Feature-to-Code Mapping in Two Large Product Lines

Thorsten Berger[1], Steven She[2], Krzysztof Czarnecki[2] and Andrzej Wąsowski[3]

[1] University of Leipzig, Germany berger@informatik.uni-leipzig.de
[2] University of Waterloo, Canada {shshe, kczarnec}@gsd.uwaterloo.ca
[3] IT University of Copenhagen, Denmark wasowski@itu.dk

Abstract. Large product lines have complex build systems, which obscure mapping of features to code. We extract this mapping out of the build systems of two operating systems kernels, Linux and FreeBSD. The mapping is presented as a set of *presence conditions* relating code fragments to features. We characterize them and make available for use as a benchmark for analysis tools for variability modeling. We hope that this work will enable the study of real-world variability models and the creation of new, scalable product-lines design and analysis tools.

1 Introduction

Large software product lines have complex *build systems* that enable compiling the source code into different products that make up the product line [1]. Unfortunately, the dependencies among the available build options, which we refer to as *features* and their mapping to the source code they control are implicit in complex imperative build-related logic. As a result, reasoning about dependencies is difficult; this not only makes maintenance of the variability harder, but also hinders development of support tools such as feature-oriented traceability support, debuggers for variability models, variability-aware code analyzers, or test schedulers for the product line. Thus, we advocate use of explicit *variability models*, consisting of a *feature model* specifying the available features and their dependencies and a *mapping* between product specifications conforming to the feature model and the implementation assets.

Previously [6] we extracted a feature model of the Linux Kernel. Now, we extract the feature-to-code mapping from the build systems of two prominent operating systems, Linux and FreeBSD. We represent the mapping as *presence conditions* [2, 3]. A presence condition (PC) is a Boolean expression on an implementation artifact written in terms of features. If a PC evaluates to *true* for a configuration, then the corresponding artifact is included in the product.

We show that extraction of mappings from build systems is feasible. We extracted 10,155 PCs, each controlling the inclusion of a source file in a build. The PCs reference the total of 4,774 features and affect about 8M lines of code. We publish the PCs for Linux and FreeBSD as they constitute a highly realistic benchmark for researchers designing tools such as dead code analysis, feature impact analysis, reasoners and configurators. Our work, combined with the feature model of Linux [6] and the work of Tartler et al. on the code-statement-level

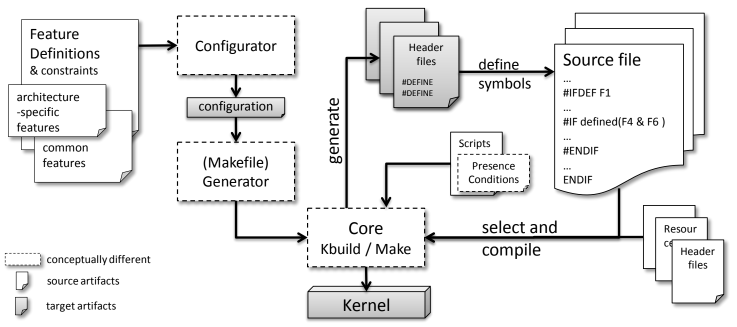


Fig. 1. Conceptual view on the build system of Linux and FreeBSD

variability [8], will eventually lead to extracting a complete variability model encompassing the feature model and the mapping from features to code. We hope that this work will enable the study of real-world variability models and the creation of new scalable design and analysis tools for such product lines.

Related Work. Previous studies of Linux's build system focus on studying only selected configurations. Adams et al. [1] argue that more insight could be gained by analyzing the entire variability in the build system. The work by Tartler et al. mentioned above, and work of Liebig et al. [4] examine the variability induced through preprocessor directives in Linux and FreeBSD. While both works deliver insights into solution space variability, we address the variability induced by the very mapping from problem to solution spaces.

2 Variability and Build Systems in Linux and FreeBSD

Both Linux and FreeBSD have a wealth of features: 5,323 in Linux (v. 2.6.28.6, x86 architecture) and 1,203 in FreeBSD (v. 8.0.0, all architectures). Here features are user-selectable increments of functionality, such as drivers, protocols, file systems, and multimedia devices. Linux provides its own feature modeling language, *Kconfig*, and a simple graphical configurator. FreeBSD has no configurator; instead, variants are specified by instantiating textual templates. Both systems support boolean and data valued features. Many Linux features assume values from a three element domain (`y`, `m`, `n`) encoding the binding mode (static linking, dynamic linking, and absence, respectively). We refer to [6, 5] for details.

Fig. 1 summarizes the architecture of the build systems of Linux and FreeBSD. The two build systems specify and instantiate the mapping between features and source differently. Linux uses KBuild—a project specific build system comprising a hierarchy of Makefiles following a specific convention. The logic selecting source files for a particular configuration is spread over more than 600 files in the entire source tree. FreeBSD relies on the standard Make tool and a generator creating a monolithic Makefile after evaluating the PCs in a given configuration. The mapping is specified explicitly in 9 central files.

Both systems generate header files that expose current configuration data to the preprocessor. We focus solely on the dependencies expressed in the build system. The dependencies in source files have already been discussed in [4, 7, 8].

Linux Kernel. The top-level KBuild Makefile declares lists collecting files for compilation in different modes: to be linked statically (`obj-y`), to be linked dynamically as modules (`obj-m`), or to be included in a library (`lib-y`). It then descends into the source tree and conditionally invokes other makefiles, which add files to the lists. In the simplest case, files are added unconditionally. In the example below two files are added to the `obj-y` list together with directory `partitions/`, which means that the Makefile located there should be included in further traversal. Note that names of object files are used, not source files. The source files are linked to object files by implicit compilation rules of Make:

```
obj-y += open.o jffs2.o partitions/
jffs2-y := compr.o dir.o file.o ioctl.o nodelist.o malloc.o          (1)
```

The second line creates a list indicating files that should be used to build `jffs2.o`. A compilation rule specified elsewhere declares a dependency between object files and lists of this kind. In the example, the PCs for all files are simply *true*. However the complete PC of a file may be different, due to inheritance of conditions from enclosing makefiles.

Files are added conditionally either by using control-flow statements of make or by constructing the name of a list conditionally. We illustrate the latter first:

```
obj-$(JFFS2_FS) += jffs2.o
jffs2-$(JFFS2_FS_WRITEBUFFER) += wbuf.o                              (2)
```

Here `$(JFFS2_FS)` denotes a value of feature `JFFS2_FS` in the configuration, in this case a string `y`, `m` or `n`, which is concatenated to create a list name. Note that `JFFS2_FS_WRITEBUFFER` can only be `y` or `n`. This example results in the following PC for `wbuf.c`:

$$\mathsf{JFFS2\_FS\_WRITEBUFFER} = \mathsf{y} \wedge (\mathsf{JFFS2\_FS} = \mathsf{y} \vee \mathsf{JFFS2\_FS} = \mathsf{m}) \qquad (3)$$

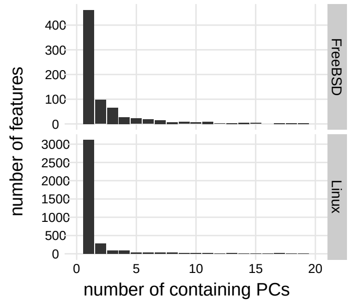
Below we show a conditional make command that induces the following PC for `xfs_qm_stats.c`: $(\mathsf{XFS\_FS} = \mathsf{y} \vee \mathsf{XFS\_FS} = \mathsf{m}) \wedge \mathsf{PROC\_FS} = \mathsf{y} \wedge \mathsf{XFS\_QUOTA} = \mathsf{y}$.

```
obj-$(XFS_FS) += xfs.o
ifeq ($(XFS_QUOTA),y)
   xfs-$(PROC_FS) += quota/xfs_qm_stats.o                            (4)
endif
```

Some dependencies are expressed in complex manners. The next example includes `dccp_ipv6` as a module if either the feature `IPv6` or `DCCP` equals `m`:

```
obj-$(subst y,$(CONFIG_IP_DCCP),$(CONFIG_IPV6)) += dccp_ipv6.o ,     (5)
```

where `subst` is a substring substitution function. We created the PCs for these complex manually. We obtain the following condition for this example: $\mathsf{IPV6} = \mathsf{m} \vee (\mathsf{IPV6}{=}\mathsf{y} \wedge (\mathsf{IP\_DCCP} = \mathsf{m} \vee \mathsf{IP\_DCCP} = \mathsf{y}))$

(a)


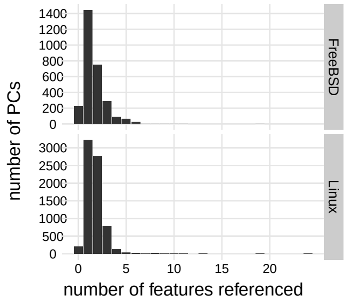
(b)


Fig. 2. (a) The no. of PCs that a feature appears in pruned to 20 PCs; features in Linux and FreeBSD appear in up to 424 and 291 PCs respectively and (b) size of PCs

FreeBSD Kernel. Here, PCs are organized in few files. Each line contains a source file name and a statement whether the file is mandatory or optional, with a condition in conjunctive normal form. Build options and dependencies can be appended if needed. An example of a simple specification of a variable file is:

```
crypto/des/des_ecb.c   optional crypto | ipsec | netsmb                (6)
```

3 Extraction and Results

Our extractor for Linux has a fuzzy parser recognizing all of the documented variability specification patterns, but also some undocumented ones, which we discovered in the KBuild Makefiles. For this work, we only analyzed the x86 architecture, starting with its main makefile and descending into the referenced ones. The resulting Abstract Syntax Tree (AST)[4] contains nodes representing Makefiles, conditional statements, lists of compound objects, variable references, and source files as leafs, some annotated with local PCs. Computing the PC of a source file involved finding all paths from the file to the root while taking variable resolution rules into account and then conjoining all expressions in a path and making a disjunction over the path conditions.

Extracting PCs from the FreeBSD Kernel was much simpler. We created parsers to analyze the build specification files and extracted PCs from the main kernel build system. Interestingly, FreeBSD does not have tristate features for addressing modules, but uses an independent module build system instead. The PCs in the module build system appear to be similar to the ones in the main system; however, analyzing the module build system is future work.

We extracted 7,243 PCs out of 596 makefiles in Linux' x86 branch, whereas we had to manually adapt 28 makefiles for our parser. Concerning FreeBSD, we extracted 2,912 PCs from the entire codebase. Both datasets are available on our website[5] and cover 94% of Linux' and 80% of FreeBSD's source files. We

[4] http://www.informatik.uni-leipzig.de/~berger/pcs/linux_build_ast.xml
[5] http://www.informatik.uni-leipzig.de/~berger/pcs/lin_fb_pcs.txt

found the following reasons for uncovered files. First, many C files were only included via other C files; second, additional obscure build logic was used; third, files belong to additional non-kernel tools; and fourth, some files were actually unreachable.

Fig, 2 visualizes basic properties of the collected PCs. We find that the majority of features (87% in Linux, 78% in FreeBSD) appear in less than four PCs (Fig. 2a). However diversity is wide, with some features like SND appearing in 424 PCs in Linux and i386 appearing in 291 PCs in FreeBSD.

Next in Fig. 2b, we see that the sizes of the PCs in the two systems are very similar, with Linux and FreeBSD having PCs referencing up to 24 and 19 unique features respectively. In Linux, the largest PC belongs to isdn/hisax/arcofi.c, which provides common functions across all HiSax drivers, a set of drivers for various Siemens ISDN cards. Features in the driver set required the compilation of this file. Similarly in FreeBSD, dev/usb/serial/usb_serial.c was a common source file required by 19 USB features. Furthermore, we found that there were only a small number files that were unconditionally included in both systems, reflected as files with zero features referenced. Therefore, we see that the majority of files indeed have variability.

4 Conclusion

We have demonstrated the feasibility of extracting feature-to-code mappings from the build systems of two operating systems. Although the extractors need to be custom built, they can enable a wide range of useful tools, amortizing the building effort. We also hope that the extracted mappings will provide designers of variability tools and languages with a realistic benchmark.

References

1. Adams, B., Schutter, K.D., Tromp, H., Meuter, W.D.: The evolution of the linux build system. In: 3rd International ERCIM Symposium on Software Evolution (2007)
2. Czarnecki, K., Antkiewicz, M.: Mapping features to models: A template approach based on superimposed variants. In: GPCE'05. pp. 422–437 (2005)
3. Heidenreich, F., Wende, C.: Bridging the gap between features and models. In: 2nd Workshop on Aspect-Oriented Product Line Engineering (AOPLE'07) (2007)
4. Liebig, J., Apel, S., Lengauer, C., Kästner, C., Schulze, M.: An analysis of the variability in 40 preprocessor-based software product lines. In: ICSE (2010)
5. She, S., Berger, T.: Formal semantics of the Kconfig language, technical Note. Available at eng.uwaterloo.ca/~shshe/kconfig_semantics.pdf
6. She, S., Lotufo, R., Berger, T., Wąsowski, A., Czarnecki, K.: Variability model of the linux kernel. In: VaMoS'10. pp. 45–51 (2010)
7. Sincero, J., Tartler, R., Lohmann, D.: An algorithm for quantifying the program variability induced by conditional compilation. Tech. rep., Univ. of Erlangen (2010)
8. Tartler, R., Sincero, J., Schröder-Preikschat, W., Lohmann, D.: Dead or alive: finding zombie features in the Linux kernel. In: FOSD (2009)