first analysis of the calculated partial configurations shows that the quality of the expansion process still leaves room for improvement: In many expanded configurations, we observe omitted and wrongly set features. Improving the expansion process would therefore improve the achieved CC as well.

Because of the skipped partial configurations and the deficiencies in the expansion process, the improvement of the calculated CC has to be seen as lower bound that can be greatly improved by more precise MAKE and KCONFIG models, and better expansion of partial configurations. We are currently working on improving these results.

6. DISCUSSION

As demonstrated by the two applications in the previous section, the implementation of our approach greatly assists variability-aware analyses. This subsection discusses the limitations and in what way the results can be transferred to other systems.

6.1 Benefits and Limitations of the Approach

Compared to parsing-based approaches for extracting variability from the build system [e.g., 4, 13, 18] our approach of build-system probing exhibits a number of unique characteristics. While existing parsing-based approaches suffer from technical implementation challenges that require manual (and error-prone) engineering for the many corner-cases, our approach handles complicated makefile constructions as presented in Section 2.3 and shell escapes (i.e., invocation of external tools in the build system) error-free. It is also much harder, as presented in Section 4.3, for a parsing based approach to keep pace with the Linux development, whereas our approach works predictably for a wide range of Linux versions and architectures.

However, we also make a number assumptions on the build system, which may impact the results of our approach:

1. We exploit the observation that the file presence implications in KBUILD correspond to the hierarchical organiztion of directories along subsystems. If a feature is a prequisite for enabling files in a subdirectories, then this constraint applies for each file in that directory.

2. We assume that in a subdirectory, each file is only dependant on single features and not by a conjunction of two or more features.

3. In KBUILD, a feature always selects additional sources files for compilation. In no case the selection of a feature causes a source file to be removed from compilation process. This is a rather uncommon feature for MAKE based systems but more commonly found in systems that employ delta-oriented programming (DOP) [22].

As shown in Section 4, the current implementation produces presence implications for 95.4% of all source files in Linux on `arch-x86`. An investigation of the remaining 4.6% source files reveales that the majority of files violate assumption #2. The violation of this assumption is best explained with an example:

```
my-obj-$(CONFIG_FB_MATROX_G)  += matroxfb_crtc2.o
obj-$(CONFIG_FB_MATROX)       += $(my-obj-y)
```

Here the the file `matroxfb_crtc2.o` is only built if both features `FB_MATROX_G` and `FB_MATROX` are enabled at the same time. The helper function *features_in_dir* fails to detect that those two features have a connection. Therefore both features are tested independently and the build product `matroxfb_crtc2.o` does not show up in the output of *list*.

In the future, we intend to cover these cases by employing some simple heuristics (e.g., with data from the KCONFIG model) in the helper function *features_in_dir* to probe for more than a single configuration variable at the same time without increasing the number of necessary probing steps excessively. We expect this to improve the resulting logical constraints both the quantitatively and qualitatively even further.

Depending on how the extracted build-system constraints are employed, the higher runtime, compared to other approaches, might be a limitation of the approach. However many applications require the KBUILD constraints to be calculated exactly once and reuse them in analyses that take much longer compared to the extraction process. This applies to both applications that have been presented in Section 5.

6.2 Generalizability

In contrast to the parsing-based approaches, which rely heavily the idiomatic style in which KBUILD makes use of the MAKE language, we avoid this dependency by treating the build system as a black box. Only two primitives, *list* and *features_in_dir*, have to be reimplemented for other build systems. This thin connection to the internal structures is the main reason for the robustness of the probing based approach with respect to the presented application on a wide range of Linux versions and architectures.

In order to show the portability of our approach, we have implemented the necessary adaptations for two further software projects: The build system of BUSYBOX [7], a toolbox of UNIX-tools for embedded systems, and the build system of FIASCO [12], a L4-like micro kernel. Both ports took less than 100 additional lines of code and were straight-forward to implement. We are convinced that the assumptions made on KBUILD in Section 6.1 also apply to other build systems.

6.3 Comparison of the Calculated Source File Constraints

For a qualitative evaluation of the extracted presence implications, we compare the output of our GOLEM tool to the results of Berger et al. [4] and Nadi and Holt [18]. For all the files that have a presence implication in our model, the presence implication from the other models is checked for semantic equivalence by using a SAT Checker.

$$\phi_{M_1}(f) \leftrightarrow \phi_{M_2}(f) \qquad f \in files(M_1) \cap files(M_2)$$

This equivalence check is done by instrumenting the SAT checker to prove that the bi-implication of the presence implications is a tautology and therefore have always the same implication. We use this check to compare the GOLEM model to the models of Nadi and Holt and Berger et al.

For the much smaller model of Nadi and Holt, 15 percent of the 7,082 common files have an equivalent presence implication and 81.9 percent have a presence implication that implies the GOLEM presence implication. We conclude that this model is mostly subsumed by the GOLEM model.

The comparison of the GOLEM model with the model from Berger et al. shows that out of 8,885 common files, 99.6 percent fulfill this bi-implication. This pratical equivalence shows that both tools are similarly mature.

7. RELATED WORK

The analysis of variability in Linux is a hot topic in the Software Engineering (SWE) and Software Product Line (SPL) community. Zengler and Küchlin [31] show an attempt to derive formal semantics of KCONFIG. She et al. [23] reverse-engineer the KCONFIG variability declaration in order to reconstruct a feature model. In [10] we have shown and quantified that the fine-grained variability implementation by CPP is dominated by a more coarse-grained management in KBUILD. We therefore think that KBUILD variability extractors, such as KBUILDMINER [3], the Nadi parser [18] or the GOLEM tool presented in this article, are a necessary complement for holistic variability analysis.

Berger et al. [6] investigate the configuration languages and tools KCONFIG and configuration description language (CDL). While the work shows that variability-management tools are employed successfully in open-source operating systems, it covers only the feature specification and modeling.

Adams et al. [2] demonstrate that analysis, visualization and in essence, re-engineering of the Linux build system is feasible. Their framework Makao [1] infers modularity in KBUILD by analyzing build traces. However, the amount of variation points that we identify in KBUILD with this article indicates that the full re-engineering of build-system variability remains an unsolved problem.

Kästner et al. [13] propose a technique coined "variability aware parsing", which essentially integrates the CPP variability into tools for variability aware type-checking. Mainly because of implementation challenges, TypeChef focuses on `arch-x86` and requires assistance in form of additional constraints by tools like KBUILD-MINER [3]. Even with this, the approach is restricted to CPP based variability—the build-system–derived variability remains out of scope.

Palix et al. [19] try to reproduce a ten year old analysis on Linux by Chou et al. [8] in order to investigate the evolutionary development of Linux across the last decade. As the old experiment misses to state the exact configuration that was used, the environment could only be approximated. Hereby, the paper indirectly discusses CC in the sense that the selected configuration can (and does) affect the results of static analysis tools considerably. We take this anecdote as call for further integration of configuration consistency checks and CC into static analysis tools.

Inside the software verification community, Post and Sinz [21] introduce a technique coined "configuration lifting", which translates the variability expressed in KCONFIG, KBUILD and CPP into C source code. The generated C files encode the variability of the original source, the makefiles, and the feature model, and is verified with the CBMC tool by Clarke, Kroening, and Lerda [9]. While "configuration lifting" has similar goals, it remains unclear if that approach scales to the size of Linux.

8. SUMMARY AND CONCLUSION

To cope with a broad range of application and hardware settings, system software has to be highly configurable. Linux v3.2, as a prominent example, offers 11,000 configurable features. The implementation of this huge amount of static variability is implemented by `#ifdef`-blocks in the source code, but especially by the Linux make system. From the maintenance point of view, this imposes big challenges, as the feature model and the configurability that is *actually* implemented in the code have to be kept in sync. This calls for tool support.

A major hurdle for acceptance by the Linux developers is that such tools have to work reliably on the latest development version of Linux. Robustness against evolutionary changes in Linux, which includes both C code and the build system, is a strong requirement. In this paper, we have presented such a robust approach for extracting variability from the Linux build system that extracts logical constraints for 95.4% of all source files in Linux v3.2 on the `x86` architecture. Unlike existing approaches, our approach does not try to analyze the makefiles, but exploits the build system itself to infer the effects of selected features on the set of compiled files. Instead of manual and error-prone engineering that tailors the variability extractor to a specific version or architecture of Linux, our approach requires only two basic and straightforward to implement primitives. This thin interface to the build system allows a straight-forward to implement adaptation of the approach to other software projects, which has been demonstrated for BUSYBOX [7] and FIASCO [12].

9. ACKNOWLEDGMENTS

We whish to thank our anonymous reviewers for their helpful suggestions. Special thanks got to Sarah Nadi and Thorsten Berger for providing access to their tools and data and their helpful comments on a draft of this paper.

This work was supported by the German Research Council (DFG) under grants no SCHR 603/7-2 and LO 1719/2-2.

References

[1] Bram Adams, Kris De Schutter, Herman Tromp, and Wolfgang De Meuter. "Design recovery and maintenance of build systems". In: *Proceedings of the 23st IEEE International Conference on Software Maintainance (ICSM'07)*. IEEE Computer Society Press, 2007. DOI: 10.1109/ICSM.2007.4362624.

[2] Bram Adams, Kris De Schutter, Herman Tromp, and Wolfgang De Meuter. "The Evolution of the Linux Build System". In: *Electronic Communications of the EASST* (2007).

[3] Thorsten Berger and Steven She. *Google Code Project: various variability extraction and analysis tools*. URL: http://code.google.com/p/variability/ (visited on 02/16/2012).

[4] Thorsten Berger, Steven She, Krzysztof Czarnecki, and Andrzej Wasowski. *Feature-to-Code Mapping in Two Large Product Lines*. Technical report. University of Leipzig (Germany), University of Waterloo (Canada), IT University of Copenhagen (Denmark), 2010.

[5] Thorsten Berger, Steven She, Rafael Lotufo, Krzysztof Czarnecki, and Andrzej Wasowski. "Feature-to-code mapping in two large product lines". In: *Proceedings of the 14th Software Product Line Conference (SPLC '10)*. Volume 6287. Lecture Notes in Computer Science. Poster session. Springer-Verlag, 2010.

[6] Thorsten Berger, Steven She, Rafael Lotufo, and Andrzej Wasowski und Krzysztof Czarnecki. "Variability Modeling in the Real: A Perspective from the Operating Systems Domain". In: *Proceedings of the 25th IEEE/ACM International Conference on Automated Software Engineering (ASE '10)*. ACM Press, 2010. DOI: 10.1145/1858996.1859010.

[7] *BusyBox Project Homepage*. URL: http://www.busybox.net/ (visited on 05/11/2012).

[8] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson Engler. "An empirical study of operating systems errors". In: *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*. ACM Press, 2001. DOI: 10.1145/502034.502042.

[9] Edmund Clarke, Daniel Kroening, and Flavio Lerda. "A Tool for Checking ANSI-C Programs". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Volume 2988. Lecture Notes in Computer Science. Springer-Verlag, 2004. DOI: 10.1007/978-3-540-24730-2_15.

[10] Christian Dietrich, Reinhard Tartler, Wolfgang Schröder-Preikschat, and Daniel Lohmann. "Understanding Linux Feature Distribution". In: *Proceedings of the 2nd AOSD Workshop on Modularity in Systems Software (AOSD-MISS '12)*. ACM Press, 2012. DOI: 10.1145/2162024.2162030.

[11] Kai Germaschewski and Sam Ravnborg. "Kernel configuration and building in Linux 2.5". In: *Proceedings of the Linux Symposium*. 2003.

[12] Michael Hohmuth. *The Fiasco kernel: System architecture*. Technical report. TU Dresden, 1998.

[13] Christian Kästner, Paolo G. Giarrusso, Tillmann Rendel, Sebastian Erdweg, Klaus Ostermann, and Thorsten Berger. "Variability-Aware Parsing in the Presence of Lexical Macros and Conditional Compilation". In: *Proceedings of the 26th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '11)*. ACM Press, 2011. DOI: 10.1145/2048066.2048128.

[14] Jörg Liebig, Sven Apel, Christian Lengauer, Christian Kästner, and Michael Schulze. "An Analysis of the Variability in Forty Preprocessor-Based Software Product Lines". In: *Proceedings of the 32nd International Conference on Software Engineering (ICSE '10)*. ACM Press, 2010. DOI: 10.1145/1806799.1806819.

[15] Marcilio Mendonca, Andrzej Wasowski, and Krzysztof Czarnecki. "SAT-based analysis of feature models is easy". In: *Proceedings of the 13th Software Product Line Conference (SPLC '09)*. Carnegie Mellon University, 2009.

[16] Andreas Metzger, Patrick Heymans, Klaus Pohl, Pierre-Yves Schobbens, and Germain Saval. "Disambiguating the Documentation of Variability in Software Product Lines: A Separation of Concerns, Formalization and Automated Analysis". In: *Proceedings of the 15th IEEE Conference on Requirements Engineering (RE '07)*. IEEE Computer Society, 2007. DOI: 10.1109/RE.2007.61.

[17] Sarah Nadi and Richard C. Holt. "Make it or Break it: Mining Anomalies from Linux Kbuild". In: *Proceedings of the 18th Working Conference on Reverse Engineering (WCRE '11)*. 2011. DOI: 10.1109/WCRE.2011.46.

[18] Sarah Nadi and Richard C. Holt. "Mining Kbuild to Detect Variability Anomalies in Linux". In: *Proceedings of the 16th European Conference on Software Maintenance and Reengineering (CSMR '12)*. To appear. IEEE Computer Society Press, 2012.

[19] Nicolas Palix, Gaël Thomas, Suman Saha, Christophe Calvès, Julia L. Lawall, and Gilles Muller. "Faults in Linux: Ten years later". In: *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '11)*. ACM Press, 2011. DOI: 10.1145/1950365.1950401.

[20] Klaus Pohl, Günter Böckle, and Frank J. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer-Verlag, 2005.

[21] Hendrik Post and Carsten Sinz. "Configuration Lifting: Verification meets Software Configuration". In: *Proceedings of the 23th IEEE/ACM International Conference on Automated Software Engineering (ASE '08)*. IEEE Computer Society, 2008. DOI: 10.1109/ASE.2008.45.

[22] Ina Schaefer, Lorenzo Bettini, Ferruccio Damiani, and Nico Tanzarella. "Delta-oriented programming of software product lines". In: *Proceedings of the 14th Software Product Line Conference (SPLC '10)*. Volume 6287. Lecture Notes in Computer Science. Springer-Verlag, 2010. DOI: 10.1007/978-3-642-15579-6_6.

[23] Steven She, Rafael Lotufo, Thorsten Berger, Andrzej Wasowski, and Krzysztof Czarnecki. "Reverse Engineering Feature Models". In: *Proceedings of the 33nd International Conference on Software Engineering (ICSE '11)*. ACM Press, 2011. DOI: 10.1145/1985793.1985856.

[24] Julio Sincero, Reinhard Tartler, Daniel Lohmann, and Wolfgang Schröder-Preikschat. "Efficient Extraction and Analysis of Preprocessor-Based Variability". In: *Proceedings of the 9th International Conference on Generative Programming and Component Engineering (GPCE '10)*. ACM Press, 2010. DOI: 10.1145/1868294.1868300.

[25] Diomidis Spinellis. "A Tale of Four Kernels". In: *Proceedings of the 30th International Conference on Software Engineering (ICSE '08)*. ACM Press, 2008. DOI: 10.1145/1368088.1368140.

[26] Richard M. Stallman, Roland McGrath, and Paul D. Smith. *GNU make manual. A Program for Directing Recompilation*. Free Software Foundation. GNU Press, 2010.

[27] Mikael Svahnberg, Jilles van Gurp, and Jan Bosch. "A Taxonomy of Variability Realization Techniques". In: *Software - Practice and Experience* 35.8 (2006). DOI: 10.1002/spe.v35:8.

[28] Reinhard Tartler, Daniel Lohmann, Christian Dietrich, Christoph Egger, and Julio Sincero. "Configuration Coverage in the Analysis of Large-Scale System Software". In: *Proceedings of the 6th Workshop on Programming Languages and Operating Systems (PLOS '11)*. ACM Press, 2011. DOI: 10.1145/2039239.2039242.

[29] Reinhard Tartler, Daniel Lohmann, Julio Sincero, and Wolfgang Schröder-Preikschat. "Feature Consistency in Compile-Time-Configurable System Software: Facing the Linux 10,000 Feature Problem". In: *Proceedings of the ACM SIGOPS/EuroSys European Conference on Computer Systems 2011 (EuroSys '11)*. ACM Press, 2011. DOI: 10.1145/1966445.1966451.

[30] *VAMOS - Variability Management in Operating Systems*. FAU Erlangen-Nuremberg, 2012. URL: http://www4.informatik.uni-erlangen.de/Research/VAMOS/.

[31] Christoph Zengler and Wolfgang Küchlin. "Encoding the Linux Kernel Configuration in Propositional Logic". In: *Proceedings of the 19th European Conference on Artificial Intelligence (ECAI 2010) Workshop on Configuration 2010*. 2010.

3 A portable Approach

Although Linux is a very big example of statically configurable software and target of many research projects, it is not the only interesting one. There are other systems with static variability, and many of them involve variability during the build process. Therefore it is useful to generalize and port the build-system probing to other systems.

Although the approach, presented in Section 2, is tailored to the build system used in Linux, the idea is abstract enough to apply it also to other variability encoding systems. For generalizing the probing it is necessary to get an abstract model of the structure, for which the approach is applicable.

3.1 Abstracting Build-System Variability

In this section the structure of Kbuild is revisited and a general abstraction, with Kbuild as a blue print of build systems in mind, is described.

In Section 1.1 the term *variation point* (VP) was introduced to describe single variability artifacts. A VP, which resides on the extensional side, is controlled by zero or more declared features, and effect one aspect of the produced software variant. In the case of Kbuild, a VP affects whether a file is included into the compiled kernel or not. So the build system can be seen as a set of VPs.

However, these VPs do not float in the build-system black box (see Figure 2) in an unstructured way. In Linux the VPs are encoded in a hierarchic system of Kbuild fragments. Those fragments reference each other and form a tree, or at least, an directed acyclic graph. The point of variabilitys (POVs) are the inner nodes and the VPs are the leaf nodes. The Kbuild fragments are organized hierarchically due the subdirectory structure and a fragment may include a subdirectory and its fragment under a certain feature selection. This implicit VP influences all VPs in the subdirectory fragment. In general, we will call an artifact where VPs are encoded and that might reference other artifacts under certain constraints a *point of variability*.

The constraints under which a POV gets referenced or a VP is activated encode the variability of the system. These constraints might be complex expressions or fairly simple tests whether a certain feature is selected or not. The presented approach works best for simple single-feature tests, but can be, as described later, tuned to work on more complex expressions.

If the structure is a directed acyclic graph (or even a tree) of POVs, at least one POV must be the top-level one and is not referenced by any other POV. This POV is the entry point for the build system. In Linux, it is the architecture-specific top-level Kbuild file. All parsing and probing-based approaches for handling build system variability start from there and go down according to the substructures. If there is more than one top-level POV, the process has to be started for all these top-level POVs.

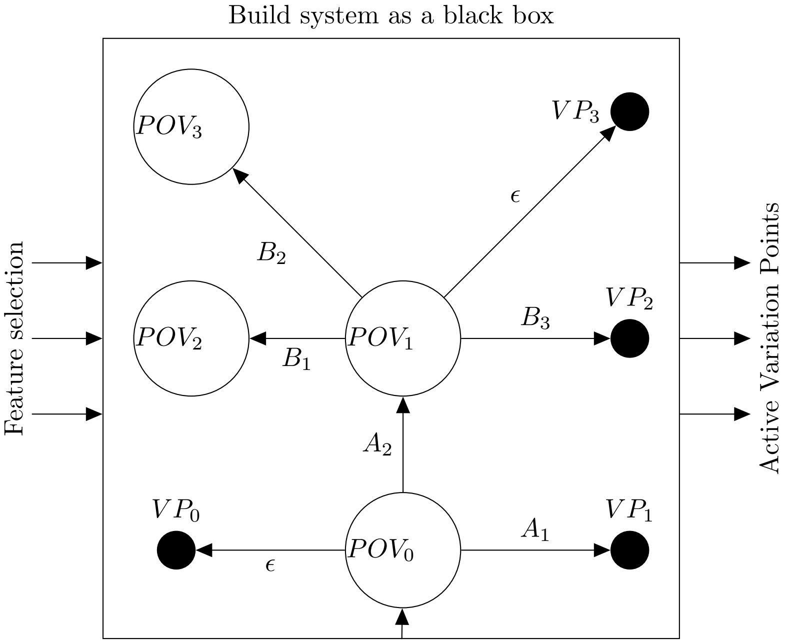


Figure 2: An abstract model of a hierarchic build system with points of variability (POV) and variation points (VP). Here POV_0 is the starting POV

3.2 Finding the Active Variation Points

For the described probing method on Linux it was necessary to obtain a list of compiled files and subdirectories from a given feature selection. For the generalized build system this LIST operation takes a feature selection and maps it to a set of activated VPs and POVs.

$$\textsc{list}\colon \text{Selection} \longmapsto (\text{VP-Set},\ \text{POV-Set})$$

Hereby it is not important how this mapping is calculated. In the best case, the build system is instrumented in a way that it outputs this information by itself. Analyzing build traces might be an option, but dependent on the size of the system, this will slow down the extraction process immensely.

As an example of the LIST operation a single mapping for the system modeled in Figure 2 is:

$$\textsc{list}(A_2 = \text{TRUE}) = (\{\text{VP}_0, \text{VP}_3\}, \{\text{POV}_0, \text{POV}_1\})$$

3.3 Collecting Expressions

On Linux, the inclusion of subdirectories and source-code files is mostly done by checks if a certain feature is selected. For the recursive exploration of the hierarchic structure, it is necessary to find all these checks for a given KBUILD fragment. This is done by finding all strings in the KBUILD fragment that start with `CONFIG_` and reference a declared feature.

For the more generalized method a similar operation is needed. It takes a POV and maps it to a set of all possible expressions that reference another POV or VP:

$$\textsc{expression\_in\_POV}\colon \text{POV} \longmapsto \text{Expression}$$

As an example of the EXPRESSION_IN_POV operation some mappings of the system modeled in Figure 2 are:

$$\begin{aligned} \textsc{expression\_in\_POV}(\text{POV}_0) &= \quad \{A_1, A_2\} \\ \textsc{expression\_in\_POV}(\text{POV}_1) &= \{B_1, B_2, B_3\} \end{aligned}$$

3.4 The Common Probing Algorithm

The common probing algorithm is very similar to the one for Linux, which is described in Section 2.3.4. It is implemented non recursively, but uses a working stack. While the general idea stays untouched, various extensions are easier to attach later.

Algorithm 1 The common build probing algorithm.

```
 1: POV_Stack := new Stack()                                ▷ The working Stack
 2: VP_PC := new Map(VP ↦ List of Selections)
 3: function COMMONBUILDPROBING
 4:     TESTPOV($S_\emptyset, \emptyset, \emptyset$)                          ▷ Test the empty selection
 5:     while POV_Stack.size() != 0 do
 6:         ($S_{base}$, $VP_{base}$, $POV_{base}$, pov) := POV_Stack.pop()
 7:         if not SHOULDVISITPOV($S_{base}$, pov) then
 8:             continue
 9:         end if
10:         for all $S_{new}$ in GENERATESELECTIONS($S_{base}$, pov) do
11:             TESTPOV($S_{new}$, $VP_{base}$, $POV_{base}$)
12:         end for
13:     end while
14:     PRINTALLSELECTIONS(VP_PC)
15: end function
```

The function COMMONBUILDPROBING, which is depicted in Algorithm 1, is the starting point of the probing algorithm. It uses two global data structures, which are shared among all presented functions. `POV_Stack` (line 1) is the working stack and contains 4-tuple, the work items:

$$\text{type}(\text{POV\_Stack}) = (\text{S}_{\text{base}}, \text{VP}_{\text{base}}, \text{POV}_{\text{base}}, \mathit{pov})$$

The work items consist of a base selection S_{base}, that activates the VPs in VP_{base} and the POVs in POV_{base}. Additionally, a specific POV *pov* from POV_{base} is the current working item. The variable `VP_PC` (line 2) will hold all the found selections for a specific VP. These selections are the result of the algorithm.

The selection that has no feature enabled is the first selection to be tested (line 4); all POVs that are unconditional activated are pushed onto the stack by TestPov for further investigation.

Now the working stack is processed until it is empty. As described, the stack consists of 4-tupel (line 6). It is checked whether the current working item is worth working on, otherwise it is skipped (line 7).

Starting from the current POV, all selections that might enable further VPs or POVs are generated and tested (line 10). This testing with TestPov might add additional work packages onto the stack, which are processed afterwards.

When the working stack is empty, the mapping from a VP to all activating selections is printed. This is the step that combines selection and generates a propositional (line 14). This step is of no further interest and very easy to implement.

Algorithm 2 The ShouldVisitPov function decides if a POV is visited, when it was reached with a given selection. A POV should only be visited (again), if the selection that activated the POV is not a superset of another selection, that enabled the same POV in the past.

```
 1: global POV_Selections := new Map(POV $\mapsto$ List of Selections)
 2: function ShouldVisitPov($\text{S}_{\text{new}}$, pov)
 3:     for all $\text{S}_{\text{visited}}$ in POV_Selections[pov] do
 4:         if $\text{S}_{\text{new}} \supseteq \text{S}_{\text{visited}}$ then
 5:             return FALSE
 6:         end if
 7:     end for
 8:     POV_Selections[pov].append($\text{S}_{\text{new}}$)
 9:     return TRUE
10: end function
```

The decision whether a POV is worth working on is checked in the ShouldVisitPov function (Algorithm 2). The basic condition for the function is that `pov` is activated by S_{new}. ShouldVisitPov holds a list, `POV_Selections`, of all selections that already activated `pov` in the past (line 1) and checks if S_{new} is a superset of any of these selections. This check is done to ensure the termination of the algorithm. For example would a POV not be revisited again under the selection $\{A, B, C\}$ when it was already visited with the selection $\{A, B\}$, since the former selection is a subset of the later.

Algorithm 3 After pov was activated by S_{base}, further selections are generated from S_{base} and EXPRESSION_IN_POV. The ALLFULLFILLINGSELECTIONS function appends all assignments of expr, that evaluate to true, to the base selection S_{base}

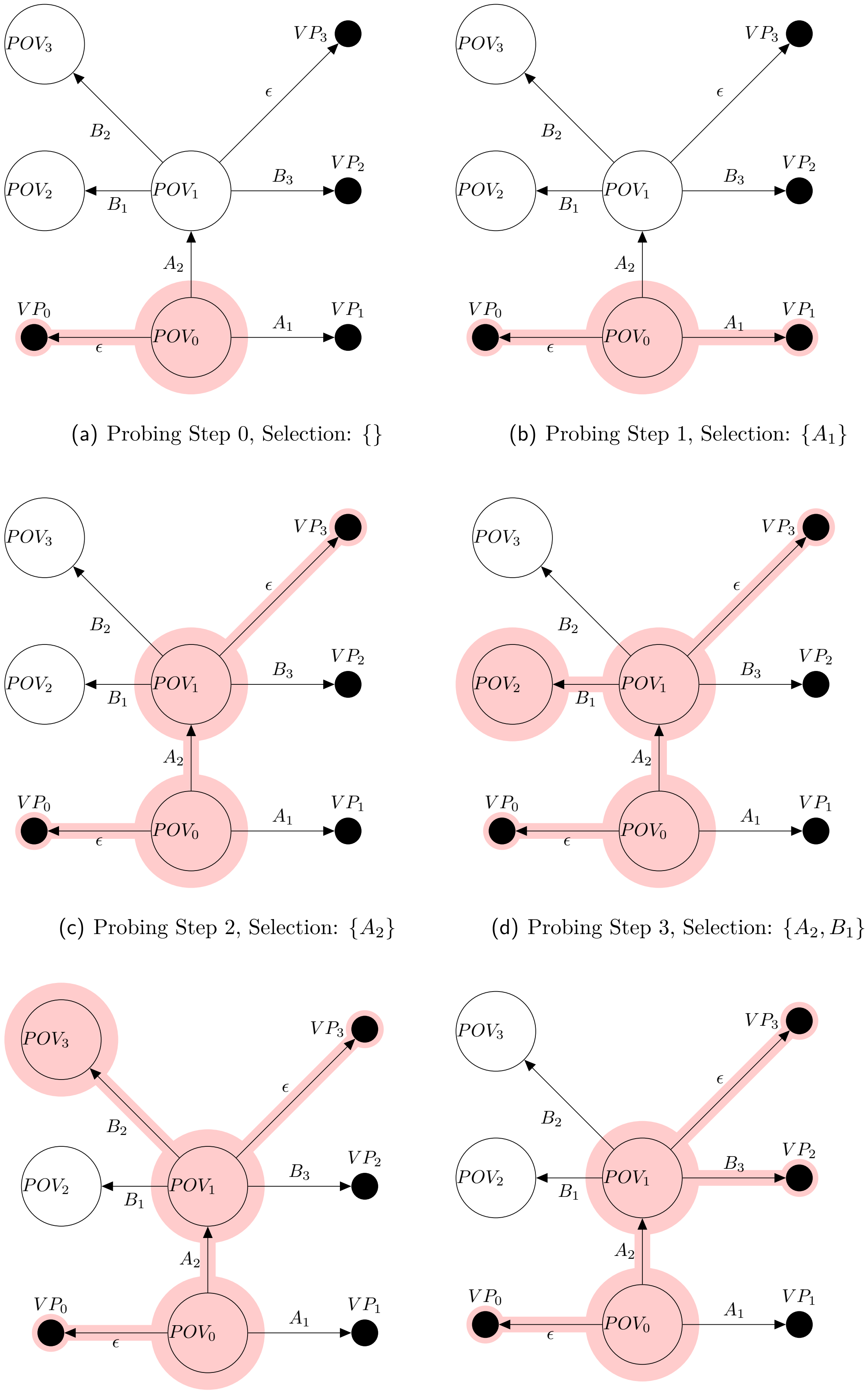
```
1: function GENERATESELECTIONS($S_{base}$, pov)
2:     $Selections_{new}$ := new List of Selections()
3:     for all expr in EXPRESSION_IN_POV (pov) do
4:         for all $S_{partial}$ in ALLFULLFILLINGSELECTIONS(expr, $S_{base}$) do
5:             $Selections_{new}$.append($S_{base} \cup S_{partial}$)
6:         end for
7:     end for
8:     return $Selections_{new}$
9: end function
```

An interesting part of the algorithm is the generation of new selections (Algorithm 3). The basic condition of the function is that `pov` was activated by S_{base}. It asks the build system for all expressions that belong to this `pov` (line 3). The ALLFULLFILLINGSELECTIONS function, which is not described in detail here, generates partial selections that fulfill one of these expressions, but do not conflict with the base selection S_{base}. The partial selections are combined with the base selection (line 5), and a list of to-be-tested selections is returned. For example, if the `pov` was activated by $\{A\}$ and has only one expression $B \vee C$, the following selections are generated: $\{\{A, B\}, \{A, C\}, \{A, B, C\}\}$.

Algorithm 4 The TESTPOV function calls the LIST build-system primitive, collects all the newly selected VPs and pushes the newly found POVs onto the working stack.

```
1: function TESTPOV($S_{new}$, $VP_{base}$, $POV_{base}$)
2:     ($VP_{new}$, $POV_{new}$) := LIST ($S_{new}$)
3:     for all vp in ($VP_{new} - VP_{base}$) do
4:         VP_PC[vp].append($S_{new}$)                ▷ VP found under new selection
5:     end for
6:                                                ▷ Push new POVs onto the Stack
7:     for all pov in ($POV_{new} - POV_{base}$) do
8:         POV_Stack.push(new tuple($S_{new}$, $VP_{new}$, $POV_{new}$, pov) )
9:     end for
10: end function
```

The testing of a newly generated selection is done in TESTPOV (Algorithm 4). The build system is asked for all items (both VPs and POVs) that are activated by S_{new} (line 2). For all VPs that are now activated, but were not activated before, S_{new} is added to the list of activating selections for this VP (line 4). All



(a) Probing Step 0, Selection: $\{\}$

(b) Probing Step 1, Selection: $\{A_1\}$

(c) Probing Step 2, Selection: $\{A_2\}$

(d) Probing Step 3, Selection: $\{A_2, B_1\}$

(e) Probing Step 4, Selection: $\{A_2, B_2\}$

(f) Probing Step 5, Selection: $\{A_2, B_3\}$

Figure 3: Steps of the Probing-Algorithm for the build system from Figure 2. The activated POVs and VPs are highlighted.

POVs that are newly activated generate a new 4-tupel work-item on the stack (line 8). These items are processed in the CommonBuildProbing function again.

3.5 Exemplary Operation

To visualize the working mechanism of the presented algorithm, the build system from Figure 2 will be probed and all steps are explained in detail. The activated POVs and VPs for each step are shown in Figure 3. We will assume that each expression checks for a single feature, and therefore results in only one additional selection and probing step. Each step show the situation after one call to TestPov is processed.

Probing Step 0: In the first step, the empty selection is probed. It activates VP_0 and POV_0, which are pushed onto the stack for further investigations.
Afterward:
$\mathrm{VP\_PC} = \{\ \mathrm{VP}_0 \mapsto \{\emptyset\}\ \}$
$\mathrm{POV\_Stack} = [\ (S_\emptyset,\ \{\mathrm{VP}_0\},\ \{\mathrm{POV}_0\},\ \mathrm{POV}_0)\ ]$

Probing Step 1: Now the POV_0 is taken from the stack, the two expressions ($\{A_1,\ A_2\}$) are found, which results in two probing steps (Step 1 and Step 2). First the expression A_1 is full filled by the selection $\{A_1\}$, which additionally activates VP_1. Afterwards, the stack seems empty, but one additional probing step is stil pending.
Afterward:
$\mathrm{VP\_PC} = \{\ \mathrm{VP}_0 \mapsto \{\emptyset\},\ \mathrm{VP}_1 \mapsto \{\{A_1\}\}\ \}$
$\mathrm{POV\_Stack} = [\ \ ]$

Probing Step 2: The probing step for POV_0 and the expression A_2 is done by the selection of $\{A_2\}$. This enables POV_1, which is put onto the working stack, and immediately VP_3, since it is reachable by a tautology from POV_1.
Afterward:
$\mathrm{VP\_PC} = \{\ \mathrm{VP}_0 \mapsto \{\emptyset\},\ \mathrm{VP}_1 \mapsto \{\{A_1\}\},\ \mathrm{VP}_3 \mapsto \{\{A_2\}\}\ \ \}$
$\mathrm{POV\_Stack} = [\ (\{A_2\},\ \{\mathrm{VP}_0,\ \mathrm{VP}_3\},\ \{\mathrm{POV}_0,\ \mathrm{POV}_1\},\ \mathrm{POV}_1)\ ]$

Probing Step 3: POV_1 has three expressions $\{B_1, B_2, B_3\}$, which will result in three additional probing steps. First, the expression B_1 will be fulfilled by probing the selection $\{A_2, B_1\}$. POV_2 is enabled and pushed onto the stack.
Afterward:
$\mathrm{VP\_PC} = \{\ \mathrm{VP}_0 \mapsto \{\emptyset\},\ \mathrm{VP}_1 \mapsto \{\{A_1\}\},\ \mathrm{VP}_3 \mapsto \{\{A_2\}\}\ \ \}$
$\mathrm{POV\_Stack} = [\ (\{A_2, B_1\},\ \{\mathrm{VP}_0,\ \mathrm{VP}_3\},\ \{\mathrm{POV}_0,\ \mathrm{POV}_1,\ \mathrm{POV}_2\},\ \mathrm{POV}_2)\ ]$

Probing Step 4: B_2 is the next expression. It is fulfilled by $\{A_2, B_2\}$. POV$_3$ is activated and pushed onto the stack.
Afterward:

VP_PC = { VP$_0$ $\mapsto$ $\{\emptyset\}$, VP$_1$ $\mapsto$ $\{\{A_1\}\}$, VP$_3$ $\mapsto$ $\{\{A_2\}\}$ }
POV_Stack = [($\{A_2, B_1\}$, {VP$_0$, VP$_3$}, {POV$_0$, POV$_1$, POV$_2$}, POV$_2$),
($\{A_2, B_2\}$, {VP$_0$, VP$_3$}, {POV$_0$, POV$_1$, POV$_3$}, POV$_3$)]

Probing Step 5: B_3 is the next expression. It is full filled by $\{A_2, B_3\}$. VP$_2$ is additionally activated by the selection, and therefore collected.
Afterward:

VP_PC = { VP$_0$ $\mapsto$ $\{\emptyset\}$, VP$_1$ $\mapsto$ $\{\{A_1\}\}$, VP$_3$ $\mapsto$ $\{\{A_2\}$,
VP$_2$ $\mapsto$ $\{\{A_2, B_3\}\}$ }
POV_Stack = [($\{A_2, B_1\}$, {VP$_0$, VP$_3$}, {POV$_0$, POV$_1$, POV$_2$}, POV$_2$),
($\{A_2, B_2\}$, {VP$_0$, VP$_3$}, {POV$_0$, POV$_1$, POV$_3$}, POV$_3$)]

The remaining items on the stack are popped, but since the to-be-examined POVs have no further expressions and no associated VPs, no additional calls to TESTPOV is done. Therefore the algorithm terminates, and prints out the collected presence implications:

$$\begin{aligned} \mathrm{VP}_0 &\rightarrow TRUE \\ \mathrm{VP}_1 &\rightarrow A_1 \\ \mathrm{VP}_2 &\rightarrow A_2 \wedge B_3 \\ \mathrm{VP}_3 &\rightarrow A_2 \end{aligned}$$

3.6 Summary

In this section the algorithm, that was developed for the Linux build system in Section 2, was rebuilt on top of an abstract build-system model. This build-system model can be applied, when the *variability points* are organized as leaf nodes in an directed acyclic graph with *points of variability* as inner nodes. On the edges *expressions* guard whether the edges is considered by a feature selection. The common build-system probing was described in detail and a concrete example was given.

4 Fine-Tuning the Probing Approach

Since the course of action described in Section 3 for extracting presence implications for each VP is rather abstract, it is easy to describe various improvements to it. These improvements make the approach faster and easier to adapt to various build systems.

4.1 Parallelization

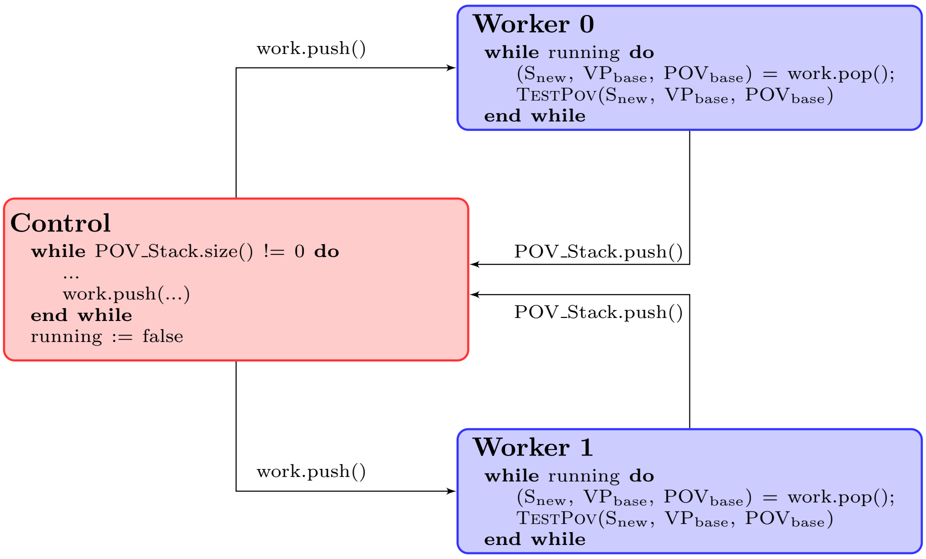

work.push()
Worker 0
while running do
(S_new, VP_base, POV_base) = work.pop();
TestPov(S_new, VP_base, POV_base)
end while
Control
while POV_Stack.size() != 0 do
...
work.push(...)
end while
running := false
POV_Stack.push()
POV_Stack.push()
work.push()
Worker 1
while running do
(S_new, VP_base, POV_base) = work.pop();
TestPov(S_new, VP_base, POV_base)
end while

Figure 4: Schema of the parallelization with work packages per POV probing step

The explorative nature of the approach results in a big number of calls to the build-system abstraction layer. These calls (LIST as well as EXPRESSION_IN_POV) may take a rather long time. For example, the probing of the Linux build system for one architecture (there are more than twenty architectures) takes 7000 calls to LIST, each of one second. Therefore a parallelization of these calls is a promising target.

For parallel operation without the need for multiple source trees, the LIST and the EXPRESSION_IN_POV primitives have to be side-effect free. This requirement forces the primitives to *not* alter or lock the source tree in any manner. Then two calls are invariant, regardless whether they are run sequential or in parallel.

The main function of the approach (Algorithm 1) is implemented sequential, but with later parallelization in mind. The basic principle, which is depicted in Figure 4, is to test the single feature selections with TestPov in concurrent worker threads, which find new POVs that are redirected to one control thread. This control thread generates new feature selections, which are processed by the workers. For the communication another (synchronized) queue has to be added to the global scope: `work`. It contains 3-tupel work items, which are exactly the three arguments to TestPov. Through the `work` queue, the to be tested selection is transported to the worker threads. The worker threads test the selection, fill newly found files into the `VP_PC` data structure, and report back newly discovered POVs via the `POV_Stack` to the controlling thread.

This parallelization speeds up the probing significantly. For the required 7000 probing steps a modern quad-core computer (3.2 Ghz, 4GB RAM) takes 164 minutes, when working sequential. When working in parallel with 6 worker threads the probing takes 92 minutes.

4.2 Non-Boolean Features

In many build systems not all features are simple boolean switches. Instead features can have more than two states, where only one (or none) state disables the feature and all other states enable the feature in a slightly different manner. An example of this are the `tristate` features in Linux: Each `tristate` feature has three states: `n`, `m` and `y`. The state `n` disables the feature at all, `m` compiles the feature as a loadable kernel modules (LKMs), which can be loaded at runtime, and `y` compiles and links the feature static into the kernel.

Another common paradigm is to use `string`-typed variables with a fixed domain of values to distinguish more than two states. For example, a feature `ARM_BOARD` with the domain {`"omap", "netwinder", "tegra"`}, where each value enables the feature, since the board type cannot be disabled.

The approach can be adopted to this situation. The AllFullFillingSelections function, which is used in Algorithm 3 in line 4, has to be adapted. It now examines the cross-product of the domains of the used features. All fulfilling selections that do not conflict with the base selection are possible new selection. A simple example of the functions operation is given, all non crossed selections are returned:

$$\begin{aligned} &\mathit{domain}(\text{BOARD}) && =\{''\mathit{omap}'', ''\mathit{tegra}''\} \\ &\mathit{domain}(\text{MMU}) && =\{\cancel{''n''}, ''y''\} \\ &\mathit{domain}(\text{DRV\_A}) && =\{\cancel{''n''}, ''m'', ''y''\} \end{aligned}$$

$$\textsc{AllFullFillingSelections}(\text{BOARD}\wedge\text{MMU}\wedge\text{DRV\_A}, \{\text{BOARD} = ''\mathit{omap}''\}) =$$

$$\begin{array}{ccc} \cancel{(''omap'', ''n'', ''n'')} & \cancel{(''omap'', ''n'', ''m'')} & \cancel{(''omap'', ''n'', ''y'')} \\ \cancel{(''omap'', ''y'', ''n'')} & \boldsymbol{(''omap'', ''y'', ''m'')} & \boldsymbol{(''omap'', ''y'', ''y'')} \\ \cancel{(''tegra'', ''n'', ''n'')} & \cancel{(''tegra'', ''n'', ''m'')} & \cancel{(''tegra'', ''n'', ''y'')} \\ \cancel{(''tegra'', ''y'', ''n'')} & \cancel{(''tegra'', ''y'', ''m'')} & \cancel{(''tegra'', ''y'', ''y'')} \end{array}$$

In this example, an expression of three different features is examined. The three tuples, which are returned here, are the values for (BOARD, MMU, DRV_A). The feature BOARD has two states, which both enable the feature. However in the base selection it is preset to "omap" and all possible combinations with BOARD="tegra" are *no* possible new selection. In all other crossed out possibilities either MMU is disabled or the tristate-feature DRV_A is set to n. So only two possible *new* selections are returned.

4.3 Implementing Special Cases

The presented build system model is abstract enough to describe most the important aspects correctly. But in some cases the model covers a build system almost, but some parts do not fit. In these cases it is necessary to specialize the unfitting parts in the used functions, instead of rewriting the whole probing logic. There are several places where build-system specific adaptions fit in without changing the modus operandi of the probing. The implementation of the algorithm does expose hooks at various points. Additionally to the build system primitives, the build-system driver can influence the algorithm with these hooks:

ShouldVisitPov: If there are parts of the source tree that should not be considered in the probing, this hook can prevent the recursive descent by filtering the unwanted POVs out. This might be useful when a subtree of the build system is too big and unnecessary for the current examination.

GenerateSelections: When there are selections that are invalid and are known to fail, they can be filtered beforehand to reduce the probing time. Also additional selections can be added if they are know to reach a certain subtree, which would not be reached by the automatic probing. By hooking into this function, it is also possible to add a certain feature, which is known to be needed in each expression by the build system. For example does the build system of Fiasco need a variable CONFIG_BSP_NAME to be set when arm is selected as architecture, otherwise it fails with an error.

PrintAllSelections: The output format can be altered by hooking into this function. For example, when a boolean model is wanted and it is known that a certain string-typed feature is caused by a certain boolean feature. In this case the output can be modified before emitting the model. For example, if it is know that BOARD="omap" is caused by OMAP=y, the sooner can be replaced by the later.

4.4 Summary

In this section a few adaptions of the probing algorithm have been presented. *Parallelization* improves the overall probing time by distribution the calls to the build system abstraction to different worker threads. Handling *non boolean* flags broadens the field for which the probing algorithm can be applied. The support for *special-casing* for different build-systems improves the portability. All of these fine-tunings were necessary for the porting to different build systems.

5 Case Studies

The abstract build system, which has been illustrated in Section 3.1, makes it easy to adapt the approach to various variability encoding systems, given they fit into the shown structure. Additionally to the port for Linux. two other ports to BusyBox [13] and Fiasco [14] were done and will be presented.

5.1 Kbuild: Linux

The Linux build system KBUILD is described great detail in various sources [15, 4, 16] and will not be further discussed here. Only the necessary adaptions to the base algorithm are presented here. In Linux, a source file (compilation unit) is a VP, and each KBUILD fragment a POV.

Necessary Adaptions: As described in Section 2, the LIST-primitive is implemented by calling a MAKE fragment that uses the build system scripts itself and prints out all files that are compiled with a given feature selection. The EXPRESSION_IN_POV-primitive is a regular expression, which returns all used `CONFIG_`-symbols in a given Makefile. This heuristic does only detect simple expressions (checks on the presence of a single feature). But these simple expressions are the majority of all expressions, since more than 90% of all VPs can be found with this heuristic.

Since about 40% of all features in Linux are `tristate` features [2], the non boolean adaption from Section 4.2 has to be applied to reach also the case where a source file is compiled to a LKM. Due the huge number of features in Linux the parallelization from Section 4.1 improves the operation runtime significantly.

Equivalence to the Berger [12] model: In order to give a hint on the quality of the extracted build-system model, the resulting presence implications were, as already mentioned in [10], compared to the presence implications extracted by Berger et al. [12]. This comparison is only a (strong!) hint on the plausibility of the model, since there is no proof that the Berger model is sound.

The equivalence comparison was done by checking whether the presence implication for each source file is semantically equivalent in both models. When a presence implications implies the one from the other model in all cases (the biimplication is a tautology), they might differ in syntax of the boolean formula, but not in its semantic.

From the 9146 files that have a presence implication in the Berger model (Linux v.2.6.33.3, `arch-x86`), [To do: NC wieder fixen]% have also an presence implication in the model, generated by probing. From these *common* files, [To do: NC wieder fixen]% have a semantically equivalent presence implication. In 23 cases, the presence implications have no implication in either direction (None of "Berger(file) ↔ Probing(file)", "Berger(file) ← Probing(file)", "Berger(file) → Probing(file)").

	presence	implications
Common Files	8940	([To do: NC wieder fixen]%)
Berger(file) ↔ Probing(file)	8903	([To do: NC wieder fixen]%)
Berger(file) → Probing(file)	8	([To do: NC wieder fixen]‰)
Berger(file) ← Probing(file)	6	([To do: NC wieder fixen]‰)
Berger(file) ≠ Probing(file)	23	([To do: NC wieder fixen]‰)

These numbers show that both methods produce nearly equivalent models. While the probing-based approach is stable in respect to the development cycle of Linux, as shown in [10], this is much harder to achieve with an parsing-based approach.

5.2 Kbuild: BusyBox

BusyBox [13] is a software product that provides many traditional UNIX tools for embedded systems. All tools are packed into a single executable, which behaves differently when it is called under a different name. When it is called under the name `echo` it behaves like the echo tool, when it is called as `sed`, it behaves like the stream editor[1], etc. BusyBox is a SPL, since the selection of tools that are put into this "big" executable can be configured at compile time. For many of the tools, features within a tool can also be configured at compile time.

The porting to BusyBox was rather easy, since BusyBox also uses KCONFIG as configurator and a modified version of KBUILD as build system. Therefore the build system primitives from Linux could be reused with little modification, and only 24 lines were added to support BusyBox.

	files	(of all `.c` files)
all `.c` files	710	100%
– helper scripts	26	([To do: NC wieder fixen]%)
– explicit unused	115	([To do: NC wieder fixen]%)
– implicit unused	8	([To do: NC wieder fixen]%)
– `#include`-ed	22	([To do: NC wieder fixen]%)
– example code	5	([To do: NC wieder fixen]%)
$\sum$ variation points	534	([To do: NC wieder fixen]%)
covered by the approach	534	([To do: NC wieder fixen]%)

The examination of the resulting build-system model gave the following quantitative results: BusyBox 1.20 consists of 710 source files. However not all files are

[1] `http://sed.sourceforge.net/`

handled by the build system. We removed all files, which are helper scripts for the configuration- and build process. Also a big ratio of files is unused and referenced nowhere. Hereby, explicit unused source files are obviously unused files[2], while implicit unused files are just not referenced from any point. Some source files are not handled by the build system, but included by a CPP `#include` statement. Additionally, some source files are only as example code in the source tree. When all files, that are obviously not handled by the build system are removed, 534 files (variation points) remain. For these remaining files, our approach gives presence implications for every file and achieves a variation point coverage of 100 per cent.

5.3 Fiasco: Hohmuth-Preprocessor

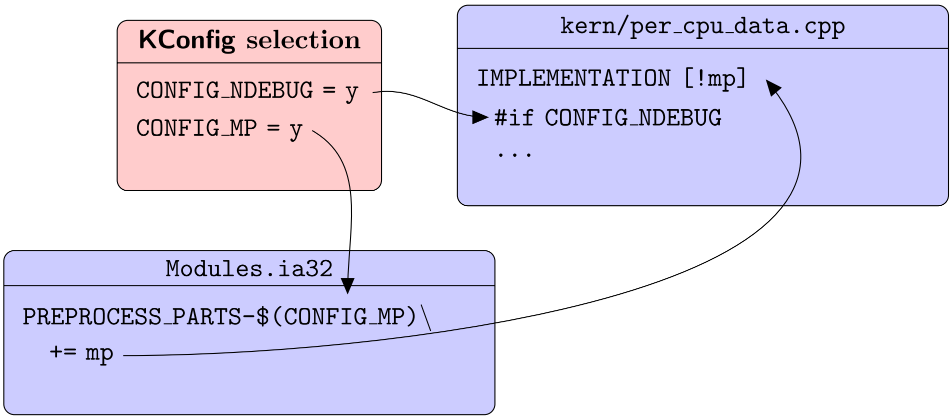


Figure 5: Influence of the KCONFIG selection in the Fiasco build process: The user selection affects the fine-grained variability in two manners. First the selections are directly exported as `CPP` symbols (`CONFIG_NDEBUG`). Secondly, the KCONFIG selection influences the Hohmuth flags that control the implementation and interface blocks (`CONFIG_MP` controls `mp`)

.

Another SPL project is the Fiasco [14] operating system, which is developed at the TU Dresden and is base of the TUDO:OS system. Fiasco is a L4-Microkernel that supports various hardware architectures (e.g., x86, ARM and PowerPC) but can also be run on top of an ordinary x86 Linux. Not only this multitude of supported hardware architectures is configurable at compile time, but also software aspects, such as the scheduling mechanism and virtualization features, can be chosen in the static configuration.

Fiasco also uses KCONFIG as configurator for managing the declaration of features, but on the extensional side the situation is quite different to Linux and BusyBox. On the coarse-grained granularity a similar approach to that used in Linux and BusyBox was taken. On the fine-grained level Fiasco does not only

[2] e.g. files named `unused_via_raid.c` or subdirectories named `old_e2fsprogs`

use the C preprocessor, but also the Hohmuth preprocessor [17]. In this case study, the focus was not on the coarse-grained variability, since it is very similar to the mechanisms in Linux and BusyBox. The focus is, how the fine-grained variability is controlled with the Hohmuth preprocessor.

The Hohmuth preprocessor is a C++-semantic aware preprocessor, which implements features like automatic header generation, automatic inlining of functions, and also the *selection of modules*. It is described as a "tool [which enables you] to write unit-style single-source-file modules in C++" [17]. The selection of modules is done by special Hohmuth flags similar to CPP flags. In Fiasco these flags are controlled directly by the feature selection.

How these flags are controlled by the selected features in Kconfig, is depicted in Figure 5. The flags are collected in `Modules` files that can reference the user selection. In the example the Hohmuth flag `mp` is dependent on the Kconfig feature `CONFIG_MP`. On the fine-grained level, `mp` controls an implementation block, that is only selected if `mp` is not enabled. The presence implication that has to be extracted in this case, for covering this indirection, is:

$$\texttt{mp} \rightarrow \texttt{CONFIG\_MP}$$

In other cases, the expressions are more complex and a Hohmuth flag depends on more than one Kconfig feature. Besides the Hohmuth preprocessor, Fiasco also uses the Kconfig selection directly with the `CPP`, like the `CONFIG_NDEBUG` feature, which is effective *within* the implementation block.

The `Modules` files have have make syntax and represent the POVs in this system. The Hohmuth flags themselves are the controlled VPs. More details on the Fiasco way of handling extensional variability is described in the bachelor thesis of Christian Schlumberger [18].

Christian Schlumberger extracted the presence implications for the Hohmuth flags by hand, but with the presented probing approach it could be automated. With only 126 lines of code the adaption is also rather small. Especially important were the hooks for special casing described in Section 4.3. For example, it is necessary to set the feature `BSP_NAME` to a correct value, when the feature `XARCH` is `"ppc"` or `"arm"`. Otherwise the build system refuses to work, since the names of some included files is calculated by the value of `BSP_NAME`.

The probing resulted in 98 presence implications for 98 Hohmuth flags. As a validation, all found presence implications were verified by hand and they indeed reflect the intended variability. It was also verified whether a Hohmuth flag was missing, and only one flag was not mentioned in the output. Manual investigation revealed that this flag was depended by a very complex expression, which was not detected by the expression_in_POV-primitive, since the checked feature name was calculated by the value of other features.

5.4 Summary

By porting the approach to three different build systems, I could show, that the abstract build-system model can be applied to various build systems that are used in real-world applications. The necessary adaptions were small in regard to complexity and lines of code. The simplicity of the porting, as well as the robustness, is reached by the small interface to the build system and the presented special-case hooks from Section 4.

6 Further Related Work

Additionally to the related work that was already discussed in [10], the analysis of build systems is a hot topic in the software engineering (SWE) and SPL community. Tamrawi et al. developed an infrastructure and tooling for MAKE system analysis due symbolic execution [19]. The analysis results in a symbolic execution trace, which is used for detecting anomalies in Makefiles and does support refactoring efforts of the build system.

The importance of the build system was shown in several empirical studies on build code maintenance. Hochstein and Jiao found that build-related code in scientific software is touched in 19-58% of all commits, while it does represent only 5-6% of the source code lines. While analyzing the KDE software versioning system Robles et al. [21] reported that many revisions contain only changes to build code. McIntosh et al. [22] analyzed ANT-based build systems and could show that the complexity of source code and build systems co-evolve. Kumfert and Epperly [23] investigated on the development overhead for maintaining the build system and revealed that developers claim, that 0%-35.71% of their time is spent maintaining the build system. All those studies emphasize the importance of build systems and their analysis.

The idea of exploitative probing of unknown acyclic graphs, presented in this work, is of course not new. In the computer network area the discovery of network topology is important for simulation and network management. Siamwalla et al. [24] used active probing heuristics with SNMP, ping requests and BGP information to discover the topology of intra- and internetworks. Huffaker et al. [25] describes the `skitter` tool, which was developed by CAIDA, the cooperative association for internet data analysis. It probes the internet routers by sending ICMP messages with different time-to-live (TTL) values and creates an IP-level graph from this data. These probing approaches are related to this work, since they also discover an unknown acyclic graph explorativly. But in contrast to the network probing the build-system probing acts on static data (at least for one version the source tree) and has to cope with much more complex expressions at edges, but has also more control over the probed system. For build systems the explicit path to a certain node can be chosen (the presence implication), while for networks the path cannot be influenced for a certain destination and changes over time. Of course also the goal differs. For build systems the presence implications are interesting and for networks the explicit graph is needed to perform further analyzes.

7 Conclusion

The role build systems have in the variability implementation of a software product is immense and much bigger than expected in the past. Therefore has a variability-aware analysis of a software product take the variability information in the build system into account. Extracting this information is challenging, especially when different build systems and more than one version of these should be analyzed.

In this work an *portable* and *robust* approach for extracting this variability from build systems has been described. It relies on an abstract model, how build systems encode variability. With this abstract model in mind the approach has been ported with little effort to the build systems of Linux, BusyBox and Fiasco. On Linux the approach could extract logical constraints for 95.4% of all source files in Linux v3.2 on the x86 architecture. While other approaches failed to extract this information for various versions of Linux, this probing-based approach worked with a constant high rate of source file coverage. For BusyBox it even achieved a source file coverage of 100 per cent and for Fiasco all extracted constraints could be verified for correctness.

However, the soundness of presence implications that are extracted by this approach was not proven. Also has the approach a higher run time than regular parsing techniques and is limited to build systems that fit into the presented model. It failed to achieve a full file coverage on the complex build system of Linux. Improving this coverage without blowing up the run time remains future work. The examination, whether the approach is applicable to totally different variability encoding system, like the intentional feature model, might also be a piece of future work.

In summary, I could show, that the presented approach is a feasible way for extracting variability from different build systems. The approach does not require much manual error-prone engineering to keep up with development cycles and is efficient enough for a day-to-day use.

References

[1] Software Engineering Institute. Software Product Lines. URL `http://www.sei.cmu.edu/productlines/`. `http://www.sei.cmu.edu/productlines/`, visited 18/05/2012.

[2] Christian Dietrich, Reinhard Tartler, Wolfgang Schröder-Preikschat, and Daniel Lohmann. Understanding Linux feature distribution. In Christoph Borchert, Michael Haupt, and Daniel Lohmann, editors, *Proceedings of the 2nd AOSD Workshop on Modularity in Systems Software (AOSD-MISS '12)*, New York, NY, USA, 2012. ACM Press. ISBN 978-1-4503-1217-2. doi: 10.1145/2162024.2162030.

[3] Reinhard Tartler, Daniel Lohmann, Julio Sincero, and Wolfgang Schröder-Preikschat. Feature consistency in compile-time-configurable system software: Facing the Linux 10,000 feature problem. In Christoph M. Kirsch and Gernot Heiser, editors, *Proceedings of the ACM SIGOPS/EuroSys European Conference on Computer Systems 2011 (EuroSys '11)*, pages 47–60, New York, NY, USA, April 2011. ACM Press. ISBN 978-1-4503-0634-8. doi: 10.1145/1966445.1966451.

[4] Sarah Nadi and Richard C. Holt. Make it or break it: Mining anomalies from linux kbuild. In *Proceedings of the 18th Working Conference on Reverse Engineering (WCRE '11)*, pages 315–324, 2011. doi: 10.1109/WCRE.2011.46.

[5] Reinhard Tartler, Daniel Lohmann, Christian Dietrich, Christoph Egger, and Julio Sincero. Configuration coverage in the analysis of large-scale system software. In Eric Eide, Gilles Muller, Olaf Spinczyk, and Wolfgang Schröder-Preikschat, editors, *Proceedings of the 6th Workshop on Programming Languages and Operating Systems (PLOS '11)*, pages 2:1–2:5, New York, NY, USA, 2011. ACM Press. ISBN 978-1-4503-0979-0. doi: 10.1145/2039239.2039242.

[6] Christian Kästner, Paolo G. Giarrusso, Tillmann Rendel, Sebastian Erdweg, Klaus Ostermann, and Thorsten Berger. Variability-aware parsing in the presence of lexical macros and conditional compilation. In *Proceedings of the 26th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '11)*, New York, NY, USA, October 2011. ACM Press. doi: 10.1145/2048066.2048128.

[7] Andy Kenner, Christian Kästner, Steffen Haase, and Thomas Leich. TypeChef: Toward type checking #ifdef variability in C. In *Proceedings of the 2nd Workshop on Feature-Oriented Software Development (FOSD '10)*, pages 25–32, New York, NY, USA, October 2010. ACM Press. ISBN 978-1-4503-0208-1. URL `http://www.informatik.uni-marburg.de/~kaestner/FOSD10-typechef.pdf`.

[8] Steven She and Thorsten Berger. Formal semantics of the Kconfig language, 2010.

[9] makesite. GNU make – GNU project – Free Software Foundation (FSF). URL `http://www.gnu.org/software/make`. `http://www.gnu.org/software/make`, visited 2011-11-12.

[10] Christian Dietrich, Reinhard Tartler, Wolfgang Schröder-Preikschat, and Daniel Lohmann. A robus approach for variability extraction from the linux build system. In *Proceedings of the 12th Software Product Line Conference (SPLC '12)*, September 2012. (To appear).

[11] Sarah Nadi and Richard C. Holt. Mining Kbuild to detect variability anomalies in Linux. In Tom Mens, Yiannis Kanellopoulos, and Andreas Winter, editors, *Proceedings of the 16th European Conference on Software Maintenance and Reengineering (CSMR '12)*, Washington, DC, USA, 2012. IEEE Computer Society Press. To appear.

[12] Thorsten Berger, Steven She, Krzysztof Czarnecki, and Andrzej Wasowski. Feature-to-code mapping in two large product lines. Technical report, University of Leipzig (Germany), University of Waterloo (Canada), IT University of Copenhagen (Denmark), 2010.

[13] BusyBox project homepage. URL `http://www.busybox.net/`.

[14] Fiaso project homepage. URL `http://os.inf.tu-dresden.de/fiasco/`.

[15] Kai Germaschewski and Sam Ravnborg. Kernel configuration and building in linux 2.5. In *Proceedings of the Linux Symposium*, pages 185–200, 2003.

[16] Michael Elizabeth Chastain, Kai Germaschewski, Sam Ravnborg, and Jan Engelhardt. Linux Kernel Makefiles, 2011. Available at `Documentation/kbuild/makefiles.txt` in the Linux source tree.

[17] Michael Hohmuth. Preprocess - A preprocessor for C and C++ modules, 2005. Available at `http://os.inf.tu-dresden.de/~hohmuth/prj/preprocess/`.

[18] Christian Schlumberger. Variabilitätsgewahre Analysen im FIASKO/L4-Mikrokernel, 2012.

[19] Ahmed Tamrawi, Hoan Anh Nguyen, Hung Viet Nguyen, and Tien N. Nguyen. Build code analysis with symbolic evaluation. In *Proceedings of the 34nd International Conference on Software Engineering (ICSE '12)*, 2012.

[20] Lorin Hochstein and Yang Jiao. The cost of the build tax in scientific software. In *International Symposium on Empirical Software Engineering and Measurement 2011 (ESEM '11)*, September 2011.

[21] Gregorio Robles, Jesus M. Gonzalez-Barahona, and Juan Julian Merelo. Beyond source code: The importance of other artifacts in software development (a case study). *Journal of Systems and Software*, (9):1233 – 1248, 2006.

[22] S. McIntosh, B. Adams, and A.E. Hassan. The evolution of ant build systems. In *Mining Software Repositories (MSR), 2010 7th IEEE Working Conference on*, pages 42 –51, may 2010.

[23] G. K. Kumfert and T. G. W. Epperly. Software in the DOE: The Hidden Overhead of "The Build". Technical report, Lawrence Livermore National Laboratory, 2002.

[24] R. Siamwalla, R. Sharma, and S. Keshav. Discovering internet topology. Technical report, Cornell University, Ithaca, 1999.

[25] B. Huffaker, D. Plummer, D. Moore, and k. claffy. Topology discovery by active probing. In *Symposium on Applications and the Internet (SAINT)*, pages 90–96, Nara, Japan, Jan 2002. SAINT.

List of Figures