GSDLAB TECHNICAL REPORT

Variability Modeling in the Systems Software Domain

Thorsten Berger, Steven She, Rafael Lotufo, Andrzej Wąsowski, Krzysztof Czarnecki

GSDLAB–TR 2012–07–06 Version 2, March 2013

Generative Software Development Lab

University of Waterloo

Generative Software Development Laboratory
University of Waterloo
200 University Avenue West, Waterloo, Ontario, Canada N2L 3G1

WWW page: http://gsd.uwaterloo.ca/

Variability Modeling in the Systems Software Domain

Thorsten Berger, Steven She, Rafael Lotufo, Andrzej Wąsowski, and Krzysztof Czarnecki

Abstract—Variability models represent the common and variable features of products in a product line. Since the introduction of FODA in 1990, several variability modeling languages have been proposed in academia and industry, followed by hundreds of research papers on variability models and modeling. However, little is known about the practical use of such languages. We study the constructs, semantics, usage, and associated tools of two variability modeling languages, Kconfig and CDL, which are independently developed outside academia and used in large and significant software projects. We analyze 128 variability models found in twelve open source projects using these languages.
Our study (1) supports variability modeling research with empirical data of the real-world use of its flagship concepts. However, we (2) also provide requirements for concepts and mechanisms not commonly considered in academic techniques, and (3) challenge assumptions about size and complexity of variability models made in academic papers. These results are of interest to researchers working on variability modeling techniques, variability analysis techniques, and for tool designers, such as feature dependency checkers and interactive product configurators.

1 Introduction

Variability models represent the common and variable characteristics, or *features*, of products in a product line. Software engineers use them to design a product line architecture and to maintain it by adding and evolving features and their dependencies. Product line users derive concrete products from variability models. A range of automated tools supports these activities: analyzers verify model consistency or detect dead features [1]; graphical configuration tools (*configurators* for short) support intelligent choice propagation and model completion [2], [3], [4]. These tools are usually optimized for models with certain properties, such as size and density of constraints, due to the computational hardness of configuration problems.

Practical significance of variability modeling is reflected in the rise of industrial tools, such as pure::variants by Pure Systems GmbH and Gears by Big Lever Software Inc., or in the recent inception of the Feature Model[1] sub-project of the Eclipse Modeling Framework Technology umbrella. Recognizing the interest, the OMG currently develops the Common Variability Language standard (CVL) [5].

Unfortunately, even though variability modeling languages have been designed both in academia [6], [7], [8] and industry (pure::variants, Gears), little scientific data has been published about their practical use and the properties of their instances. Recent surveys [9], [10], [11], [1] report hundreds of research contributions on feature models, but no empirical studies on their practical application.

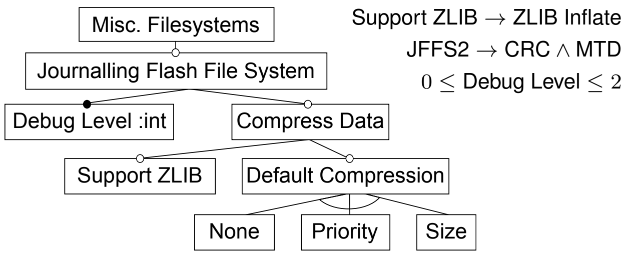


Fig. 1. Feature model interpretation of JFFS2

Feature models are among the most well-researched variability modeling languages. They were originally introduced as part of the Feature-Oriented Domain Analysis (FODA) methodology [6]. They gained popularity with product-line researchers and practitioners alike—mostly due to the simple and intuitive notation. Feature models are tree-like menus of configuration options, or features, with cross-tree constraints among the features. Figure 1 presents a sample feature model in the FODA notation. The model illustrates the core concepts shared by many feature modeling languages. It shows the variability of the Journalling Flash File System—one of the numerous files systems supported in open source operating systems. The boxes represent features. The hierarchy represents dependencies; for instance, the Default Compression feature allows a further choice of sub-features that refine it: None, Priority, or Size. Filled dots mark *mandatory* features (like Debug Level), which must be selected if the parent is. Hollow dots represent *optional* features, which do not have this constraint. Further, several features can be related by a *group constraint*: the sub-features of Default Compression are connected by an arc denoting

• *T. Berger is with the IT University of Copenhagen, Denmark.*
• *S. She, R. Lotufo, and K. Czarnecki are members of the Generative Software Development Lab, University of Waterloo, Canada.*
• *A. Wąsowski is with the IT University of Copenhagen, Denmark.*

1. http://eclipse.org/proposals/feature-model

the XOR group constraint—exactly one of the three choices has to be selected. Finally, textual cross-tree constraints are listed to the right.

The popularity of feature models resulted in many extensions and analysis techniques that build upon them. But while comparative analyses of many of these academic languages exist, there is no accessible data on how the various language constructs are used. Large models have been reported [12], [13], [14], but are not available to research. Public model repositories, such as S.P.L.O.T. (Software Product Line Online Tools [15]) and SPL2Go [16], only contain relatively small and mostly academic models, expressed in very simple languages. Whether the models reflect the complexity of real-world models remains speculation.

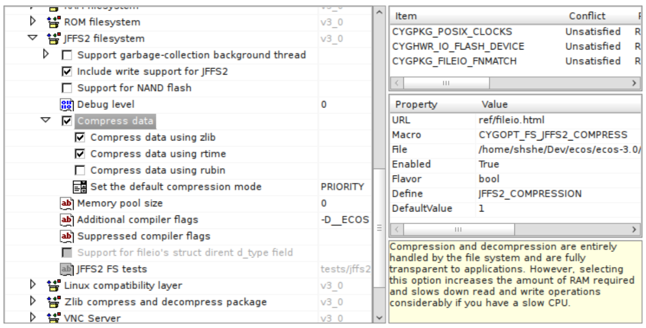
Our study aims at addressing this gap. We study real-world variability modeling languages *and* their models. Therefore, we investigate rich languages that i) use a hierarchical organization of features, ii) are used in real projects, iii) are equipped with configurators whose source code can be inspected, and iv) have publicly available models. These should be centralized easily identifiable models contained within a project, which excludes manifests of package management systems that are cross-project specification means. We found two languages matching these criteria: Kconfig [17] and the Component Definition Language (CDL) [18]. Both were conceived as part of open-source operating systems. Kconfig was created to describe the variability of the Linux kernel. At least ten other open source projects have also adopted the language. Similarly, CDL emerged as part of eCos, a real-time operating system for embedded devices. Both Linux and eCos have vast configuration spaces with thousands of features, which explains their need for variability management. Complex variability has driven the inception and evolution of both languages.

Kconfig and CDL are interesting and highly relevant study objects. Designed not by researchers, but by developers of large industrial-strength systems, they are tailored to satisfy the needs of these large projects (8M SLOC for Linux and 0.9M SLOC for eCos). The size of the models with up to 6320 features witnesses the scalability of the respective modeling approaches. Furthermore, both languages were developed independently from each other, and independently from feature modeling research. Since they share many similar concepts, they can confirm the importance of the modeling constructs discussed in the literature. As all our study objects are open source, they can be studied openly, and researchers can independently validate and replicate such studies.

Our study addresses two main research questions.

RQ1: *What variability modeling concepts are used in real-world languages, and how?*

We analyze the concepts and the semantics of Kconfig and CDL, and compare them against the



```
▷ ROM filesystem                                        v3_0
▽ JFFS2 filesystem                                      v3_0
   ▷ ☐ Support garbage-collection background thread
     ☑ Include write support for JFFS2
     ☐ Support for NAND flash
       Debug level                                      0
   ▽ ☑ Compress data
        ☑ Compress data using zlib
        ☑ Compress data using rtime
        ☐ Compress data using rubin
          Set the default compression mode              PRIORITY
       Memory pool size                                 0
       Additional compiler flags                        -D__ECOS
       Suppressed compiler flags
     ☐ Support for fileio's struct dirent d_type field
       JFFS2 FS tests                                   tests/jffs2
▷ Linux compatibility layer                             v3_0
▷ Zlib compress and decompress package                  v3_0
▷ VNC Server                                            v3_0
```

Item	Conflict
CYGPKG_POSIX_CLOCKS	Unsatisfied
CYGHWR_IO_FLASH_DEVICE	Unsatisfied
CYGPKG_FILEIO_FNMATCH	Unsatisfied

Property	Value
URL	ref/fileio.html
Macro	CYGOPT_FS_JFFS2_COMPRESS
File	/home/shshe/Dev/ecos/ecos-3.0/
Enabled	True
Flavor	bool
Define	JFFS2_COMPRESSION
DefaultValue	1

Compression and decompression are entirely handled by the file system and are fully transparent to applications. However, selecting this option increases the amount of RAM required and slows down read and write operations considerably if you have a slow CPU.

Fig. 2. The eCos configurator GUI

well-established concepts of *feature modeling*—our main frame of reference. We also inspect the configurators of both languages.

Our goal is to provide empirical data on the use of variability modeling concepts in languages originating from practice, and to widen the understanding of the design space of variability modeling. By inspecting the configurators, we aim to characterize the configuration processes and the analysis support available for our subjects, but also to discuss potential improvements.

RQ2: *What are the characteristics of real variability models?*

We study all Kconfig and CDL models we can find. We analyze their content, their structure, and their constraints. We define metrics and develop an analysis infrastructure to derive core characteristics. Finally, we compare our subjects to publicly available models in the S.P.L.O.T. repository.

Our goal is i) to investigate whether and how frequently the identified variability modeling concepts are used in real models, ii) to understand their content, structure, and their benefit for the projects, and iii) to assess whether, and if so how, these models differ from typical models used in academic literature.

With respect to **RQ1**, our study shows that the core concepts used in feature modeling are supported by both Kconfig and CDL: Boolean (optional), integer and string features, a hierarchy, group constraints, and cross-tree constraints. Interestingly, both languages and the models use concepts that are beyond core FODA concepts and have not been as widely studied, for instance, visibility conditions, computed defaults, binding modes, and domain-specific vocabulary. Most of these concepts aim at scaling variability modeling. Finally, our study reveals interesting insights in how Kconfig and CDL provide these concepts.

We observe limitations in the configurators for Kconfig and CDL (see Fig. 2 for the user interface of the latter). The Kconfig configurator lacks reasoning procedures to support choice propagation, which instead is handled by an error-prone imperative statement. eCos boasts a far more intelligent configurator, based on a home-grown inference engine. Unfortunately, the reasoning procedures of the engine are incomplete and may propose undesirable configuration choices.

Interestingly, both configurators follow a *reconfiguration* paradigm: any configuration task starts with an initial, possibly default, configuration and continues by modifying this initial configuration.

With respect to **RQ2**, our study shows that the investigated models represent a new class of models that differ from most of the models used in literature. They are significantly larger, have a very different shape, and are used to configure nearly any variable aspect of the projects. They contain surprisingly many data features, including integers, floats, and strings. They also have more constraints—including derived (computed) features—than commonly assumed in literature. Our analysis shows that the high connectivity and density of their dependency graph challenges many existing model reasoners.

Our results can be used in various ways. The models and their characteristics provide the basis for benchmarking and optimizing configurators and reasoners. The occurrence and frequency of modeling concepts in these models can be used to prioritize the implementation of concepts in tools. Our results about the content and structure of models can help developers organizing their features when creating and maintaining models. The language analysis provides a fresh view on syntax and semantics of variability modeling languages, widens the understanding of their design space, and identifies problems that can occur by making certain design decisions. For example, we provide requirements for constraint languages within the systems software domain, confirm the need for concepts such as derived defaults, and discover intricate semantic interactions between some language constructs.

We believe that our empirical data is of interest to a growing audience of variability modeling practitioners and researchers, especially in standardization efforts, such as the ongoing development of OMG's CVL standard. In fact, two authors of this work actively contribute to it.

We see our work as one, but self-contained step within the long-term effort of improving the empirical understanding of variability modeling. Our results have to be complemented using other methods, such as interviews or user studies, and by studies of other domains, which confirm or refute our conclusions. This process is known as theory-building from cases [19], [20] and will eventually lead to a refined theory behind variability modeling.

An earlier version of this work has appeared in [21]. In this version, we broaden the scope of our study and analyze all models in the respective languages we can identify (only two were studied before), which amounts to a total of 128 models. Thus, our focus is shifted towards the analysis of models, rather then the exploratory investigation of concepts in variability modeling languages. Nevertheless, we add some more discussion about these concepts, including the new concept of capabilities. We also add a qualitative discussion of structure and content of 13 models. We improve presentation of the entire work, discuss additional concepts, and correct some minor problems of the original version.

The remainder of this paper is organized as follows. Section 3 introduces the subject systems and their models. Section 4 describes concepts of Kconfig and CDL. Section 5 reports the results of our model analyses; more precisely, it describes content, organization, and constraints of the models. Section 6 examines the configurator tools, focusing on their user assistance and reasoning capabilities. Section 8 discusses threats to validity and Section 9 the related work. Section 10 concludes and summarizes the most interesting findings.

2 Methodology

To address question **RQ1**, we qualitatively analyzed the Kconfig and CDL languages. We set off to reverse engineer and formalize the semantics of both. We extracted the information in user documentation and tested the tools on the models found in the open source projects and on manually created examples. Finally we inspected implementation code of these tools. The result is available in two technical notes on formal semantics [22], [23]. In order to enable SAT-based analyses, we also developed propositional abstractions of these semantics. The entire process allowed us to understand the languages in depth and discover many subtle differences and connections.

Subsequently, we have mapped the concepts of the FODA feature models to corresponding concepts in Kconfig and CDL according to their semantics. As a side effect, we have obtained a list of concepts not supported in FODA.

To discuss the tool support, we inspected the configurators of the two languages and their source code with respect to configuration process, user assistance, and reasoning support—in particular, the facilities to propagate choices and to resolve conflicts (unsatisfied constraints).

To understand the characteristics of real variability models (question **RQ2**), we searched online for other open source projects using CDL or Kconfig, beyond the host projects of eCos and the Linux kernel. We have identified 128 models, which we analyzed qualitatively and quantitatively. The qualitative part focuses on characterizing the model contents and organizational structures reflected in the feature hierarchy. To understand the content, we manually inspected the models and iteratively developed a classification schema for features. This schema was discussed among the authors, and after reaching consensus, it was used to characterize the content of each model. To identify patterns of feature organization, we inspected the first three levels of the hierarchy shown in the configurator for each model.

TABLE 1
Projects and their models

Language	Model	Version	Features
Kconfig	**Linux X86**	2.6.32	6320
	axTLS	1.2.7	108
	BuildRoot	2010.11	1938
	BusyBox	1.18.0	881
	CoreBoot	4.0	2269
	EmbToolkit	0.1.0-rc12	1357
	Fiasco	2011081207	171
	Freetz	1.1.3	3471
	ToyBox	0.1.0	71
	uClibc	0.9.31	369
	uClinux-base	20100825	383
	uClinux-dist	20100825	1620
CDL	**eCos i386PC**	3.0	1256
	all 116 models	3.0	$1116^1 \leq 1247^2 \leq 1396^3$

[1] min [2] mean [3] max

For the quantitative analysis, we developed our own analysis tools LVAT[2] and CDLTools[3]. We first extended the original configurators of Linux and eCos to exploit their parsers and to export the relevant data (feature tree and feature properties) into our own format. We then loaded these files into our developed analysis infrastructure to calculate statistics, and to further transform the models into propositional formula for SAT-based analyses (see Sections 5.2.2 and 8). Both infrastructures are freely available.

To characterize feature hierarchy, feature kinds, feature representation, and most importantly, constraints as the primary source of complexity, we re-used and defined appropriate metrics. For each model, their values are given in tables or visualized using diagrams throughout Section 5.

Finally, we compare our set of models to those available in the S.P.L.O.T. repository by aggregating core characteristics. Since the S.P.L.O.T. feature modeling language is less expressive than Kconfig and CDL, we relied on a subset of metrics to compare both datasets.

3 The Systems

Table 1 lists the subject systems and the sizes of their variability models. Twelve of these are Kconfig models stemming from different projects, and 116 are CDL models stemming from eCos. Since Linux' code base contains models for multiple hardware architectures (23), we focus on its Intel x86 models. We are confident that our extensive search has identified all open-source models expressed in Kconfig and CDL available as of March 2013.

The models range from very small (ToyBox, 72 features) to very large (Linux, 6320 features). The average number of features is 1708 (median 1357) for the Kconfig, and 1247 (median 1250) for the CDL models. This means that most of our subjects are among the largest variability models known so far, especially when compared to the feature models available in academia (cf. Section 5.4).

2. http://code.google.com/p/linux-variability-analysis-tools
3. http://code.google.com/p/variability/wiki/CDLTools

3.1 Kconfig Systems

The Kconfig language and its tools were designed for the Linux kernel and are developed and distributed together with the kernel codebase. Although it never became a standalone project, Kconfig has been adopted by at least ten other open source projects in the systems domain—perhaps naturally, as the strict resource requirements of systems software often require static configuration. In the following, we briefly introduce the systems using Kconfig, starting with Linux and following with the other projects in alphabetic order.

3.1.1 Linux Kernel

Kconfig is used to specify build-time configurations of the Linux kernel since 2002. The graphical configurator (*xconfig*) reads the Kconfig model and allows users to select features in a user interface closely resembling the CDL configurator of Fig. 2. It outputs a set of feature symbol and value mappings that are referenced in Makefiles and in the source code as preprocessor directives.

The studied version 2.6.32 of the Linux kernel supports 23 hardware architectures. The code base spans 1880 directories and 701 Kconfig files. Kconfig models are distributed over multiple files, organized according to the source code hierarchy. Each Kconfig specification is placed alongside the related code. An architecture-specific Kconfig file is used as a starting point for the specification; a simple inclusion mechanism is used to include other files. The x86 architecture model is distributed over 504 Kconfig files.

3.1.2 axTLS

AxTLS is a small, memory-optimized client/server library implementing the TLSv1 SSL protocol. It contains a tiny http and https server, test tools, and various interfaces for major programming languages, such as Java and C#. AxTLS' model is rather small with 108 features distributed over five Kconfig files.

3.1.3 BuildRoot

BuildRoot is a tool for developers of embedded systems that generates a complete embedded Linux system, with a root file system and all necessary packages, as opposed to just a kernel. The project is a large collection of scripts to stepwise generate the system. All steps are configurable and comprise: downloading and building a cross-development toolchain for the target architecture; building development and debugging tools; building core system programs and shell commands (preferably BusyBox- and uClibc-based, see below); as well as installing a kernel and boot loader. BuildRoot also has several hundreds of packages containing user space applications and libraries, such as GUI-, networking- or system-related programs.

3.1.4 BusyBox

BusyBox is a command-line tool for Linux-based embedded systems that combines many standard shell commands, such as `ls`, `cp` or `rm`, in a single executable. The BusyBox configurator allows customizing the executable by selecting only commands and capabilities needed on the target system. In particular, it allows linking BusyBox to uClibc (described shortly) to save even more space.

3.1.5 CoreBoot

The CoreBoot project delivers a free open source BIOS as an alternative to proprietary BIOS implementations in PCs and Workstations. CoreBoot provides the basic code that is necessary to initialize the mainboard with all its devices, such as RAM, PCI bus, and serial interface. After initialization, CoreBoot executes third-party payload, which can be a bootloader for an operating system, device-specific firmware (such as the sub-project OpenBios), or an operating system kernel directly.

3.1.6 EmbToolkit

EmbToolkit (Embedded Systems Toolkit) is a build system designed for embedded system developers similar to BuildRoot. EmbToolkit creates a cross-development toolchain with a custom C compiler, C library, and other development and debugging tools. The preferred C library is EGLIBC (another lightweight implementation of the standard C library), but uClibc can also be used alternatively. EmbToolkit generates a root filesystem containing core system tools including BusyBox and GUI-, networking- or system-related programs. These are installed as packages selectable in the configurator.

3.1.7 Fiasco

Fiasco is a derivative of the L4 microkernel family, used in conjunction with the real-time operating system DROPS[4]. Fiasco runs on a variety of systems, ranging from small embedded to large multi-processor architectures with Intel x86, ARM, or PowerPC processors. It supports preemptive multi-tasking with hard priorities for processes, hardware-assisted virtualization, in-kernel debugging, and provides an object-oriented kernel API. In contrast to the Linux kernel, Fiasco has much fewer configuration options; they concern the target hardware, debugging, and build configuration options, but not whole subsystems or drivers, which are outside the kernel.

4. http://os.inf.tu-dresden.de/drops/overview.html

3.1.8 Freetz

The Freetz (for *Free Fritz*) project provides an alternative firmware for consumer internet routers of the popular AVM FritzBox series. Freetz extends the proprietary firmware with extra functionality, such as an improved firewall, various servers (such as HTTP, VPN, SMB), and many other tools as packages. It also allows users to remove unnecessary features of the original firmware by selecting individual patches.

3.1.9 ToyBox

ToyBox is our smallest subject. It has the same goal as BusyBox: combining a subset of the GNU shell commands into one executable. It was started by a former BusyBox maintainer, who found that BusyBox was too difficult to extend. ToyBox currently implements 35 of BusyBox' 309 commands and three additional ones. The project appears to have been a playground for the author. It is now largely abandoned, the last release dates back to the end of 2009.

3.1.10 uClinux

uClinux is a Linux distribution for embedded systems. At its core is a tailored version of the Linux kernel for micro-controllers, which today supports 14 hardware architectures, such as ARM, ADI Blackfin, or MIPS. Originally created as a fork of the Linux 2.2 kernel, it is widely recognized today and its core parts had been merged into the official Linux kernel.

The configuration of uClinux forgoes in a multi-level fashion (similar to staged configuration [24]) in three steps, each governed by a dedicated Kconfig model. First, basic features are configured: hardware architecture and libraries (the uClinux-base model in Table 1). Then the kernel is configured using templates for supported architectures. Finally, a wide variety of software packages can be selected from the uClinux distribution (the uClinux-dist model in Table 1). We study the model of the first and the third step. The second step model is essentially a Linux kernel model for the target hardware—similar to the mainline x86 kernel model already included in our study.

3.1.11 uClibc

Initially a sub-product of uClinux, the uClibc project is now an independent implementation of the standard C library for embedded microprocessors. It provides only a tailored subset of the functions present in the regular C library (glibc) used with Linux distributions, excluding functionality not needed on embedded systems. To further address the space requirements, it can be configured to support a minimal set of needed functions, reflecting the needs of a given project.

3.2 eCos CDL

The Component Definition Language (CDL) was designed to directly meet the needs of the configurable

```
k-1  menuconfig MISC_FILESYSTEMS
k-2    bool "Miscellaneous filesystems"
k-3
k-4    if MISC_FILESYSTEMS
k-5
k-6     config JFFS2_FS
k-7      tristate "Journalling Flash File System" if MTD
k-8      select CRC32 if MTD
k-9
k-10
k-11
k-12
k-13    config JFFS2_FS_DEBUG
k-14     int "JFFS2 Debug level (0=quiet, 2=noisy)"
k-15     depends on JFFS2_FS
k-16     default 0
k-17     range 0 2
k-18     --- help ---
k-19       Debug verbosity of ...
k-20
k-21
k-22    config JFFS2_FS_WRITEBUFFER
k-23     bool
k-24     depends on JFFS2_FS
k-25     default HAS_IOMEM
k-26
k-27
k-28    config JFFS2_COMPRESS
k-29     bool "Advanced compression options for JFFS2"
k-30     depends on JFFS2_FS
k-31
k-32    config JFFS2_ZLIB
k-33     bool "Compress w/zlib..." if JFFS2_COMPRESS
k-34     depends on JFFS2_FS
k-35     select ZLIB_INFLATE
k-36     default y
k-37
k-38    choice
k-39     prompt "Default compression" if JFFS2_COMPRESS
k-40     default JFFS2_CMODE_PRIORITY
k-41     depends on JFFS2_FS
k-42     config JFFS2_CMODE_NONE
k-43      bool "no compression"
k-44     config JFFS2_CMODE_PRIORITY
k-45      bool "priority"
k-46     config JFFS2_CMODE_SIZE
k-47      bool "size (EXPERIMENTAL)"
k-48    endchoice
k-49  endif
```

```
c-1  cdl_component MISC_FILESYSTEMS {
c-2   display   "Miscellaneous filesystems"
c-3   flavor    none
c-4  }
c-5
c-6  cdl_package CYGPKG_FS_JFFS2 {
c-7    display     "Journalling Flash File System"
c-8    requires    CYGPKG_CRC
c-9    implements  CYGINT_IO_FILEIO
c-10   parent      MISC_FILESYSTEMS
c-11   active_if   MTD
c-12
c-13   cdl_option CYGOPT_FS_JFFS2_DEBUG {
c-14    display          "Debug level"
c-15    flavor           data
c-16    default_value    0
c-17    legal_values     0 to 2
c-18    define           CONFIG_JFFS2_FS_DEBUG
c-19    description      "Debug verbosity of...."
c-20   }
c-21
c-22   cdl_option CYGOPT_FS_JFFS2_NAND {
c-23    flavor           bool
c-24    define           CONFIG_JFFS2_FS_WRITEBUFFER
c-25    calculated       HAS_IOMEM
c-26   }
c-27
c-28   cdl_component CYGOPT_FS_JFFS2_COMPRESS {
c-29    display          "Compress data"
c-30    default_value    1
c-31
c-32      cdl_option CYGOPT_FS_JFFS2_COMPRESS_ZLIB {
c-33       display          "Compress data using zlib"
c-34       requires         CYGPKG_COMPRESS_ZLIB
c-35       default_value    1
c-36      }
c-37
c-38      cdl_option CYGOPT_FS_JFFS2_COMPRESS_CMODE {
c-39       display          "Set the default compression mode"
c-40       flavor           data
c-41       default_value    { "PRIORITY" }
c-42       legal_values     { "NONE" "PRIORITY" "SIZE" }
c-43    }
c-44   }
c-45 }
c-46
c-47
c-48
c-49
```

Fig. 3. A model excerpt expressed in Kconfig (left) and CDL (right). Corresponding definitions are aligned.

embedded operating system eCos, which aims at a a high degree of portability, low memory usage, and small code image sizes. With a market share of 5-6%, eCos powers, for example, multimedia, networking, automotive, and even satellite and space-based devices.[5]

Unlike Kconfig, which is a standalone domain-specific language (DSL), CDL is an internal DSL [25] embedded in Tcl—an extensible dynamic scripting language. CDL inherits characteristics from Tcl, such as syntactic nesting of blocks and the ability to embed Tcl control structures (conditional statements, for-loops) in models. CDL's configurator has an inference engine to support interactive conflict resolution. Recall that Fig. 2 presents a glimpse of the configurator's user interface.

We studied version 3.0 of eCos, which supports 116 hardware architectures, called *targets*, and comprises almost a million lines of code. The code base is divided into 500 packages, each containing the source code and a set of CDL files declaring the variability of the package. Each target defines a set of packages specific to the hardware architecture. So-called *templates* aggregate packages with hardware-independent functionality. In the configurator, a user first selects a *target* and then one of the *templates*; finally, the user may decide to load additional packages into the configuration tree. We have chosen to study the model of the i386PC target and the so-called all template—the most inclusive template containing almost all hardware-independent packages.

5. http://ecoscentric.com/ecos/examples.shtml

4 The Languages

We now summarize the key concepts found in the languages. We use the feature model in Fig. 1 as the running example. Fig. 3 shows the same model in Kconfig (to the left) and CDL (to the right). Both snippets are extracted from the original Linux and eCos models. They define the features of the Journalling Flash File System, version 2 (JFFS2), supported by both systems. In fact, eCos's JFFS2 implementation was ported from Linux. JFFS2 is one of the very few of such ports, but it makes an ideal example to illustrate the similarities and differences between Kconfig and CDL. To give a realistic impression of both languages, we keep the examples close to the originals; in particular, we retain the original identifiers, which differ somewhat from the names in Fig. 1. The few lines introduced purely for the purpose of the example are

TABLE 2
Mapping of concepts between Kconfig, CDL, and feature modeling

	concept	Kconfig	CDL	feature modeling
feature kinds	Grouping	menu, menuconfig, choice	package, component	feature (non-leaf)
	Individual	config	option, interface	feature (leaf)
feature representation	Composition	single value	bool. value with opt. data value	bool. value with opt. attribute[1]
	Feature type			
	Switch	bool, tristate	bool, booldata	(optional)
	Data	hex, int, string	booldata, data	integer[1], string[1]
	None	(menu)	none	(mandatory)
feature hierarchy	Specification	syntactic and computed in configurator	syntactic and re-parenting	syntactic
	Child-to-parent impl.	visibility	configuration & visibility	configuration
	Root	synthetic	synthetic	explicit
group constraints	Mutex $[0..1]$	optional Boolean choice	interface constraint, INT ≤ 1	MUTEX group[1][26]
	Or $[1..*]$	mandatory tristate choice	interface constraint, INT ≥ 1	OR group [26]
	Xor $[1..1]$	mandatory Boolean choice	interface constraint, INT $= 1$	XOR group
	Interval $[m..n]$	N/A	interface constraint, $m \leq$ INT $\leq n$	$[m..n]$ group[1][27]
	Cross-hierarchy group	N/A	interface constraint, $m \leq$ INT $\leq n$	N/A
feature constraints	Configuration	select	requires, active_if	cross-tree constraint
	Value restrictions	range	legal_values	cross-tree constraint[1], enum attribute[1][8]
	Derived features	non-prompt default	calculated, interface	rare[1][7]
	Defaults	prompt default	default_value	rare[1][26]
	Visibility conditions	prompt condition	active_if	rare[1][7], [28]
	Expression operators	&&, ||, !, =, !=	also inequality, arithm. and str. ops.	not standardized[2]
	Binding modes	three-value logic	N/A	rare[1][26], [28]
other	Textual content	prompt, help	display, description	description
	Modularization	textual inclusion	dynamic loading/unloading	rare[1][8], [29], [30], [31]
	Build symbols	one-to-one	one-to-many	unspecified[1]
	Code mappings	no, uses KBuild (m:n)	yes (1:n), and build specifications	N/A[1]

[1] Not supported in the S.P.L.O.T. model repository [2] S.P.L.O.T. expression operators: &&, ||, !

underlined, and we leave out some unnecessary parts of the corresponding sources to avoid clutter. Notice that we drew the Debug Level feature in Fig. 1 using the mandatory feature with integer attribute notation introduced by FODA [6], because it closely resembles how features are represented in Kconfig and CDL. There is, so far, no consensus on a unified notation for attributes in feature models [1].

The features shown in Fig. 3 configure different aspects of the JFFS2 filesystem driver. The first child of the main JFFS2 feature sets the debugging level, which is an integer ranging from 0 to 2; the second one enables a write buffer; and the third one configures the compression capability of the filesystem. The third one is further subdivided into features configuring the use of the ZLIB library for compression, and setting the compression mode to one of none, priority or size. As the example shows, the developers of Linux and eCos not only used different language constructs, but also a slightly different structure to model the filesystem's configurability. The screenshot in Fig. 2 shows the configuration of JFFS2 in the CDL configurator.

Our discussion in Sections 4.1 to 4.6 follows the outline given in Table 2, from top to bottom. The table maps concepts from Kconfig and CDL to feature modeling. Thus, the last column does not show all existing feature modeling concepts, but only those we could map to Kconfig and CDL concepts. However, we provide citations for concepts that go beyond the original FODA notation. Since we will later compare the Kconfig and CDL models with models available in the S.P.L.O.T. repository, the last column also indicates which concepts are supported by the S.P.L.O.T. feature modeling language.

4.1 Feature Kinds

Both in CDL and Kconfig, features are labels organized in a hierarchy, as known from most variability modeling languages (Table 2, row 3). We introduce two orthogonal classifications for different kinds of features. First, we distinguish between *grouping* and *individual* features, according to their purpose in the feature hierarchy; see Table 2, row 1. Second, we distinguish between various roles that features can take.

Grouping and individual features. Grouping features are used to structure models by gathering a set of features as their children. Nevertheless, a grouping feature can also provide a configuration option. An example is the "Journalling Flash File System" in Fig. 1. Some grouping features further impose cardinality constraints on their children (see Section 4.4), such as the exclusive choice "Default Compression" in Fig. 1, which has exactly one selectable child at a time. In contrast, individual features have no children; they are leaves in the hierarchy. Individual features are used purely for providing configuration options.

Roles of features. Features that represent configuration options can take one or more of the following roles:

1) *User feature*: a configuration option that can be set by the user in a configurator, like all active (not grayed-out) features shown in Fig. 2;
2) *Implementation feature*: a configuration option accessed by the build system or a generator, like those referenced with #IF and #IFDEF preprocessor directives in the Linux code excerpts in Fig. 4;
3) *Derived feature*: a configuration option automatically computed via constraints, like "JFFS2 FS tests" with a grayed out value in Fig. 2.
4) *Capability*: an abstraction of functionality that can be provided by several features interchangeably. For example, Linux' `HAVE_IDE` feature represents hardware IDE support. Other features can depend on this capability instead of on a concrete IDE controller. This way coupling is reduced.

Since Kconfig and CDL are domain-specific languages, they provide specialized keywords for these different kinds of features.

In Kconfig, feature kinds reflect their appearance in the configurator UI. *Menus* are pure grouping features. *Menuconfigs* are grouping features that also represent configuration options; they look like menus that can be enabled and disabled by clicking. *Choices* are like menus or menuconfigs except that they also impose cardinality constraints on their children. *Configs* are individual features; however, some are rendered as grouping features with children in the configurator, as we will see later in Section 4.3.

Kconfig has no syntax to indicate the role of features. Every config or menuconfig can be an implementation feature, that is, their names can always be referenced in build scripts or code. *User* and *derived features* are distinguished by their *prompt* clause—a label shown to the user and declared right after the type of the feature, such as in Lines k-7 or k-14 in Fig. 3 (for details, see visibility of features in Section 4.5). Derived features have no prompt, their value is always restricted by constraints and cannot be changed directly by the user. Finally, *capabilities* are modeled by constraints that other features declare on them; more precisely, if a feature provides a capability, it declares a constraint that automatically selects the capability feature.

In Fig. 3, the menuconfig MISC_FILESYSTEMS (Line k-1) corresponds to the root node in Fig. 1. It contains a choice (k-38) corresponding to the parent feature of the XOR-group, Default Compression, and eight configs corresponding to the remaining features of Fig. 1—all enclosed by a pair of matching `if` (k-4) and `endif` (k-49) keywords. Among all individual features, JFFS2_FS_WRITEBUFFER (k-22) is a *derived feature* that is not visible in the configurator, because it has no prompt clause (k-23). Its value is calculated as equal to the value of the HAS_IOMEM *capability* (referenced in Line k-25, but defined elsewhere). All other individual features are both *user* and *implementation* features.

In CDL, feature kinds reflect types of implementation entities they map to: *packages* are top-level containers for features, mapping to eCos packages. *Components* are nested features grouping other features. *Options* are individual configuration options, nested under packages or components. Several—possibly exclusive—features can provide equivalent functionality required elsewhere. *Interfaces* represent such *capabilities*. In our example (Fig. 3), Line c-9 states that CYGPKG_FS_JFFS2 implements the interface CYGINT_IO_FILEIO (not shown). The value of an interface is the number of selected features implementing it. Declaring constraints over this value allows imposing cardinality constraints on the implementing features.

Packages and components represent both grouping and individual features; options and interfaces are always individual and cannot group features. By default, all features can be *implementation features* unless they explicitly suppress defining a symbol with the keyword `no_define`. Being a *user* or *derived* feature is determined by the declared constraints, except for interfaces, which are are always *derived* and not shown to the user. Interfaces explicitly represent capabilities.

```
#if CONFIG_JFFS2_FS_DEBUG > 0
/* Enable "paranoia" checks and dumps */
#define JFFS2_DBG_PARANOIA_CHECKS
#define JFFS2_DBG_DUMPS

...

#ifdef CONFIG_JFFS2_ZLIB
        jffs2_zlib_init();
#endif
```

Fig. 4. Feature symbols referenced in code

4.2 Feature Representation

The semantics of a feature model is a set of configurations. A configuration specifies the presence or absence of each feature, and a value for the related integer or string if the feature is present (when applicable). The configurations are represented differently in the Kconfig and CDL language (see Table 2, row 2). Partial configurations, where features can be in an undecided state, are not supported by Kconfig and CDL, as described later in Section 6.

In Kconfig, a configuration assigns a single value to each feature. If F is the set of all features in the model, and Val is a set of all possible values, then a particular configuration σ maps features to values:

$$\begin{aligned}\sigma\colon F &\mapsto \text{Val}\\ &\text{and if } \sigma(f)=v\text{, then } v\in\text{type-of}(f) \qquad (1)\end{aligned}$$

Table 2 lists the possible feature types in three categories: *switch*, *data*, and *none*.

Switch features appear as a checkbox in the configurator. Data features allow the user to input a value in a text box. Kconfig's menus have no type, which corresponds to features of type `none` in CDL (see below). In (1), we assume that the type *none* contains a single uninterpreted element representing no value.

The Kconfig type `bool` has two values, `y` and `n`, internally represented by 2 and 0. The latter, 0, denotes feature absence, while 2 means that the feature's implementation should be compiled statically into the kernel. `Tristate` resembles `bool`, except for the additional value `m`, internally represented by 1. It indicates that the implementation should be compiled as a dynamically loadable module—Linux' mechanism to load drivers at runtime. For example, for the `tristate` feature JFFS2_FS (k-6), the user can choose to deselect it, to create a dynamically linked module, or to link it statically. Its descendant JFFS2_ZLIB (k-32) of type `bool` can only be de-activated; but when selected, its implementation is always linked statically into the JFFS2 compilation unit, without creating a separate module.

Kconfig supports two integer types: `int` (decimal) and `hex` (hexadecimal). Both types also allow an empty value, which is used to encode the absence of an integer feature. The type `string` is ambiguous in this respect: a string feature with the empty value can be seen as a present feature with that value or an absent feature; the two cases are indistinguishable.

In CDL, every feature is composed of two values: an *enabled value* and a *data value*. The enabled value is a Boolean and encodes the presence or absence of the feature; the data value is dynamically typed and used to store numbers and strings. Thus, a configuration maps features to value pairs:

$$\begin{aligned}&\sigma\colon F \mapsto \{0,1\}\times \text{Val}\\ &\qquad\text{and if } \sigma(f)=(e,d),\text{ then } d\in\text{type-of}(f)\end{aligned} \tag{2}$$

CDL terminology for a feature type is *flavor*. Flavors map to FODA features as follows:

`none`	$\mapsto$	Mandatory with no attribute
`bool`	$\mapsto$	Optional with no attribute
`data`	$\mapsto$	Mandatory with attribute
`booldata`	$\mapsto$	Optional with attribute

The example model in Fig. 3 includes features of various flavours. CYGOPT_FS_JFFS2_DEBUG (c-13) of flavour data takes numeric values. CYGOPT_JFFS2_NAND (c-22) takes Boolean values (flavour bool), and the data feature CYGOPT_FS_JFFS2_COMPRESS_CMODE (c-38) assumes string values.

An important aspect of variability modeling techniques is the support for partial configurations. Kconfig has only very basic support for them. The configurator allows to update a configuration for an old model (previous version of the corresponding project) to a new one, by removing features that do not exist anymore and adding default values for new features to the configuration. However, the mechanism is very simple and can cause invalid configurations, as mentioned in the documentation. In CDL, expanding a partial configuration to a full configuration is explicitly supported. In fact, it was the standard way of configuring eCos with the preceding non-graphical configuration. The current graphical configurator employs its inference engine to assure a valid full configuration.

4.3 Feature Hierarchy

All major variability modeling languages that stem from academic research admit a single feature hierarchy in the model, which is then reused in the respective configuration tools. In the FODA example in Fig. 1, the diagrammatic tree represents both the intended configuration hierarchy and the syntactic nesting.

In contrast, the hierarchies displayed in the Kconfig and CDL configurators deviate from the syntactic structure of the models. Thus, we distinguish between the syntactic model hierarchy and the configurator hierarchy. The former is given by the syntactic nesting of features in the model, such as the nesting of configs under menus or choices in Kconfig, or options and components under other components and packages in CDL. The latter is shown to the user in the configurator, as in Fig. 2.

In Kconfig, syntactic nesting within menuconfigs and choices is reflected in the configurator hierarchy. However, configs can also appear as children of other configs in the configurator, even though they cannot be nested syntactically in the model. The configurator has an algorithm to additionally nest syntactic sibling configs based on their declared dependencies. For example, a group of consecutive configs declaring dependency on the same parent (lines k-13–25) is placed under this parent (JFFS2_FS).

In CDL, the configurator hierarchy follows the syntactic nesting of features, unless declared otherwise. *Re-parenting* is a mechanism to explicitly specify a different parent for a feature than its syntactic scope in the model (see Line c-10). It allows adjusting the developer-oriented structure of the model, which is primarily driven by eCos' packaging mechanism, to a more user-oriented view, before it is shown in the configurator.

An important property of the feature hierarchy in FODA-like languages is that the presence of a child feature implies the presence of its parent: for each edge from child c to parent p, we have that $\sigma(c) \rightarrow \sigma(p)$. The configurator hierarchy in CDL has this property too. In contrast, the configurator hierarchy in Kconfig only enforces visibility between a child and its parent—a feature is visible if its parent is visible. However, if the parent is not selected, a feature can still be selected (even exclude the parent) in some cases. Such a configuration is still valid in Kconfig, unlike in any other feature modeling language known to us.

Deselecting branches in the configurators has different impact on the configuration. In Kconfig, the previous values of all descendants are kept in memory, but not saved and therefore lost on reload—except for invisible derived features. In CDL, the user-selected values will always be saved in the configuration file; they are annotated with the source of the selection (by user, by inference engine, or by a default; cf. Section 6), thus, re-enabling a branch always restores previous values.

Finally, both Kconfig and CDL configurators show a *synthetic root*—a fresh root node that is not explicitly specified in the model. This enables working with diagrams that are forests and not trees like in FODA.

4.4 Group Constraints

In feature modeling, group constraints restrict the number of selectable sibling features if their parent is selected (Table 2, row 4): exactly one child for XOR, at least one for OR, and at most one for MUTEX. Alternatively, the constraint can be given as an interval.

In Kconfig, the `choice` keyword groups a set of features and imposes a group constraint on them—either XOR or MUTEX. Technically, such a choice is either `bool` or `tristate` with a mandatory or optional modifier flag. If not specified otherwise, a choice is mandatory and `bool`, which semantically represents an XOR group, such as the choice in line k-38[6]. If the choice is optional and `bool`, it realizes a MUTEX group. `Tristate` choices behave differently and cannot be interpreted as feature modeling groups. Mandatory `tristate` choices either admit exactly one feature set to `y` (all others to `n`), or any number of features set to `m`. This behavior is useful if various drivers exist for one hardware device where only one can be compiled into the kernel, but all can be built as modules. This realizes an XOR group at runtime, as only one driver can be loaded per device. Finally, optional `tristate` choices—surprisingly—do not impose any cardinality constraint. Note that previously [21], we misinterpreted the semantics of tristate choices as OR groups.

CDL interfaces are a more expressive construct for restricting cardinality of a set of features beyond OR, XOR, and MUTEX. The value of an interface counts the number of its selected implementations (concrete features). Restricting this value introduces a cardinality constraint ($= 1$ for XOR, ≥ 1 for OR, and < 1 for MUTEX). In contrast to FODA-like languages, CDL does not require that all implementing features are siblings—the feature activating the group constraint does not need to be a parent of the constrained features, which allows creating groups that cross-cut the hierarchy.

4.5 Feature Constraints

CDL and Kconfig support three types of constraints (Table 2, row 5): (1) *configuration constraints* restrict the legal combinations and values of features; (2) *defaults* provide default values for features, possibly depending on other features (computed defaults); they can be overridden by the user; (3) *visibility conditions* control the visibility of features in the configurator UI. Features whose visibility condition is false are not shown or otherwise disabled for user input. Computed defaults and visibility conditions have not been widely considered in feature modeling. Unlike configuration constraints, defaults and visibility conditions have no direct impact on the configuration semantics. However, they interact with each other in complex ways that may impact configuration semantics. We will explain this soon.

A configuration constraint is expressed using `select` in Kconfig and `requires` or `active_if` in CDL. For instance, the dependency Support ZLIB$\rightarrow$ZLIB Inflate of Fig. 1 is expressed as a `select` in line k-35 and as a `requires` in c-34. Both `select` and `requires` take a condition, say p, and denote the configuration constraint $f \rightarrow p$, where f is the feature in which they are defined. While p can only be a feature identifier for `select` (Kconfig), it can be an arbitrary Boolean expression for `requires` (CDL), possibly accessing multiple features via logical, arithmetic, and string operators.

CDL's `active_if` has the same syntactic form and configuration semantics as `requires`, except that it also enforces a visibility condition. While the visibility of a child in both Kconfig and CDL is inherited from its parent in the configuration hierarchy, an explicit visibility condition allows non-parent features to control the visibility too. For example, the visibility of CYGPKG_FS_JFFS2 is controlled by the parent (c-10) and another feature, MTD (c-11).

In Kconfig, the visibility of a feature is controlled by a *prompt* condition. A prompt is a string that follows a type declaration (k-7). It is shown to the user when the feature is visible (the condition is satisfied). The condition is specified after the prompt: here MTD in line k-7. Note that the `select` statement in line k-8 is also conditioned on the same condition as the prompt. This pattern of guarding other constraints by the prompt condition is frequent in Kconfig; thus, the language provides a syntactic sugar for it. The `depends on` keyword adds a condition to the prompt and all other constraints of a feature. For example, the prompt, default, and range specifications of JFFS2_FS_DEBUG are only active if JFFS2_FS is selected, as specified in line k-15. Constraint expressions in Kconfig can use logical operators and equality tests over `bool`, `tristate`, integers and strings.

Range restrictions on integer values are specified using `range` in Kconfig and `legal_values` in CDL (k-17, c-17). The latter can also be used to specify enumerations of values (numbers, strings, or both), such as in c-42. Enumerations are easier to handle for reasoners (such as SAT or CSP solvers) than ranges,

6. Note that eCos developers decided to model this group differently (c-38): with a `data`-flavoured option holding one of three string values encoding the three compression modes.

which tend to have larger domain sizes.

Default values are introduced using the keywords `default` in Kconfig (k-16) and `default_value` in CDL (c-16). If no default value is specified, Kconfig assumes `n` (0) for `bool` and `tristate`, and the empty string for `string`, `int`, and `hex`. In CDL, the assumed defaults for boolean and data values is 0, which is dynamically cast to an empty string if needed.

In Kconfig, visibility conditions, defaults, and configuration constraints interact in intricate ways. If the visibility condition of a feature is false, its default value specification becomes a configuration constraint because the feature cannot be accessed by the user to modify the default value. Such invisible features with calculated values are *derived features*, as defined previously in Section 4.1. JFFS_FS_WRITEBUFFER in line k-22 is derived since it has no prompt declared, thus, its visibility condition is false and its default determines the value. Notice that this feature was not shown in Fig. 1, as FODA notation does not include syntax for invisible, derived features.

An example of a *conditionally-derived* feature is JFFS2_ZLIB, with a stronger visibility condition (`prompt` and `depends on`) than its default condition (just `depends on`). Thus, when the feature is not visible, its value is derived using its default. This happens even if its parent JFFS2_COMPRESS is not selected. Consequently, JFFS2_ZLIB does not establish a child-parent implication, as in feature modeling notations.

CDL clearly separates defaults, which can be overridden by the user and have no configuration semantics, from derived features, which cannot be changed directly by the user. Default values are specified using `default_value` and only take effect when the feature is visible. Invisible features cannot be part of a configuration. Derived features comprise interfaces as well as other feature kinds with the `calculated` keyword, which carry an expression that computes their values (for example Line c-25). A feature can either use `default_value` or `calculated`, but not both. Thus, complex conditionally-derived features do not appear in CDL.

A unique feature of Kconfig is its first-class support for a three-valued logic. Its main operators are defined as follows:

$$\begin{aligned}\text{eval}(!\,e) &= 2 - \text{eval}(e)\\ \text{eval}(e_1 \;\&\&\; e_2) &= \min(\text{eval}(e_1), \text{eval}(e_2))\\ \text{eval}(e_1 \;||\; e_2) &= \max(\text{eval}(e_1), \text{eval}(e_2))\end{aligned}$$

The semantics of expressions follows the logic of Kleene, where `m` corresponds to the unknown state. The equality and inequality test is only defined between features and constants (i.e. `tristate`, `int`, `hex` and `string`). It evaluates to `y` (2) if the values match, and to `n` (0) otherwise.

4.6 Further Concepts

Textual content. Both Kconfig and CDL allow providing natural language descriptions for features (Table 2, row 6): a short text, called `prompt` (k-7) and `display` (c-7), that is displayed to the user to elicit a configuration decision; and a longer description called `help` (k-19) and `description` (c-19) that explains the feature in detail.

Modularization. Modularization allows division of specifications into parts. Kconfig and CDL have modularization capabilities that range from static source inclusion in Kconfig to more complex mechanisms for dynamic loading of packages during configuration in CDL.

Mapping to code. All configs and menuconfigs in Kconfig correspond directly to symbols controlling the build system, and to the preprocessor directives of the same name (see Fig. 4). These symbols and their values are referenced in imperative build logic inside the KBuild system and control the inclusion of particular source files from the Linux codebase.

In Kconfig, it is possible that separately declared features have the same name and, thus, define the same preprocessor symbol. Since all these features share the same state (same value) in the configurator, doing so can lead to intricate interactions of dependencies. In CDL, feature names do not always correspond directly to symbols; instead, a more fine-grained control over symbols is supported, such as suppressing symbols (keyword `no_define`), defining additional ones, or changing their formatting. Line c-18 in Fig. 3 shows an example of a feature defining a build symbol (CONFIG_JFFS2_FS_DEBUG), which actually appears within a preprocessor directive in the code ported from Linux to eCos.

5 The Models

We now turn to the actual models. Our qualitative analysis aims at identifying design criteria that modelers used when creating the models. Our quantitative analysis determines which language concepts of Kconfig and CDL are actually used in the models and how frequently. We report detailed data on all Kconfig models and eCos' i386PC model. Since all eCos models are relatively similar, we only provide aggregated quantitative data for them.

In this section, we first characterize the contents of our models, then discuss their organizational structures, then analyze their constraints, and finally compare them with models in the S.P.L.O.T. repository.

5.1 Content

Our subject models span fairly different domains and are used to configure diverse aspects of the projects. To illustrate their content, we report observations from our qualitative analysis in the first part, and

TABLE 3
Themes of features in the models

Theme	EmbToolkit	Freetz	eCos-i386	BuildRoot	uClibc	uClinux-dist	BusyBox	Fiasco	axTLS	Linux	CoreBoot	uClinux-base	ToyBox
domain-specific	•	•	•	•	•	•	•	•	•	•	•	•	•
diagnostics	•	•	•	•	•	•	•	•	•	•	•		•
build	•	•	•	•	•	•	•	•	•	•	•		
hardware env.	•	•	•	•	•	•		•		•	•	•	
lifecycle	•	•	•	•	•	•	•	•		•	•		
deployment	•	•		•	•	•	•		•				
i18n	•	•	•		•		•			•			
imported	•	•	•	•		•							
ext. library	•	•	•	•		•						•	
software env.		•			•		•	•	•				
test case	•		•						•				

the quantitative analysis in the second part of this subsection.

5.1.1 Feature Themes

Although most features configure domain-specific functional aspects of the systems, we found technical features that are less concerned with functionality, but instead configure the build process, debugging levels, the target hardware environment, and so on. To characterize the model contents, we defined themes of features by manual inspection. We distinguish between domain-specific and multiple technical feature themes:

- **Domain-specific** features are those representing the main content of a model. They describe functional and non-functional concepts within the domain of the host project and belong to none of the following technical themes.
  Most features in the models are domain-specific, for example *networking options* in Linux, *SSL encryption options* in axTLS, or the *JFFS2 filesystem* in eCos.
- **Build** features configure the build process of a system and have no impact on functionality. A sub-theme is *test cases*, which we define separately below.
  Examples of build features are *compilation (CC flags) and linker (LD flags) options*, but also *download sites* in projects that download software packages (such as EmbToolkit).
- **Deployment** features configure the installation process.
  Examples are installation options, such as the *target folder* for axTLS, but also decisions so as to *move files from the firmware image to a USB drive* in Freetz, or to *create symbolic filesystem links to the* BusyBox *executable*.
- **Diagnostics** features aim to provide runtime analysis facilities, such as debugging or profiling.
  Examples are the *BigInt Performance Test feature* in axTLS, a feature enabling *debugging symbols* in BusyBox, or a feature adding *tracing tools* to the Freetz firmware image.
- **Hardware environment** features customize the system to run on a specific hardware, such as CPU, memory, or I/O devices.
  Typical examples are features that determine whether the *processor supports APIC* in Linux, set the *router's flash memory size* in Freetz, or configure *serial ports* in eCos.
- **Lifecycle** features configure explicitly deprecated or experimental functionality.
  Deprecated features are obsolete or not officially supported any more, but often remain in the model for compatibility or dependency reasons. Examples are the *Open Sound System* in Linux, the *msh command* in BusyBox, the *PS/2 keyboard init* in CoreBoot, or hardware architectures whose support is broken in uClibc.
  Experimental features enable functionality in alpha or beta mode, such as *profiling support* in Linux, a *central configure cache file* in BuildRoot, or the *CYG_HAL_STARTUP* feature's value *ROM* in eCos.
- **I18N** features comprise internationalization options.
  Examples are features that select the *firmware language (EN, DE, A-CH)* in Freetz, enable *Unicode support* in BusyBox, or configure *timezone support* in uClibc.
- **Imported** features were copied from other models. They often occur in projects that include other projects with their own variability models.
  For example, EmbToolkit includes both BusyBox and uClibc, therefore, most of their features were copied into EmbToolkit's model.
- **External library** features configure included libraries in the project. Note that if the library has its own model, we classify copied features as *imported*.
  Examples are features that *include certain shared libraries* in Freetz, configure the *EGLIBC library* in EmbToolkit, or select a specific *thread library* in BuildRoot.
- **Software environment** features configure the presence of certain software (libraries or applications) in the target runtime environment.
  Examples are features to select the *target execution platform (Linux, Cygwin or Win32)* in axTLS, to configure whether the *platform has shadow passwords* in uClibc, or to set the *location of existing kernel modules* in BusyBox.
- **Test case** features trigger and configure unit tests during the build process.
  Test cases exist for almost every major component in eCos. Sample features comprise *HTTP server tests*, *POSIX CRC tests*, or *CPU load measurement tests*.

Table 3 shows all themes and their occurrence in each model. The models (columns) are ordered according to the number of feature themes they comprise; and themes (rows) according to the number of models containing them.

While some models contain features of almost every theme, such as EmbToolkit and Freetz, others are very sparse, such as the minimalistic ToyBox model. Nevertheless, all models contain technical features in addition to "ordinary" domain-specific features; mainly to configure diagnostics (debugging) and the build process.

We also observe that many models contain deprecated and experimental features (theme *lifecycle*), both to the same extent. However, no explicit concept for lifecycle features exists in the languages, although we know from experience that many companies need to support such features in their models. Instead, a distinguished feature often switches the visibility of lifecycle features, such as BR2_DEPRECATED ("Show packages that are deprecated or obsolete") in BuildRoot or EXPERIMENTAL ("Prompt for development and/or incomplete code/drivers") in Linux.

5.1.2 Feature Kinds

In Section 4.1, we introduced two classifications for different kinds of features: grouping and individual features, and the role of features. With respect to both classifications, most models are similar, but significant outliers exist.

Grouping and individual features. Recall that the configurator hierarchy shown to the user can deviate from the syntactic hierarchy in the models—in Kconfig due to the nesting of configs based on dependencies,[7] and in CDL due to re-parenting. Thus, we consider two statistics—the syntactic and the configurator grouping of features; the former by counting the grouping features (see feature kinds in Table 2); the latter by counting non-leaf features in the hierarchy shown in the configurators.

The proportion of syntactic grouping features (menus, menuconfigs, and choices) is similar among all Kconfig models, but very low with 3.5%; in contrast, the the proportion for the CDL models is 25% in average. Interestingly, the proportion of configurator grouping features differs significantly from the syntactic grouping in all Kconfig models; indicating that many `configs` are additionally nested in the configurator. This proportion ranges between 11% and 28% (average 19%), except for the two outliers CoreBoot and Freetz with only 4%. In the CDL models, the proportions of syntactic and configurator grouping features only differ by 2%, since some syntactic grouping features are leaves without children. Table 4 shows detailed numbers.

Inspecting the outliers CoreBoot and Freetz reveals different reasons for their low proportion of configurator grouping features. In CoreBoot, large groups exist that contain up to 293 of the leaf features; interestingly, most of these leaves are invisible derived features (mainboard-specific constants). When considering only the visible features in the hierarchy, this proportion is within the normal range (21%) again. In Freetz, the tree is significantly degenerated with one feature having 68% (2377) of all features as children—almost all are leaves. These children represent specific "terminfos" (holding characteristics of Unix consoles) for the ncurses library, which is used to build textual user interfaces. However, these 2377 features are not shown by default due to a visibility condition controlled by the feature "Show all items".

Roles of features. We observe that every model contains user features, implementation features, and derived features. Capability features are difficult to identify, but are certainly contained in one third of the models, see below.

Specifically, the percentage of user features (shown and modifiable by users) is similarly high among almost all models, ranging from 68% to 97%, with the outlier CoreBoot (18%) due to its high degree of derived features (as explained above). For implementation features, we only give upper bounds by counting those features that define a symbol that can be referenced in code (regardless of whether actually used). This upper bound is in average 96.5% for all Kconfig and 81% for all CDL models. The proportion of derived features is rather low, ranging from 1% to 23% (average 13%) among almost all models, but again with the outlier CoreBoot (78%). Capabilities are difficult to identify in Kconfig, since there is no explicit language support. A pattern we found is to prefix such features with HAVE_. Searching for this pattern reveals lower bounds: Linux: 0.8%, CoreBoot: 0.6%, uClibc: 0.8%, and none in any other Kconfig model. CDL has an explicit capability concept (interfaces); in average, 11% of the features in the CDL models are capabilities.

5.1.3 Feature Representation

While switch features are the most frequent type, every model except the minimalistic ToyBox also contains features with data values—numbers or strings. Their proportions are rather low (0-11%) compared to switch features; yet, this observation calls for adequate language and tool support, especially with regard to constraints. Fig. 5 and Table 5 show the breakdown of features by type.[8]

Surprisingly, in a quarter of the Kconfig and all CDL models, we find relatively high proportions of data

7. The hierarchy induced by menus and menuconfigs is shown in the left window of the configurator and requires explicit drill-down by the user. Config hierarchies are shown by indentation in the right window and are, thus, more lightweight to navigate.

8. Note that the eCos-i386 percentages do not add up, since features can be both switch and data (type booldata) in CDL.

TABLE 4
Grouping statistics

		Linux	axTLS	BuildRoot	BusyBox	CoreBoot	EmbToolkit	Fiasco	Freetz	ToyBox	uClibc	uClinux-base	uClinux-dist
grouping	syntactic[2]	4%	12%	5%	3%	1%	2%	4%	1%	4%	3%	1%	2%
	configurator[4]	16%	28%	13%	22%	4%	20%	11%	4%	21%	15%	20%	15%
	difference	12%	16%	8%	19%	4%	18%	7%	2%	17%	12%	19%	13%
grouping with constraints	XOR	1%	5%	4%	1%	3%	8%	6%	0.4%	0%	6%	19%	1%
	MUTEX	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
	runtime XOR[5]	0.03%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
	sum	1%	5%	4%	1%	3%	8%	6%	0.4%	0%	6%	19%	1%
	model size[6]	6,320	108	1,938	881	2,269	1,357	171	3,471	71	369	383	1,620

	eCos-i386	eCos-all[1]
syntactic[3]	26%	25%
configurator[4]	24%	23%
difference	2%	2%
XOR	1%	1%
MUTEX	0.1%	0.1%
OR	0%	0%
sum	1%	1%
model size[6]	1,256	1,247

[1] mean [2] menuconfig, menu, optional tristate choice [3] component, package [4] features with children [5] mandatory tristate choice [6] number of features

features, in particular 27% in axTLS and more than 50% in CoreBoot and the CDL models. This observation is interesting, since the majority of examples found in the literature has few or no such features [32]. Further, Linux frequently uses the three-state logics for controlling binding mode; more than half of the features are of the tristate type. However, since support for loadable kernel modules is unique to Linux, no other model has tristate features.

Supporting number (int, hex, float) and string data features appears to be equally important in most models; their proportions are similar, but slightly tending to string features. Only Linux and CoreBoot have significantly more number than string features.

Usage of Data Features. Inspecting the models with high proportions of data features—axTLS, CoreBoot, and eCos-i386—shows that data features are used for diverse purposes.

In axTLS, data features configure the built-in web-server (e.g. port, ssl expiry time, folders), paths to external libraries (e.g. Java, Perl), or SSL certificate details (e.g. common name, organization name). However, the high percentage of data features might be biased by the rather small model.

In CoreBoot, almost every data feature (98%) is derived, invisible, and represents a constant (e.g. number of IRQ slots, mainboard-specific source folders). These constants exist for each of the 166 mainbords supported (as explained previously).

In eCos-i386, some feature kinds contain data values by default: interfaces always carry a number (count of implementing features that are enabled), and packages always have the flavor `booldata`, with the data part representing the package version as a string. 15% of eCos-i386's features belong into this category. Further, 2% of features represent enumerations. There are also 6% of features representing compiler flags, 0.3% linker flags, and 3% holding names of files with test code. The remaining data features (28% of all features) represent diverse configuration constants, such as priorities, buffer sizes, and supported I/O ports. Apparently, many of these constants are specific to an embedded operating system and would either be set dynamically or not be configurable in a system like the Linux kernel.

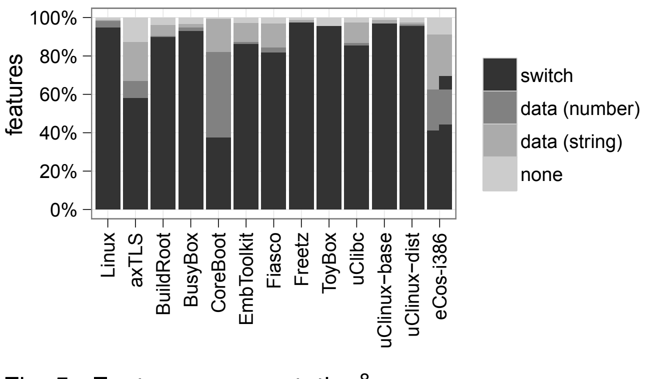


Fig. 5. Feature representation[8]

5.2 Organization and Hierarchy

This section describes the organizational structure and summarizes characteristics of the feature hierarchies found in the models. The first part reports qualitative observations and aims at understanding how the systems are decomposed into features. The second describes quantitative measures of the configurator hierarchies, aiming at providing useful assumptions for tools in order to effectively visualize models.

5.2.1 Organizational Structures

A qualitative analysis of how features are organized in the models shows that projects use different strategies to group features. Such strategies vary not only from project to project, but also within a project. For example, as we will show, some features are grouped together by their functionality, such as networking and filesystem features; others are grouped by the mechanism by which variability is realized, such as features that are applied as patches or compiler flags.

We describe model composition strategies by showing how each project organizes features. We start with Freetz as it uses many different strategies, we

TABLE 5
Feature representation statistics

		Linux	axTLS	BuildRoot	BusyBox	CoreBoot	EmbToolkit	Fiasco	Freetz	ToyBox	uClibc	uClinux-base	uClinux-dist
switch	bool	36%	56%	87%	92%	36%	79%	77%	97%	96%	80%	79%	95%
	tristate	58%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
	sum	94%	56%	87%	92%	36%	79%	77%	97%	96%	80%	79%	95%
data	number[3]	3%	8%	1%	2%	43%	1%	2%	0.1%	0%	1%	0%	1%
	string[4]	0.4%	19%	6%	2%	17%	9%	12%	1%	0%	10%	1%	1%
	sum	4%	28%	6%	4%	60%	10%	14%	1%	0%	11%	1%	2%
none	menu	1%	12%	4%	3%	1%	2%	3%	1%	4%	2%	1%	2%
	choice	1%	5%	4%	1%	3%	8%	6%	0.4%	0%	6%	19%	1%
	sum	2%	17%	7%	4%	4%	11%	9%	2%	4%	8%	20%	3%
	model size[6]	6,320	108	1,938	881	2,269	1,357	171	3,471	71	369	383	1,620

	eCos-i386	eCos-all[1]
bool	37%	37%
booldata[2]	15%	15%
sum	52%	52%
number[2,5]	22%	24%
string[2,5]	29%	29%
sum	50%	53%
none	9%	9%
model size[6]	1,256	1,247

[1] mean [2] switch and data features overlap (type booldata) [3] type int, hex [4] type string [5] dynamic type, identified with heuristics [6] number of features

then proceed with the remaining Kconfig projects, and finally describe the organization of the Linux kernel and the eCos operating system. Summaries of the Freetz, Linux, and eCos-i386 feature hierarchies are shown in Fig. 6a, Fig. 6b, and the left-hand side of Fig. 7. Each box represents a grouping feature labeled by the feature name, the number of its descendants (excluding descendants of the sub-groups that are already shown in the figure), and a label[9] indicating the theme of the group according to Section 5.1.1 and Table 3, if applicable. The height of each box indicates the number of features within the group.

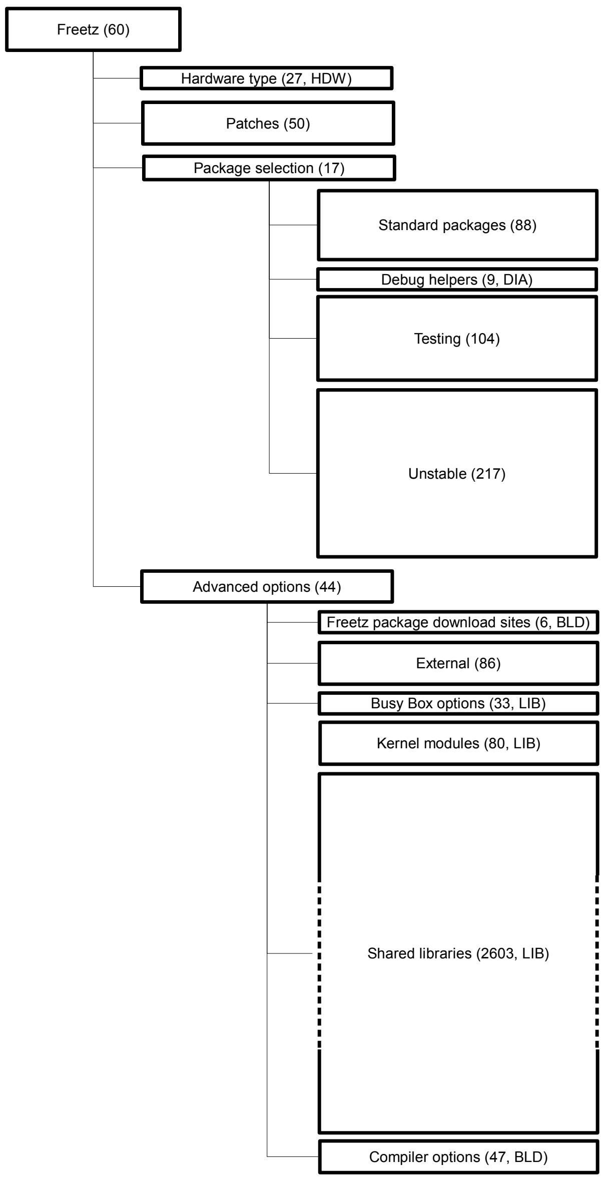
Freetz. The Freetz model—summarized in Fig. 6a—is a prime example of a project that uses different strategies to group features. The "hardware type" group allows detailed configuration of the hardware, such as WLAN version. These features are grouped according to the *hardware env.* theme from Table 3. The "patches" group contains features that are applied as patches to the code to change the system by removing branding, help, altering storage names, and so on. The strategy used for this group is the variability mechanism by which such features are realized. The "package selection" group contains options to include certain utility packages, such as curl (a command line tool for transferring data) and inetd (a server daemon for internet services). In general, this group contains diverse features that all configure functionality (theme *domain-specific*). It contains the groups "standard packages", "web interface", "debug helpers", "testing", and "unstable". The features in "debug helpers" are grouped by the *diagnostics* theme, and the "testing" and "unstable" packages by the *lifecycle* theme. Lastly, the "advanced options" group contains a large number of configuration options that can be used to: configure package download sites; add external processing features (e.g. IP anonymizer and bittorrent server); configure BusyBox; add modules from the Linux kernel; add cryptography, compression, and other shared libraries; and to set compiler options.

In summary, Freetz uses a variety of strategies to organize features. A common strategy that Freetz uses is to group features by one of the themes from Section 5.1.1 and Table 3. Some strategies, however, follow an even more specific theme, such as the package download sites feature, which can be considered a sub-theme of the *build* theme. Finally, some strategies, such as compiler options and patches, are cross-cutting across themes. For example, the features "Remove dtrace" and "Remove ftpd" are features located in the "patches" group; these features apply patches to the original firmware to remove existing functionality, mainly to save space. However, as dtrace and ftpd are external libraries, these features also belong to the *external library* theme.

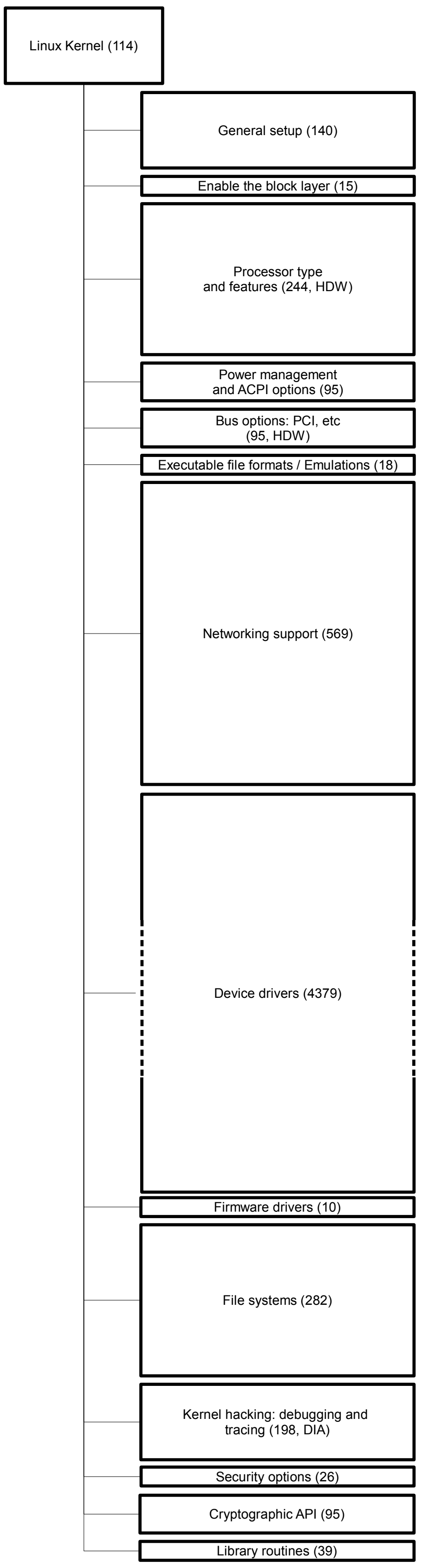
BuildRoot, EmbToolkit, uClinux. These projects group features by hardware architecture (theme *hardware environment*) and by the root file system (theme *domain-specific*). The choice of architecture affects values of architecture-dependent features using defaults and visibility conditions. Unlike the other two projects, uCLinux separates the configuration of architecture and root file system through staged configuration. A configurator is initially launched for the architecture selection. Depending on the choices made in the first configurator, a different default configuration for the root file system is used.

axTLS, CoreBoot, uClibc. These projects use the same strategy as above where an architecture choice (theme *hardware environment*) affects the choices of architecture-specific features (theme *domain-specific*). In axTLS, the architecture is a platform choice (e.g. Linux, Cygwin or Win32); CoreBoot's architectures are motherboards (e.g. AMD or Intel); and uClibc has processor architectures (e.g. Alpha, ARM or i386). Interestingly, CoreBoot extensively uses multiple declarations of a single feature to define mainboard-specific constants (as

9. BLD=Build, DIA=Diagnostics, HDW=Hardware environment, LIB=External library



(a) Freetz


Linux Kernel (114)
General setup (140)
Enable the block layer (15)
Processor type and features (244, HDW)
Power management and ACPI options (95)
Bus options: PCI, etc (95, HDW)
Executable file formats / Emulations (18)
Networking support (569)
Device drivers (4379)
Firmware drivers (10)
File systems (282)
Kernel hacking: debugging and tracing (198, DIA)
Security options (26)
Cryptographic API (95)
Library routines (39)

(b) Linux

Fig. 6. Summarized Freetz and Linux hierarchies

pointed out previously in Section 5.1.2). For example, the BOARD_SPECIFIC_OPTIONS feature is declared 142 times. The model is modularized such that each motherboard is declared in its own Kconfig file. CoreBoot relies on the configurator merging the multiple declarations into a single feature.

Fiasco. Fiasco's hierarchy only has four top level groups. Like the previous projects, it starts with a group of hardware features (theme *hardware environment*) comprising architecture (Intel, AMD64, ARM), platform (PC or Linux usermode), CPU, and more detailed options that all affect derived invisible constants used in the remainder of the model. This group is the largest in the model with 105 features. Thereafter, a group of only 16 features configures the functionality of the kernel (theme *domain-specific*); while the third group comprises debugging (theme *diagnostics*; and the fourth compiler options (theme *build*).

BusyBox, Toybox. These two projects separate their features into two groups: build-related features that affect compilation (theme *build*), and by the shell commands (theme *domain-specific*). ToyBox has two top-level menus for these groups. BusyBox, being the larger project, further groups the shell commands into sub-categories, such as archival, console, or networking.

Linux. Although, as we will explain later, the Linux model hierarchy has a depth of 8, we found that, for the purpose of describing the overall organization, it is sufficient to present only the top hierarchy level, as shown in Fig. 6b. Differently from Freetz, Linux' top level of groups is already quite specific.

Similar to Freetz, top level groups are about core hardware configuration: "General setup", "Enable the block layer", "Processor type and features", "Power management and ACPI options", and "Bus options: PCI, etc". The remaining groups—except "Kernel hacking", "Security options", and "Library routines"—are for configuration of different functionality (e.g. networking, file systems, and cryptography), devices, and architectural components. "Kernel hacking", "Security options" and "Library routines" are groups of features that cross-cut functionality and architectural component groups. While "Security options" and "Library routines" are grouped by the *domain-specific* theme, the "Kernel hacking" features are grouped for their common *diagnostics* theme.

We conclude that Linux' main strategy for grouping features is their common functionality, architectural component or hardware. While experimental or deprecated features (theme *lifecycle*), for example, could also be grouped together, Linux gives priority to grouping them by functionality or by their architectural components. Alternatively in Freetz, *lifecycle*-themed features are put into separate groups, such as testing and unstable, but these cross-cut the functionality groups. Although Linux does not place *lifecycle*-themed features into groups, they are tagged with a dependency on the EXPERIMENTAL feature. This allows their visibility to be toggled by enabling or disabling EXPERIMENTAL. Examples are the features "User namespace" in the "Namespaces support" group and "PCI Express ASPM support" in the "PCI Express support" group. Both are only visible when EXPERIMENTAL is selected.

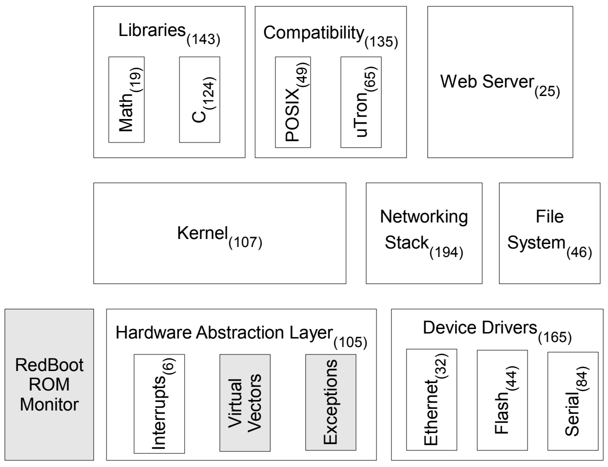


Fig. 8. eCos architecture, adapted from [33]. Shaded architectural concerns could not be mapped to grouping features in the model.

eCos-i386. In eCos, the variability model is aggregated from smaller models that are distributed over the 500 packages in the codebase. Each package forms a subtree with a feature of kind package at its root. By default, all these subtrees become children of the synthetic root of the aggregated model, except for re-parented features. We find two common use cases for re-parenting: The first use case is to place global build options under a top-level *component* with this name. The other use case is to place packages into the subtree of other packages. For example, many core hardware-specific packages are re-parented into the "eCos HAL" (Hardware Abstraction Layer) package.

The organization of the model can be characterized as follows. The first child of the synthetic root node is "Global Build Options" containing the aforementioned re-parented, build-specific features from several packages. The next child is the package "eCos HAL" with hardware-specific options, into which other hardware packages are mounted, such as the many i386-specific packages. If the user selects another target (hardware architecture) in the configurator, other packages would be mounted into this HAL subtree. Thereafter, the packages for the I/O subsystem and several rather technical packages appear, such as the configuration of the eCos kernel or various C libraries, such as libc, libm (math) or snmplib. The rest of these top-level packages comprise more domain-specific functionality, such as networking, clients and servers, but also the filesystems supported in the final eCos instance.

In summary, eCos-i386 features are grouped largely

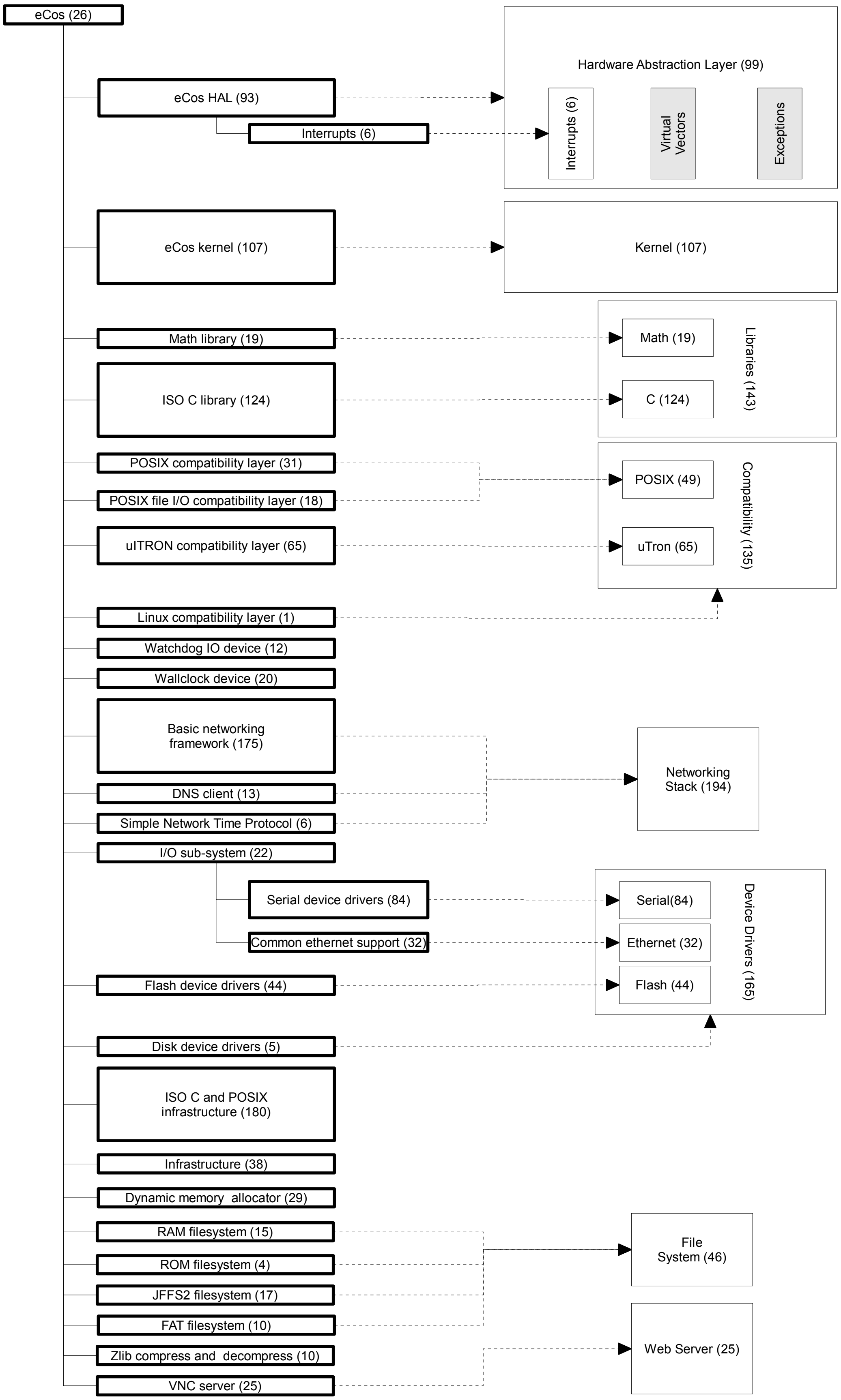


Fig. 7. Summarized eCos-i386 model hierarchy. 64% of the features can be mapped to architectural concerns.

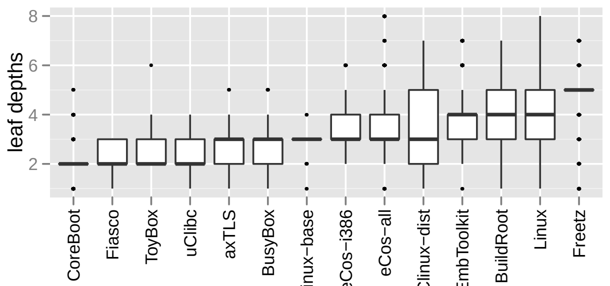


Fig. 9. Depth of leaf features in the models.

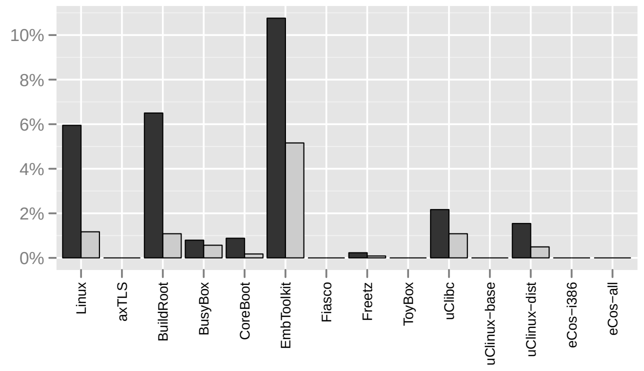


Fig. 11. Proportion of features violating hierarchy rules. The black bar represents features not implying their parent; the gray bar the unique parents that have children not implying them.

by having a common architectural component. Fig. 7 shows the eCos-i386 model with links from the groups to the architectural concern for whose configuration the group is responsible for. These concerns were extracted from an eCos book [33], which we reproduce in Fig. 8. For Linux, we did not find such a clear mapping from feature groups to architectural concerns, using the interactive Linux kernel map[10] as a reference. Linux has a much more fine-grained and complex architecture and feature model. For example, although the Linux architecture has a networking component, it is subdivided into "socket access", "protocol families", "protocols", "virtual network device", and "network device drivers". Such subdivisions are not explicit in the Linux model.

5.2.2 Model Hierarchies

A quantitative analysis of the configurator hierarchy shows that all models are wide and shallow. Their average depth ranges between 3 and 4 (shallow outlier CoreBoot with 2). The maximal depth is as low as 4 for uClibc and uClinux-base and not more than 8 for the huge Linux model; see the leaf-depth distributions in Fig. 9. At the same time, branching factors (number of children per feature) vary to a great extent in our models, which is in contrast to nicely-balanced trees in the literature. Although the vast majority of features (between 72%–96%; 84% in average for Kconfig and 77% for CDL models) are leaves, we observe many features with more than 100 children. Practically, none these models could be rendered as a tree structure like in Fig. 1, which is the common visualization in literature. For illustration, we provide plots of the three smallest models ToyBox (Fig. 17), axTLS (Fig. 18), and Fiasco (Fig. 19) in the Appendix. With their scaling, the Linux model would only be a flat line.

Further analysis shows that in all models, the number of features with a given number of children decreases sharply with the increase of the number of children. Fig. 10 shows histograms of branching factors in the models. It excludes leaves, which represent the majority of features in the models. The second-largest class are single-child parents (7% average in Kconfig, 6% in CDL models), followed by two-child parents (3% average in Kconfig, 5% in CDL models). Note that single-child features increase the proportion of inner to leaf features. Features with more than ten children are very seldom; nevertheless, the maximum number of children is as much as 157 in Linux and 36 in one of the CDL models (ipaq). The median of maximum branching is 85 in the Kconfig, and 33 in the CDL models; however, we find outliers with 173 (uClinux-dist), 293 (CoreBoot), and whopping 2377 (Freetz) child features.

Relatively few features violate hierarchy rules—child-to-parent implications—of feature modeling. Thus, we believe that practitioners find hierarchical organization of dependencies natural. Recall that, unlike in feature modeling and CDL, Kconfig uses hierarchy to depict a visibility relation instead of a presence condition, allowing a child feature to be configured without its parent. This possibility is indeed exploited in the Linux model. Sometimes, children even exclude their parent. We verified with a SAT solver applied to a derived boolean semantics of the Kconfig models [22] that all models except axTLS, Fiasco, and ToyBox contain features not implying their parents in the configurator hierarchy. Fig. 11 shows these proportions among all models. A nice example from the Linux model is the conditionally-derived feature JFFS2_ZLIB in Fig. 3 (Line k-32), which is automatically selected if the parent is not, as we explained in Section 4.5.

In CDL, all features in the configurator hierarchy imply their parent. However, by manual inspection, we found 39 (3%) re-parented features in eCos-i386, which do not imply their syntactic parent anymore. Most re-parentings move packages in the hierarchy, but 10 options and 2 components were re-parented as well. For example, the GLOBAL_OPTIONS component from HAL_I386_PC package was promoted to the top-

10. http://makelinux.net/kernel_map

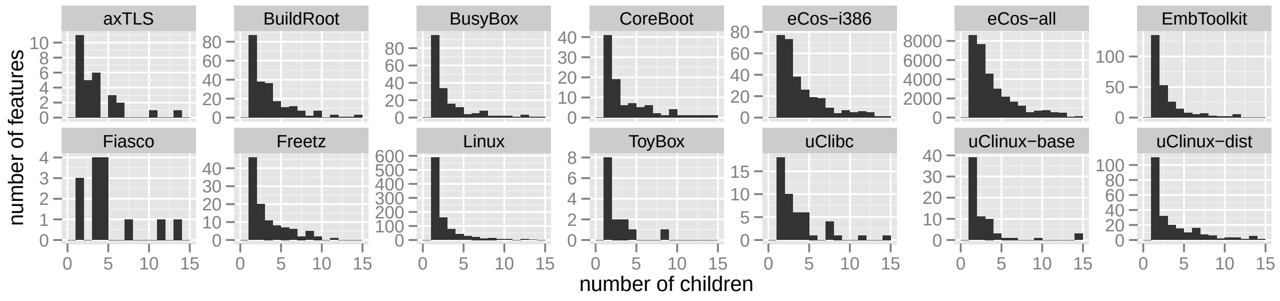


Fig. 10. Branching factors, excluding leaves and x-axis cut off at 15. The eCos-all diagram aggregates features from all CDL models.

level and, in addition to its syntactic children, two new options were re-parented under this component.

5.3 Constraints

In addition to the constraints residing in the hierarchy (child-parent implications), each model has additional constraints declared over features. This section reports observations about group constraints and the various types of feature constraints.

5.3.1 Group Constraints

Feature groups (Table 2, *group constraints*) are among the core concepts in feature modeling. In fact, groups are regularly used in all of our models. But surprisingly, OR groups—commonly mentioned in literature—are not supported by the Kconfig language and only occur in three CDL models. Kconfig's slightly similar grouping concept—runtime XOR (cf. Section 4.4)—appears only twice in Linux. The most frequent type of group constraints is XOR, which is contained in every model except ToyBox. Table 4 (*grouping with constraints*) shows detailed numbers.

In Linux and all CDL models, at most 1% of the features impose group constraints on their children. The other models have higher percentages. Among all Kconfig models, the average is 4%, whereas outliers are EmbToolkit with 8% and uClinux-base with even 19% of features representing XOR groups. MUTEX groups are very rare—only the CDL models have one each, except for one model with two MUTEX groups.

The insignificance of OR and MUTEX groups is surprising. We speculate that both are realized separately with constraints, such as dependencies to a capability. Unfortunately, this is difficult to measure due to a lack of a syntactic construct for capabilities in Kconfig (cf. Section 4.1).

Let us see how group constraints are used in practice. The two runtime XOR groups in Linux are motivated by binding time: this constraint allows including multiple alternative features in the configured kernel as dynamically-loadable modules; only one of them will be loaded at runtime. The only MUTEX group in eCos-i386 represents three alternative random number generators. A possible reason for the lack of MUTEX groups in Kconfig models is the need to define a build symbol even when no group member is selected, see for example the feature JFFS_CMODE_NONE in Fig. 3.

Recall that CDL interfaces generalize group cardinality constraints. This generality is not exploited in practice though. There is no cardinality constraint that is a proper (m, n)-interval, as opposed to intervals with lower bound of 0 or 1 and upper bound 1 or *. Moreover, although an interface can place a group constraint on features that are not siblings, all interfaces are implemented by sibling features. Still, interfaces and implementing features are usually far apart, that is, do not have a common parent and are implemented across different packages. In other words, the group constraint is activated (implied) by the parent of the interface, which is not the parent of the set of constrained features. This form of a group constraint is more general than what is found in feature modeling, where the parent of the group activates the group constraint. Such generalized group constraints are used to model the case where a given package defines an interface required by its implementation and multiple other packages provide alternative implementations of that interface. This case is relatively frequent, 81 interfaces are constrained this way in the eCos-i386 model.

5.3.2 Feature Constraints

All models declare additional configuration, default, and visibility constraints for their features. In the following, we discuss the frequency and usage of the various types of feature constraints, and the number of cross-tree dependencies per feature. The latter is defined as the reference of another feature in a constraint. Our observations are supported by Table 6, which shows the percentage of features declaring a certain type of constraint, and Fig. 12 and Fig. 13, which show dependencies and their growth.

Types of Constraints. The vast majority of features, in average 77% in the Kconfig and 86% in the CDL

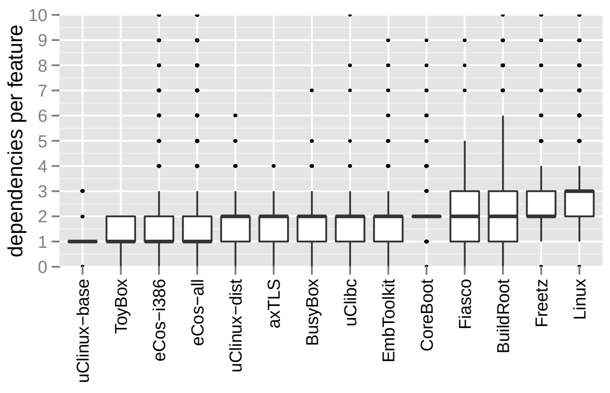


Fig. 12. Dependencies per feature, including dependency on parent feature. The y-axis is cut-off at 10.

models declare constraints of some sort[11] (configuration, visibility, default), as can be seen in Table 6. In the following, we explore the usage and quantity of different types of constraints.

Derived features are mostly used to perform calculations that otherwise would be hidden in the build system. This way, feature dependencies are specified uniformly and explicitly in the model. Recall that Linux supports conditionally-derived features, which are derived or user-changeable with a default value, depending on a condition. However, only 1,5% (mean) of the features in the Kconfig models (max 3% in Linux) belong into this category. The proportion of unconditionally derived features is higher, with 13% in average for Kconfig, and 7% in CDL models (cf. Section 5.1.2).

Visibility control is very frequently used in the models. All except ToyBox declare explicit visibility conditions. In the Kconfig models, in average 6% of the features have an explicitly-specified prompt condition (like JFFS2_ZLIB in Fig. 3, Line k-32), rather than just via `depends on`, and 10% of features in the CDL models use `active_if`.

Default values (also computed) are used frequently in the models; all except uClinux-base declare explicit defaults. However, their proportions differ significantly: Half of the Kconfig models have low (<20%) percentages of features with explicit defaults, such as Linux (16%). The others make heavy use of explicit defaults for at least half of the features (uClibc, uClinux-dist, and axTLS), or significantly more: 69% in the CDL models, and at least 90% in ToyBox, BusyBox, and Freetz. Only three Kconfig models compute (via expression) their defaults: 4% of features in Linux, and only one feature in both BuildRoot and uClibc. However, in all CDL models, between 6–11% of the features are computed. All other defaults are specified with literals.

Dependencies. To characterize dependencies, we discuss two metrics: the number of features referenced in constraints of a feature, and the Cross Tree Constraints Ratio (CTCR), a metric adapted from [34].

In average, 60% of the features in the Kconfig and 35% in the CDL models have dependencies to other features across the hierarchy (cross-tree constraint). The highest can be seen in the Linux model, where surprising 85% of the features reference other features.

The average number of features referenced per feature (excluding parent implication) among all features is 1.4 in the Kconfig, and 0.6 in the CDL models. In Linux[12], most features refer to 1–3 other features (maximum of 52); this range is much lower in the CDL models, with typically 0–1 cross-tree dependencies (maximum of 20). Some features declare a large number of cross-tree dependencies; up to 127 in BuildRoot and 101 in EmbToolkit. We visualize the number of dependencies per feature as a boxplot in Fig. 12, which includes the hierarchy dependencies (+1).

Interestingly, the average number of dependencies per feature seems to grow linearly with the size of the models and does not explode, as can be seen in Fig. 13. This finding is in line with our observations from a study of the Linux model evolution [35]. It indicates that feature models abstract over implementation dependencies, since code dependencies usually grow quadratically [36].

Table 6 also provides an adapted version of the CTCR metric [34] (ECR in [37], [38]) for all models. It provides the percentage of features that have a dependency or are the dependency target of another feature—put simply, the proportion of features participating in cross-tree constraints.

Examples. Let us look at some examples of constraints. Linux constraints are mostly logical expressions, such as a single feature or more complex expressions, e.g.,

```
SMP && (X86_32 && !X86_VOYAGER || X86_64)
```

11. In [21], we discounted unconditionally and conditionally derived features in the Linux model.

12. In [21], numbers included dependency on parent.

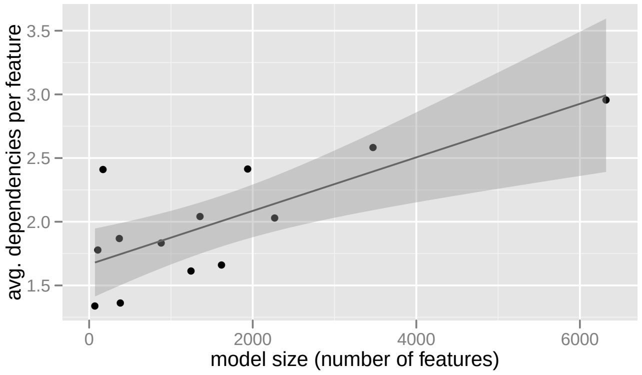


Fig. 13. Mean feature dependencies per model (CDL models combined)

TABLE 6
Percentage of features with constraints and CTCR metric

	Linux	axTLS	BuildRoot	BusyBox	CoreBoot	EmbToolkit	Fiasco	Freetz	ToyBox	uClibc	uClinux-base	uClinux-dist	eCos-i386	eCos-all[1]
any constraint	92%	77%	60%	95%	86%	61%	85%	98%	96%	75%	19%	74%	86%	86%
configuration constraint	88%	41%	48%	64%	82%	48%	77%	92%	37%	51%	1%	53%	38%	38%
value restriction	1%	0%	0%	2%	0%	0.1%	1%	0%	0%	1%	0%	1%	11%	11%
unconditionally derived	12%	1%	2%	1%	78%	17%	23%	1%	6%	18%	1%	1%	7%	7%
conditionally derived	3%	0%	0.4%	0%	0.4%	13%	0%	1%	0%	0.3%	0%	0.2%	N/A	N/A
visibility condition	5%	10%	5%	1%	3%	22%	3%	1%	0%	5%	18%	2%	10%	10%
explicit default	16%	59%	12%	92%	4%	10%	11%	94%	90%	41%	0%	46%	69%	69%
expression (computed)	1%	0%	0.1%	0%	0.3%	0%	1%	0%	0%	0.3%	0%	0%	7%	7%
literal	15%	59%	12%	92%	3%	10%	10%	94%	90%	41%	0%	46%	62%	62%
CTCR metric[2]	93%	70%	71%	79%	96%	88%	87%	96%	46%	75%	55%	68%	48%	49%
model size[3]	6,320	108	1,938	881	2,269	1,357	171	3,471	71	369	383	1,620	1,256	1,247

[1] mean [2] Cross Tree Constraints Ratio (percentage of features participating in cross-tree constraints) [3] number of features

Linux constraints often reference integer or string features using equality tests. In a single case, an integer feature in Linux uses another feature as a bound in a range constraint.

Many eCos constraints are logical expressions too, but arithmetic and string operations are not uncommon. For example,

```
requires { CYGNUM_FS_FAT_NODE_POOL_SIZE >=
    ( CYGNUM_FILEIO_NFILE + 2 ) }
```

String concatenation (denoted by ".") is often used to produce lists of test or implementation source files:

```
calculated {"tests/sprintf1 tests/sprintf2 " .
    ((FILEIO && RAM) ? "tests/fileio" : "")}
```

Other constraints check whether a particular file name is included in a list; e.g. **requires** `is_substr(LIBS, "libtarget.a")`. Such constraints implement code mappings. In Linux, these are computed in KBuild [39], outside the model.

5.4 Comparison with Available Feature Models

We now compare the properties of our models against the models available in the popular S.P.L.O.T. repository [15], which currently hosts 264 feature models extracted from research literature, or donated by visitors. These models originate from various domains, such as insurance, entertainment, and home automation. SPL2go [16] is another online repository, which contains example product lines including their feature models. Presently, SPL2Go hosts 32 models, which are on average smaller then those in S.P.L.O.T., so we focus on the latter.

S.P.L.O.T. models are expressed in their own feature modeling language, which offers a subset of FODA concepts (see Table 2). Naturally, we limit the comparison to these concepts: model size, model shape (leaf-depth and branching), feature groups (OR, XOR, and MUTEX), and constraints (CTCR metric). Figures 14–16 show the distribution of these metrics for our sets of S.P.L.O.T., Kconfig, and CDL models.

The model sizes (Fig. 14) differ significantly. The S.P.L.O.T. models are mostly small, with sizes ranging

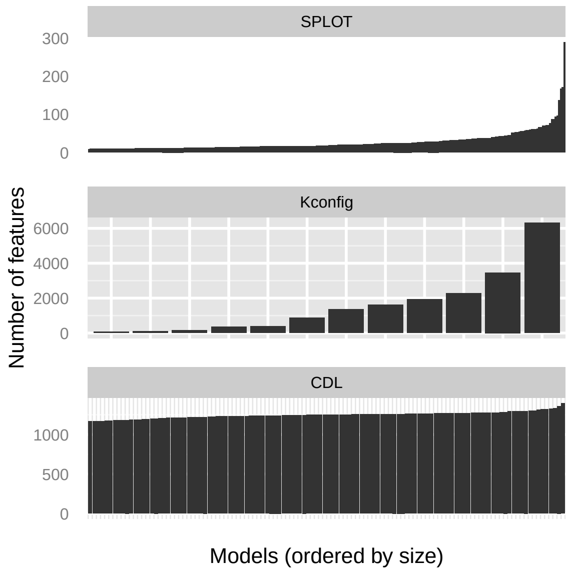


Fig. 14. Sizes of S.P.L.O.T., Kconfig, and CDL models

from only 9 to 290 features. Only 25% of the models have more than 31 features, and only four more than 100 features. The average model size amounts to 18 (median 27) features, which shows that our Kconfig and CDL models are significantly larger.

S.P.L.O.T. models are relatively deeper than Kconfig and CDL models (Fig. 15, left diagram). Given the substantial differences in the model sizes, it is significant that the average depth of leaf features among the S.P.L.O.T. models is 2.7, while the Kconfig leaves (average depth of 3.7) and the CDL leaves (average depth 3.3) are only slightly deeper. Prime examples are the Electronic Shopping and the Thread model. The Electronic Shopping model (size 290 features) is the model with most features (70 in fact) with a depth greater than 5. In this model, the average leaf depth is 5, with a maximum depth of 9. The Thread model (size 44 features) has the deepest hierarchy, with leaves up to a depth of 10.

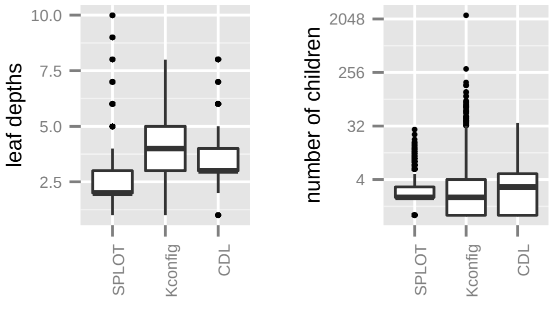
Fig. 15 (right diagram) shows the model branching factors on a log_2 scale. The Kconfig and CDL models have a similar median, but higher variations in branching then the S.P.L.O.T. models. The maximal branching there is 28—a feature aggregating test cases in the Billing model (size 88 features). The median of maximum branching among the S.P.L.O.T. models is only 5, in contrast to 85 for the Kconfig and 33 for the CDL models (cf. Section 5.2.2).

Comparing the constraints shows that the S.P.L.O.T. models typically have more feature groups than the Kconfig and CDL models—there is an average of 12.5% feature groups among a model's features (Figure 16a). In contrast, S.P.L.O.T. models have a significantly lower cross-tree constraint ratio (CTCR metric) than our models (Figure 16b).

6 The Configurators

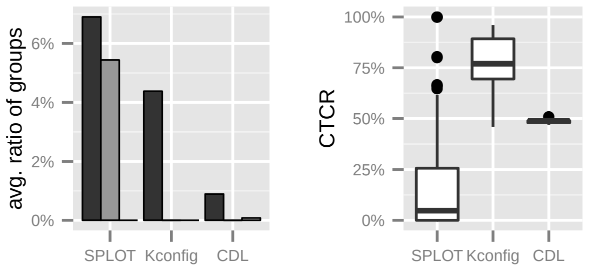
Kconfig and CDL are equipped with GUI-based configurators that both support a configuration process known as *reconfiguration*: The tool is initialized with a configuration loaded from a file, or based on default values, which is modified by the user to reach a desired state. Each of the two configurators takes a different approach to ensure that the user retains a valid configuration. The Kconfig configurator prevents the user from modifications that violate constraints; the CDL configurator allows such modifications, but it detects violations and helps in resolving them.

The Kconfig configurator offers little support for propagating user configuration choices. If the dependencies of a given feature are not satisfied, the tool prohibits selecting it. The user has to find out which other features need to be reconfigured to enable the selection. A rudimentary propagation support is offered by the `select` construct; it enforces a selection of a single feature, when the feature hosting the statement is selected. The selection is made without respecting any constraints. This imperative behaviour can lead to illegal configurations and requires Kconfig developers to explicitly specify any transitive dependencies to maintain consistency. For example, LATENCY_TOP contains selects for both KALLSYM and KALLSYM_ALL. KALLSYM_ALL depends on KALLSYM, thus, the sole selection of KALLSYM_ALL would be sufficient if the configurator used a propagating reasoner. In fact, the official documentation and the Linux kernel commit log contain multiple warnings and complaints about the error-proneness of using this construct [35]. Still, the Linux model is full of `select` statements, as this is the only way to obtain (limited) propagation in the configurator.



(a) leaf-depth (b) branching factor

Fig. 15. Shapes of S.P.L.O.T., Kconfig, and CDL models



(a) Average ratio of feature groups; three bars each: XOR (first), OR (second), MUTEX (third) (b) CTCR metric

Fig. 16. Feature groups and cross-tree constraint ratios in S.P.L.O.T., Kconfig, and CDL models

The CDL configurator is far more intelligent than its Kconfig counterpart. When the user modifies a configuration, the tool detects all constraint violations and offers the user support to resolve them via an inference engine.

This engine works as follows. Every change to the model is wrapped in a transaction and the configurator checks for any constraint violation. If one occurs, the inference engine tries to resolve the conflict by a heuristics-based recursive search algorithm. It builds a tree of transactions, starting a transaction for each new sub-conflict that arises when testing conflict resolutions. The engine estimates the benefit of particular (sub-)conflict resolutions by using the number of required changes and source of the values being changed, that is, user, default or inference. If a sub-resolution is beneficial, it gets committed to the parent transaction. If one overall solution is found for the top-level conflict, the tool lists necessary changes and requests confirmation. Otherwise, the conflict requires manual resolution.

We investigated the inference engine's source code with respect to *correctness* and *completeness*. The resolution is *correct*, since the proposed resolutions are verified against the model constraints. The resolution is *incomplete* as:

- The inference rules are incomplete. For example, the engine has rules for handling cardinality constraints on interfaces of 0 or 1, but not for arbitrary bounds.

- The recursion depth is limited to three levels; thus, reasoning on transitive *requires* dependencies is incomplete.
- The engine uses a greedy search, evaluating resolutions to sub-conflicts in separation and pruning all but the optimal one. This may prune all successful branches.

Although the inference engine is less powerful than general CSP solvers, it performs very well on the actual eCos model. The support for MUTEX and XOR groups is particularly effective and the resolution of `requires` dependencies is far more maintainable than the `select` statement in Kconfig.

The main limitation of the CDL configurator is that if several resolutions exist, it finds at most one and possibly not the desired one. The following comment[13] on the mailing list indicates that developers struggle with this problem:

> *[...] if* CYGPKG_MYPKG_OP1 *is active, make sure that the list of tests for that package is a substring of* CYGDAT_MYPKG_ACTIVE_TESTS*. This works 50% of the time. Problem is the other 50% of the time, rather than fiddling with the substrings, it enables / disables my subpackage!*

Our findings underscore the importance of building configurators based on strong reasoners. Both the Kconfig and the CDL configurator support only a small subset of analyses introduced by researchers. A recent literature review [1] discovered 30 analysis operators on feature models that have been addressed in research, mainly for simple languages. Among the 30 operators, the Kconfig configurator only supports the "valid product" operator, whereas the CDL configurator implements the "valid product", "corrective explanations" (conflict resolution), "dependency analysis" (complete a partial configuration) operators. In addition, our developed infrastructure enables many SAT-based analysis operators, such as finding "void products", "dead features", or "core features".

7 Lessons Learned

We now summarize the main lessons learned.

7.1 Modeling Concepts

Essentially all core FODA concepts are used in all models, that is, Boolean (switch), int and string features, a hierarchy, group and feature (cross-tree) constraints.

Feature types. Boolean (switch) features are the most common type. Data features (numbers and strings) appear to be typical for systems software—the CDL models often use arithmetic operators and comparisons. Strings are primarily used for file names.

Hierarchy. The hierarchy shown to users in the Kconfig and CDL configurators largely follows the syntactic nesting in the corresponding model, just like in feature models. Still, both languages offer mechanisms to control the configuration hierarchy independently from syntactic nesting, which helps to maintain modularity of the developer view.

Groups. Group constraints are used, but much less frequent than assumed in literature. OR groups are not supported by the Kconfig language, and only occur in three CDL models. The studied models use groups with mostly XOR (interval [1..1]) and few MUTEX (interval [0..1]) cardinalities.

Although CDL is more flexible, allowing arbitrary cardinalities using CDL interfaces, they appear overly general. It suffices to include n-ary XOR and MUTEX operators in the constraint language. However, CDL interfaces represent capabilities and, thus, improve modularity in the CDL models.

Constraints. Feature constraints are used extensively. The constraint language should support arbitrary Boolean constraints, including mutual exclusion. Arithmetic operators and comparisons are important for embedded systems—they are often used in the CDL models. String operations other than equality tests seem essential if the build system lacks appropriate support. Arithmetic operations are more likely to be used in embedded software, such as eCos; whereas string operations could be dealt with in the build system outside of the models, as in Linux.

Furthermore, Kconfig uses three-valued logics to specify whether a feature implementation is linked statically, built for dynamic linking, or absent.

Scalability concepts. The concepts beyond FODA are important to scale variability modeling.

Default values, either represented as literals or expressions, are used a lot in the models, saving the user unnecessary configuration work. In particular, derived defaults are often neglected in academic languages.

Visibility conditions are essential in the models; they help to hide whole branches of the configurator hierarchy. Visibility conditions are applied immediately during the configuration process and cannot be temporarily ignored, in contrast to configuration constraints. Despite the different behavior in the configurator, visibility conditions should also define a configuration constraint. Two language constructs are useful: a pure configuration constraint (like CDL's `requires`) and a combined configuration-and-visibility condition (like CDL's `active_if`). Realizing visibility independent of configuration leads to intricate semantics, as can be seen in Kconfig.

Derived features are mostly used to simplify constraints and to perform calculations that otherwise would be hidden in the build system. This way, feature dependencies are specified uniformly and explicitly in one model. Many languages do not support this concept.

Domain orientation. The languages benefit from being domain-specific. The vocabularies are specific to the

13. http://sourceware.org/ml/ecos-discuss/2001-11/msg00161.html

projects and likely improve the understandability of the models within the communities. For example, Kconfig has tristate features, which are heavily used in the Linux model for controlling binding mode; more than half of the features are tristate. CDL uses architectural terms, such as packages, components, and interfaces.

Feature cardinalities. Feature cardinalities [40] were introduced as FODA extensions that enable multiple instantiations (or cloning) of features including all descendants. Their benefit is discussed in the research community. We have, so far, not found any empirical evidence for the need of feature cardinalities, in contrast to a study of configuration challenges [41] (see Section 9.5).

Feature cardinalities are not supported by our languages—perhaps not surprisingly, since even in feature modeling, they are a heavyweight extension. However, we found embedded Tcl `for` loops in some CDL models that generate a number of occurrences of a feature in the configurator, which indicates that developers—although rarely—might find feature cardinalities useful. In some cases, even the number of generated features varied depending on the processor type (determined by the currently loaded target).

Feature attributes. Among the most expressive types of feature modeling languages are attributed feature models [1]. Attributed (sometimes also called *extended*) feature models allow to define an arbitrary number of attributes per feature, their domain, default values, and cross-tree constraints over the attribute values.

In fact, we believe that many concepts, such as visibility and binding modes could be emulated using additional attributes for each feature, when the configurator's implementation is extended to evaluate these attributes. However, since many concepts (like visibility) need to influence the configuration space, modeling them in this way would require additional cross-tree constraints over attributes, which might significantly increase complexity of the models. Especially representing the three-state logics for binding modes in the Linux kernel would literally spoil the models with many min/max constraints (assuming that these are supported in a dialect of attributed feature models). Thus, explicit representation of binding mode using additional switch features, or using attributes in extended feature models [42], would be much less succinct.

Other characteristics of the languages, such as computed defaults, separation of model and configurator hierarchy, or capabilities are not supported by popular attributed feature model languages. For instance, neither CVL nor pure::variants support computed defaults.

7.2 Model Properties

The comparison with S.P.L.O.T. models shows that our subjects represent a new class of models that are larger and more complex than existing ones. No S.P.L.O.T. model displays basic characteristics at the levels close to our subject models. We primarily attribute this to the academic nature of the models available previously.

Our hierarchy analysis indicates special needs to develop tool interfaces: first, to support wide and shallow models; second, to support high variation in branching from very limited to very wide. This finding emphasizes that tools and languages should support visibility control of particular features or whole subtrees.

Our analysis of cross-tree dependencies shows that features are highly inter-related. The high amount of these dependencies challenges existing reasoners, especially the existence of outliers that reference up to 127 other features in their constraints. Still, the linear growth of density of dependencies in relation to the model size indicates that the models abstract over code dependencies and can reduce complexity when reasoning can be done over models instead of code.

Assumptions in the literature about content, structure, and constraints of models differ from our results. For example, Thüm et al. [43] and Mendonça et al. [34] both present reasoning techniques for feature models. Thüm et al. generate trees with maximal branching factors of 10 (too low, see Section 5.2.2), with 25% of inner features representing OR groups (too high, see Section 5.3.1), and 10% of all features having additional constraints (too low, see Section 5.3.2). Mendonça et al. [34] assume an average CTCR of 30% (too low, see Section 5.3.2). Our results challenge all these assumptions.

7.3 Configurators

The Kconfig configurator lacks any kind of reasoning support. To mitigate this, the Kconfig language includes an imperative construct for specifying choice propagation: the `select` statement allows modelers to specify limited propagation directly in a model; however, both the Kconfig user manual and many developer comments in the Linux revision history [35] acknowledge that this construct is very error-prone.

Conflict resolution support in the configurators helps scaling the configuration process. Only the CDL configurator supports it, but the resolution is incomplete and may produce undesirable results. Interestingly, this analysis operator has rarely been considered in literature [1], [44].

Finally, the observation that both configurators only support a very small subset of analysis operations that have been conceived in academia [1] shows that there is significant potential for improving these tools.

8 Threats to Validity

External Validity. The main threat to the external validity of our findings is that they are based on only two languages and a limited set of models. On the other hand, most are used in large, independently developed real-world projects with different objectives, ranging from Linux as a general purpose kernel, over highly configurable system software tools, to eCos as an entire specialized real-time operating system for embedded devices. They are the result of an extensive search for freely analyzable models, languages, and their configurators. This selection can be seen as a theoretical sampling [19], thus, we believe that our results are representative for the domain of open source systems software. We also conjecture that software components in other related domains, especially embedded real-time, such as automotive and avionic control software, will share many characteristics with the studied systems. For the systems of systems architectures in these domains, our observations might not hold, however.

Our subject models and their host projects were likely not developed according to a dedicated software product line engineering process, including systematic domain and application engineering. They are not engineered by a company that carefully manages an explicit portfolio of products. However, the projects share many characteristics with product lines, such as integration in one platform, high configurability, and support for automated derivation of products. Yet, it is not clear whether models developed with a systematic domain engineering share characteristics with our models. This issue is part of our ongoing research [45].

Projects such as Mozilla Firefox or Eclipse IDE are organized as plug-in architectures, with dynamically loadable extensions. Such extensions are often listed on marketplace sites, rather than managed centrally in a closed feature hierarchy. Variability languages for these systems (extension manifests) only capture use dependencies and required version ranges, but no exclusions or other complex constraints. Our study does not apply to such systems.

On a final note, we carefully avoid drawing generalized conclusions for the whole space of variability modeling. Our main contribution is the qualitative and quantitative analysis of individual cases. Our conclusions, such as showing the existence of concepts both in languages and models, or untangling interesting semantic interactions, do not require representativeness of our cases (cf. [19]). It would not even be possible to judge the representativeness of our cases, since the whole population of variability modeling solutions is unknown. Only the space of published academic approaches can be considered well-known so far.

Internal validity. A threat to the internal validity is that our statistics are incorrect. To reduce this risk, we instrumented the native tools to export models in our own format rather than building our own parsers, and we thoroughly tested our analysis infrastructure using synthetic test cases and cross-checked overlapping statistics. We tested our formal semantics specification against the native configurators and cross-reviewed the specifications. We used the Boolean abstraction of the semantics to translate both models into Boolean formulas and run a SAT solver on them to find dead (always inactive) features. We found 114 dead features in Linux and 28 in eCos-i386. We manually confirmed that all of them are indeed dead, either because they depend on features from another architecture or were intentionally deactivated. The other models mostly have no (axTLS, BusyBox, Fiasco, uClinux-dist) or just a few (four in Freetz, Toybox, and uClinux-base) dead features. Only Buildroot (54), CoreBoot (58) and EmbToolkit (53) have significantly many dead features.

Further, we only look at the available artifacts: the languages, manuals, models, and mailing lists. We have not interviewed developers and users. We plan to perform such interviews in the future. In this work, our confidence is based on formalizing the language concepts and on exhaustively testing the configurators and build systems with hand-crafted examples.

For Linux, we only examined the `x86` architecture; however, it represents large and mature portions of the system: it covers 61% of the total of 10415 features and 67% of the total of 8M SLOC.

Since we have not performed interviews with the language designers, we might have misunderstood the original intention of certain language concepts and of actual features in the models. For example, the feature themes were determined by manual model analysis, and the corresponding author could be biased classifying features according to a theme. On the other hand, these themes are based on a consensus of all authors.

9 Related Work

Variability modeling is a key discipline to manage variability in software product lines. Over the past twenty years, academic and industrial research has introduced around 91 variability management approaches [11], most of which (33) are based on feature models to specify variability information. Unfortunately, there are relatively few empirical studies that aim at understanding the use of these techniques in practice.

In fact, a recent survey on the use of feature models [9] identified only five papers (2%) reporting practical experience. References 14, 16, and 17 in [9] are experiences from researchers applying feature modeling to sample problems from industry. References 31 and 37 therein are self-reported industry experiences: the first on using a feature modeling tool prototype on automotive control software and the second one on managing avionic control software with feature models, but with few details on the languages and tools used. Another systematic literature review on variability management (VM) [10] concludes that "there is only little, if any, experimental or detailed comparative analysis to show the relative advantages or disadvantages of different VM approaches". The authors argue that all approaches share similar concepts, and that a reference model would be needed for model transformations, tools, and further research.

9.1 Variability Modeling Languages

Concepts and semantics of variability modeling languages have been studied before.

Sinnema et al. [46] provide a classification of five academic (CBFM [40], COVAMOF [47], VSL [48], ConIPF [49], Koalish [50]) and one commercial (pure::variants [51]) variability modeling language. Each represents a different modeling style: CBFM, ConIPF, and pure::variants model variability as features, COVAMOF and VSL as variation points, and Koalish is embedded in the architecture description language Koala [52]. Using a small sample product line, each technique is categorized according to its modeling capabilities and tool support. Notably, the authors point out the lack of defined modeling processes, particularly to extract and evolve variability.

Schobbens et al. [53] survey seven feature modeling languages—all variants of FODA. Arguing that most of them are not defined formally enough to avoid ambiguities, they develop a common abstract syntax (Free Feature Diagrams) and define individual formal semantics. They also introduce a new language (Varied Feature Diagrams) that is as expressive, but more succinct than the others. The authors conclude that many of the existing variants are expressively-complete, thus, further extensions cannot be justified by expressiveness.

Czarnecki et al. [54] systematically compare feature modeling with decision modeling [55]—another prominent variability modeling technique that became popular with the Synthesis method [56] for software reuse. They compare both techniques on ten dimensions (inspired by [21], [57]), using Kconfig, CDL, and a current proposal [58] of CVL [5] as a reference. The study concludes that there are no major conceptual differences between feature and decision modeling—except for the support of modeling commonality (via mandatory features) in feature models, as decision models focus purely on variability.

Our work complements these three studies mainly in two respects: we analyze languages originating from practice in their full richness, and we quantitatively analyze their real, large-scale instances. In consequence, we provide empirical evidence for the occurrence and frequency of variability modeling concepts in practice. Compared to Sinnema et al. [46], we study language concepts on a more fine-grained level and also reverse-engineer and analyze their formal semantics. Compared to Schobbens et al. [53], our quantitative analysis shows that more advanced concepts than found in the FODA variants are commonly used, which challenges their conclusion about expressiveness. With Czarnecki et al.'s work [54], we can also confirm the use of decision modeling concepts in practice. Studying real decision models would be valuable future work, however.

Our first publication on this topic was a workshop paper [32], in which we reported very early findings on the Linux model. Its contribution was to extract a FODA feature model from Linux and to compare it with feature models from research papers. Our subsequent conference publication [21] differed significantly; it compared the two languages Kconfig and CDL in their full richness, including formal semantics, and analyzed the Linux and eCos-i386 model—less broad then our current analysis, however.

9.2 Variability Modeling in Practice

Although large industrial product lines with thousands of features exist [12], [13], [14], studies or experience reports on variability modeling are sparse. We now describe some notable exceptions.

Grünbacher et al. [59] report on the industrial use of their Dopler tool suite for variability modeling and product derivation. It has been used by Siemens VAI to automate component-based software development since 2007 and to manage Eclipse-based tools. While the language and its semantics are formally defined [7], unfortunately, neither the models nor further empirical data is available. In line with our findings, the authors also point out the need for domain-specific adaptions of tools and languages in various papers [2], [60], with [61] focusing directly on this topic.

Reiser et al. [62] report industrial experiences on variability modeling from the automotive domain. They sketch a framework based on FODA and define seven requirements for highly-flexible variability modeling: (1) principal feature modeling concepts with some extensions, (2) feature meta information with typed attributes, (3) determined order of features (for wide and shallow trees), (4) domain-independence without project-specific cases, (5) formal foundation, (6) open reference implementation and mapping to XML, and (7) compliance constraints (restrict modeling to a subset of concepts). Our study confirms requirements 1, 3, and partly 6; refutes 4; and shows absence of 2, 5, partly 6 (no XML mapping), and 7 in our languages.

Gillan et al. [63] report on application challenges of feature modeling in the telecommunications domain. They conclude that there are many ways to express a feature model for a telecommunications system, which calls for research on methodologies for variability modeling. We confirm the absence of documented methodologies for our languages.

Recognizing the lack of experience reports, researchers have performed case studies on variability modeling. Hubaux et al. [64] present a case study on reverse-engineering variability models. They migrate the heterogeneous configuration mechanisms of

PloneGov [65] to a feature-oriented approach, unifying its configurability into a feature model. The authors report challenges, such as modeling binding times, large numbers (>50) of direct children, or the need to introduce intermediate derived features to optimize dependencies. Unfortunately, neither the size nor further statistics about the model are available. Kästner et al. [66] refactor the Berkeley database into a configurable product line, concluding that very fine-grained variability mechanisms are necessary (even to split expressions in IF statements), which are not even provided by any mature aspect-oriented framework. The created, but relatively small model (38 features) is freely available. Unfortunately, both case studies are performed by researchers and neither product line went into production.

In [35], we study the evolution of the Linux model. Specifically, we investigate how the statistics from our workshop paper [32] have evolved over the last five years and classify the types of edits applied to the model. The analysis shows that the number of dependencies has grown proportionally to the number of features over the last five years, which supports our finding that dependencies grow linearly with the size of a model (see Section 5.3.2).

9.3 Variability Modeling Benchmarks

Using benchmarks or generating variability models are common approaches to evaluate new variability modeling tools or techniques. For both, realistic assumptions about real models are crucial.

Segura et al. [67] introduce a framework for testing and benchmarking feature modeling analysis tools, after recognizing the lack of such [68], [69]. However, the feature model generator requires parameters as input, which the user has to provide. Our study provides realistic properties for such benchmark generators within the domain of systems software.

Passos et al. [70] (including authors of the present study) perform an in-depth analysis of the non-Boolean constraints in all 116 CDL models. The results show that these constraints tend to be simple, but closer analysis identifies challenges to reasoners. The semantic expansion of individual expressions leads to non-linear constraints including multiplication and division of data features; and even the linear constraints appear within complex combinatorial expressions.

The S.P.L.O.T. website also contains generated large-scale feature models based on assumptions stemming from the models in the repository (cf. Sections 5.4 and 7.2). These generated models are used by others to evaluate the performance and scalability of their approaches, such as by Bagheri et al. [29].

9.4 Variability in Open Source Projects

Research community has recognized the appeal of studying configurable open source software due to large source code archives available.

Particularly the Linux kernel has been a frequent study object; several variability-related aspects have been addressed. Sincero et al. [71] are the first to discuss whether the Linux kernel can be seen as a product line, concluding that it shares many characteristics with software product lines, such as configurability and code reuse. In [39], [72], we study the mapping between features and source code in the Linux kernel. This mapping is hidden in imperative build logic, thus, we implement static analysis techniques for Makefiles to derive explicit presence conditions. The latter are expressions that determine inclusion or exclusion of individual source code files for a certain configuration. Tartler et al. [73] apply SAT checks to #IFDEF conditions in Linux source code in order to identify dead code. Furthermore, the code cloning research community has extensively studied the Linux kernel [74], for example to aid product line analysis [75].

Also eCos was studied before. A survey on configurable operating systems [76] emphasizes eCos' component-oriented architecture. Lohmann et al. [77] quantitatively analyze aspects (cross-cutting concerns) in the eCos system and perform a feasibility study on the refactoring of these code parts into an aspect-oriented approach with AspectC++. Our works on CDL ([21], [70], [41], [54]) complement these studies and identify eCos and its configurator infrastructure as highly interesting study objects for further research.

Software ecosystems represent massive variability. Schmid [78] compares metadata (manifest files) of Debian packages and Eclipse bundles with FODA. Notably, Schmid considers manifest files a form of variability models. We take another perspective: while manifests are always fully distributed and evolved as individual units that have relations to other manifests, variability models (even if split over multiple files) are created around a central hierarchy and used and evolved as a whole. Despite this conceptual difference, Cosmo et al. [79] and Galindo et al. [80] show that subsets of variability models can be converted into a Debian package structure with manifest files and back.

9.5 Work Related to our Findings

Table 2 provides references to research on feature modeling concepts. Most of them were present in FODA; however, computed defaults, visibility conditions, and derived features, are marked as rare. State-of-the-art feature modeling languages, such as TVL [8] and pure::variants do not support them. Computed defaults were proposed by researchers [26], but not provided by feature modeling languages.

None of the other variability languages supports binding modes via three-valued logics. Interestingly, Dopler supports visibility conditions. Although it has been defined as a *decision modeling language* [7], it shares many characteristics with feature modeling.

The connection between Kconfig and feature modeling was made in [81]. We advance this work by studying Kconfig's semantics and the Linux model.

Interactive support for resolving variability was ranked highest in a recent expert survey of requirements for product derivation [82]. A study on configuration challenges in Linux and eCos by Hubaux et al. [41]—performed by surveying actual users—also emphasizes the lack of guidance for making choices and the low quality of advice offered by the configurators. Users spend substantial time trying to enable features. This emphasizes the necessity to investigate real-world tools like the Kconfig and CDL configurators, to identify their shortcomings, and to discuss potential improvements.

Interestingly, Hubaux et al. [41] also identify one participant claiming the need for feature cardinalities (cf. Section 7.1). The participant reported having to extend a CDL model in order to support two instances of a flash device. We found emulations of multiple instantiations in CDL models using `for` loops, which limit the number of instantiations to static properties, such as the processor type. The reported case would, in fact, require dynamically setting the number of instantiations, which is not supported by the configurators.

A variety of reasoners have been used to create feature model analyzers and configurators, including CSP solvers [3], SAT solvers [43], [34], and BDD packages [38]. These works tested the reasoners on either small meaningful models or large automatically generated models; however, it is not clear how these tools will scale to handle the Linux and eCos model. This remains future work. However, our previous work [21] meanwhile has inspired the work by Xiong et al. [44], who provide scalable conflict resolution support that is evaluated on the eCos model.

9.6 Knowledge-Based Configuration

While our focus is on variability modeling for software, product configurators for physical goods or services have been actively researched in the field of Knowledge-Based Configuration [83], [84], a branch of Artificial Intelligence (AI). A public catalog[14] currently lists impressive 900 web-based configurators.

Recognizing many overlaps between software configuration, and the older AI-related field of knowledge-based configuration, recent research has started to investigate their relationships, including work on leveraging product configurators and AI techniques for software product lines [85], [49], [50]. Although the relationships between software and product configuration are blurred and part of ongoing research, the following works are closely related to ours.

Hubaux et al. [86] present a research agenda on unifying software and product configuration. Their comparison of both fields concludes that software configuration can benefit from existing techniques in product configuration, such as in the expressiveness of modeling languages and reasoning support (e.g. to optimize configurations according to certain criteria). Notably, they emphasize that both fields lack research on evolution of models.

Abbasi et al. [87] study 111 web-based product configurators. They develop a JavaScript-based analysis tool to semi-automatically extract datasets (configuration options and attributes) from the configurators. They investigate the visualization of configuration options, the handling of constraints, and the type of configuration process supported. Among others, the authors confirm that hierarchical organization and grouping of configuration options is commonly used, and that XOR groups are the most frequent kind of grouping with constraints. Furthermore, cross-tree constraints exist. The authors also identify many limitations in the reasoning procedures, with regard to reliability and runtime efficiency.

Rabiser et al. [88] study user guidance support in product configurators. They identify seven core capabilities from the literature, implement these in their DOPLER tools suite, and evaluate each capability in a user study with industrial participants. Among others, capabilities such as visibility control (hide and show options), views and filters, or freedom in navigation are very important, while immediate feedback turns out to be hard to comprehend for users. Reset and undo functionality is essential to experiment with choices and their impact.

10 Conclusions

We have thoroughly analyzed two real-world variability modeling languages and all identifiable instances used in open source projects, most of which—particularly Linux and eCos—are also used in industrial contexts. Altogether, we quantitatively analyzed 128 models and performed a deep qualitative analysis of 13 of these models.

Kconfig and CDL are interesting and highly relevant study objects. Designed not by researchers, but by developers of large industrial-strength product lines, they are tailored to satisfy the needs of both small (ToyBox, axTLS) and large projects (Linux kernel). The size of the models with up to 6320 features witnesses the scalability of the respective modeling approaches. Furthermore, both languages were developed independently from each other, and independently from feature modeling research. Since they share many similar concepts, they can confirm the importance of the modeling constructs discussed in the literature. As all our study objects are open source, they can be studied openly, and researchers can independently validate and replicate such studies.

Our study raises three main conclusions. First, we confirm that core feature modeling concepts are used

14. http://www.configurator-database.com

in real-world models. However, some concepts have characteristics not discussed in literature yet, such as the separation of syntactic and configurator hierarchy.

Second, our study shows that more advanced concepts, such as visibility conditions, derived features, derived defaults, modularization or binding modes are frequently used. These concepts aim at scaling variability modeling, particularly the creation, maintenance, and configuration of models. While supporting binding modes or modularization in the languages might merely be for convenience, visibility conditions and derived features with expressive constraints seem to be essential for large-scale models. Our language comparison also showed intricate semantic interactions among the advanced concepts, deepening our understanding of such languages.

Third, our studied Kconfig and CDL models have significantly different properties than feature models available to research. They challenge existing tools and common assumptions in the literature about model content, structure, and constraints. They are used to configure very diverse variable aspects of the projects, including debugging, external libraries or test cases. Their shapes significantly deviate from the deep and well-balanced feature trees we commonly see in research. They also follow very different strategies to organize features; these strategies vary across and within the models. More diverse types of constraints, such as visibility, configuration, derived features, and defaults, are used in the models along with a high density of the dependency graph between features.

The publication of the Kconfig and CDL models makes a significant difference in the feature modeling research community. So far, realistic industrial quality models of substantial size were hardly available. This situation has hindered development of trustworthy evaluation methods in research works about variability models. In our study, we explored the differences between these benchmarks, and the state of the art available prior to this work.

As one follow-up of this work, we currently perform an empirical study on industrial variability modeling[15], using a survey questionnaire, expert interviews, and grounded theory as research tools. Initial results are available [45]. Furthermore, we strive to investigate the boundaries between variability models and code, for example to characterize what information is uniquely contained in variability models, and what can be extracted from code by static or dynamic analysis.

Acknowledgments

This work has been supported by the German National Academic Foundation, the Federal Ministry of Education and Research (01IS10033D), and the Ontario Research Fund.

15. http://gsd.uwaterloo.ca/industrial-variability-modeling

References

[1] D. Benavides, S. Segura, and A. Ruiz-Cortés, “Automated analysis of feature models 20 years later: A literature review,” *Information Systems*, vol. 35, no. 6, pp. 615–636, 2010.

[2] D. Dhungana, R. Rabiser, P. Grünbacher, and T. Neumayer, “Integrated tool support for software product line engineering,” in *ASE*, 2007.

[3] J. White, D. Schmidt, D. Benavides, P. Trinidad, and A. Cortés, “Automated diagnosis of product-line configuration errors in feature models,” in *SPLC*, 2008.

[4] M. Janota, G. Botterweck, R. Grigore, and J. P. M. Silva, “How to complete an interactive configuration process?” in *SOFSEM*, 2010.

[5] Object Management Group, “Common variability language (CVL) RFP,” Document ad/2009-12-03, 2009.

[6] K. Kang, S. Cohen, J. Hess, W. Nowak, and S. Peterson, “Feature-oriented domain analysis (FODA) feasibility study,” CMU-SEI, Tech. Rep. CMU/SEI-90-TR-21, 1990.

[7] D. Dhungana, P. Heymans, and R. Rabiser, “A formal semantics for decision-oriented variability modeling with DOPLER,” in *VaMoS*, 2010.

[8] Q. Boucher, A. Classen, P. Faber, and P. Heymans, “Introducing TVL, a text-based feature modelling language,” in *VaMoS*, 2010.

[9] A. Hubaux, A. Classen, M. Mendonça, and P. Heymans, “A preliminary review on the application of feature diagrams in practice,” in *VaMoS*, 2010.

[10] L. Chen, M. Ali Babar, and N. Ali, “Variability management in software product lines: a systematic review,” in *SPLC*, 2009.

[11] L. Chen and M. A. Babar, “A systematic review of evaluation of variability management approaches in software product lines,” *Information and Software Technology*, vol. 53, no. 4, pp. 344–362, 2011.

[12] V. Sugumaran, S. Park, and K. C. Kang, “Software product line engineering,” *Commun. ACM*, vol. 49, no. 12, pp. 29–32, Dec. 2006.

[13] M. Steger, C. Tischer, B. Boss, A. Müller, O. Pertler, W. Stolz, and S. Ferber, “Introducing pla at bosch gasoline systems: Experiences and practices,” in *SPLC*, 2004.

[14] F. Loesch and E. Ploedereder, “Optimization of variability in software product lines,” in *SPLC*, 2007.

[15] M. Mendonca, M. Branco, and D. Cowan, “S.P.L.O.T.: software product lines online tools,” in *OOPSLA*, 2009, http://www.splot-research.org.

[16] U. of Magdeburg, “Spl2go catalog,” http://spl2go.cs.ovgu.de.

[17] R. Zippel and contributors, “kconfig-language.txt,” available in the kernel tree at www.kernel.org, seen 2012-04/10.

[18] B. Veer and J. Dallaway, “The eCos component writer’s guide,” seen Mar. 2010 at ecos.sourceware.org/ecos/docs-latest/cdl-guide/cdl-guide.html.

[19] K. M. Eisenhardt and M. E. Graebner, “Theory building from cases: Opportunities and challenges.” *Academy of management journal*, vol. 50, no. 1, pp. 25–32, 2007.

[20] S. Easterbrook, J. Singer, M.-A. Storey, and D. Damian, “Selecting empirical methods for software engineering research,” in *Guide to Advanced Empirical Software Engineering*. Springer, 2008.

[21] T. Berger, S. She, R. Lotufo, A. Wąsowski, and K. Czarnecki, “Variability modeling in the real: A perspective from the operating systems domain,” in *ASE*, 2010.

[22] S. She and T. Berger, “Formal semantics of the Kconfig language,” technical Note. Available at http://eng.uwaterloo.ca/~shshe/kconfig_semantics.pdf.

[23] T. Berger and S. She, “Formal semantics of the CDL language,” technical Note. Available at http://www.informatik.uni-leipzig.de/~berger/cdl_semantics.pdf.

[24] K. Czarnecki, S. Helsen, and U. Eisenecker, “Staged configuration through specialization and multi-level configuration of feature models,” *Software Process Improvement and Practice*, vol. 10, no. 2, pp. 143–169, 2005.

[25] M. Fowler, *Domain-specific languages*. Addison-Wesley, 2010.

[26] K. Czarnecki and U. W. Eisenecker, *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.

[27] M. Riebisch, K. Böllert, D. Streitferdt, and I. Philippow, “Extending feature diagrams with UML multiplicities,” in *IDPT*, 2002.

[28] K. Schmid and I. John, "A customizable approach to full lifecycle variability management," *Sci. Comput. Program.*, vol. 53, no. 3, pp. 259–284, Dec. 2004.

[29] E. Bagheri, F. Ensan, D. Gaševic, and M. Boškovic, "Modular feature models: Representation and configuration," *Journal of Research and Practice in Information Technology*, vol. 43, no. 2, p. 109, 2011.

[30] M. Bošković, G. Mussbacher, E. Bagheri, D. Amyot, D. Gašević, and M. Hatala, "Aspect-oriented feature models," in *MODELS*, 2010.

[31] D. Dhungana, P. Grünbacher, R. Rabiser, and T. Neumayer, "Structuring the modeling space and supporting evolution in software product line engineering," *J. Syst. Softw.*, vol. 83, no. 7, pp. 1108–1122, Jul. 2010.

[32] S. She, R. Lotufo, T. Berger, A. Wasowski, and K. Czarnecki, "The variability model of the Linux kernel," in *VaMoS*, 2010.

[33] A. J. Massa, *Embedded Software Development with eCos*. Prentice Hall, 2003.

[34] M. Mendonça, A. Wasowski, and K. Czarnecki, "SAT-based analysis of feature models is easy," in *SPLC*, 2009.

[35] R. Lotufo, S. She, T. Berger, K. Czarnecki, and A. Wasowski, "Evolution of the Linux kernel variability model," in *SPLC*, 2010.

[36] A. Egyed, "A scenario-driven approach to trace dependency analysis," *IEEE Trans. Softw. Eng.*, vol. 29, no. 2, pp. 116–132, Feb. 2003.

[37] M. Mendonca, "Efficient reasoning techniques for large scale feature models," Ph.D. dissertation, School of Computer Science, University of Waterloo, Jan 2009.

[38] M. Mendonca, A. Wasowski, K. Czarnecki, and D. D. Cowan, "Efficient compilation techniques for large scale feature models," in *GPCE*, 2008.

[39] T. Berger, S. She, K. Czarnecki, and A. Wąsowski, "Feature-to-Code mapping in two large product lines," in *SPLC*, 2010.

[40] K. Czarnecki, S. Helsen, and U. Eisenecker, "Formalizing cardinality-based feature models and their specialization," *Software Process Improvement and Practice*, vol. 10, no. 1, 2005.

[41] A. Hubaux, Y. Xiong, and K. Czarnecki, "A user survey of configuration challenges in Linux and eCos," in *VaMoS*, 2012.

[42] D. Benavides, P. Trinidad, and A. Ruiz-Cortés, "Automated reasoning on feature models," in *CAiSE*, 2005.

[43] T. Thüm, D. Batory, and C. Kästner, "Reasoning about edits to feature models," in *ICSE*, 2009.

[44] Y. Xiong, A. Hubaux, S. She, and K. Czarnecki, "Generating range fixes for software configuration," in *ICSE*, 2012.

[45] T. Berger, R. Rublack, D. Nair, J. M. Atlee, M. Becker, K. Czarnecki, and A. Wąsowski, "A survey of variability modeling in industrial practice," in *VaMoS*, 2013.

[46] M. Sinnema and S. Deelstra, "Classifying variability modeling techniques," *Information and Software Technology*, vol. 49, no. 7, pp. 717 – 739, 2007.

[47] M. Sinnema, S. Deelstra, J. Nijhuis, and J. Bosch, "Covamof: A framework for modeling variability in software product families," in *Software Product Lines*. Springer, 2004, pp. 25–27.

[48] M. Becker, "Towards a general model of variability in product families," in *Workshop on Software Variability Management*, 2003.

[49] L. Hotz, K. Wolter, T. Krebs, S. Deelstra, M. Sinnema, J. Nijhuis, and J. MacGregor, *Configuration in Industrial Product Families - The ConIPF Methodology*. IOS Press, 2006.

[50] T. Asikainen, T. Soininen, and T. Männistö, "A koala-based approach for modelling and deploying configurable software product families," in *Software Product-Family Engineering*. Springer, 2004, pp. 225–249.

[51] P.-S. GmbH, "Technical white paper variant management with pure::variants," Tech. Rep., 2006. [Online]. Available: http://www.pure-systems.com/fileadmin/downloads/pv-whitepaper-en-04.pdf

[52] R. van Ommering, F. van der Linden, J. Kramer, and J. Magee, "The koala component model for consumer electronics software," *IEEE Computer*, vol. 33, pp. 78–85, 2000.

[53] P.-Y. Schobbens, P. Heymans, J.-C. Trigaux, and Y. Bontemps, "Generic semantics of feature diagrams," *Comput. Netw.*, vol. 51, no. 2, pp. 456–479, 2007.

[54] K. Czarnecki, P. Grünbacher, R. Rabiser, K. Schmid, and A. Wasowski, "Cool features and tough decisions: A comparison of variability modeling approaches," in *VAMOS*, 2012.

[55] D. Dhungana and P. Grünbacher, "Understanding decision-oriented variability modelling," in *SPLC-ASPL*, 2008.

[56] S. P. C. H. VA., *Reuse-Driven Software Processes Guidebook. Version 02.00. 03*. Defense Technical Information Center, 1993.

[57] K. Schmid, R. Rabiser, and P. Grünbacher, "A comparison of decision modeling approaches in product lines," in *VaMoS*, 2011.

[58] "Common variability language (CVL), OMG initial submission," 2011, available on request.

[59] P. Grünbacher, R. Rabiser, D. Dhungana, and M. Lehofer, "Model-based customization and deployment of Eclipse-based tools: Industrial experiences," in *ASE*, 2009.

[60] D. Dhungana, P. Grünbacher, and R. Rabiser, "Decisionking: A flexible and extensible tool for integrated variability modeling," in *1st International Workshop on Variability Modelling of Software-intensive Systems*, 2007, pp. 119–128.

[61] ——, "Domain-specific adaptations of product line variability modeling," in *Situational Method Engineering: Fundamentals and Experiences*, ser. IFIP International Federation for Information Processing. Springer, 2007, vol. 244, pp. 238–251.

[62] M. Reiser, R. Tavakoli, and M. Weber, "Unified feature modeling as a basis for managing complex system families," in *VaMoS*, 2007.

[63] C. Gillan, P. Kilpatrick, I. Spence, T. Brown, R. Bashroush, R. Gawley *et al.*, "Challenges in the application of feature modelling in fixed line telecommunications," in *VaMoS*, 2007.

[64] A. Hubaux, P. Heymans, and D. Benavides, "Variability modeling challenges from the trenches of an open source product line re-engineering project," in *SPLC*, 2008.

[65] G. Delannay, K. Mens, P. Heymans, P. Schobbens, and J. Zeippen, "Plonegov as an open source product line," in *SPLC-OSSPL*, 2007.

[66] C. Kästner, S. Apel, and D. Batory, "A case study implementing features using AspectJ," in *SPLC*, 2007.

[67] S. Segura, J. A. Galindo, D. Benavides, J. A. Parejo, and A. Ruiz-Cortés, "Betty: benchmarking and testing on the automated analysis of feature models," in *VaMoS*, 2012.

[68] D. Benavides, A. Ruiz-Cortés, P. Trinidad, and S. Segura, "A survey on the automated analyses of feature models," in *JISBD 2006: XV Jornadas de Ingeniería del Software y Bases de Datos*, J. Riquelme and P. Botella, Eds., Barcelon, 2006.

[69] S. Segura and A. Ruiz-Cortés, "Benchmarking on the automated analyses of feature models: A preliminary roadmap," in *VaMoS*, 2009.

[70] L. Passos, M. Novakovic, Y. Xiong, T. Berger, K. Czarnecki, and A. Wasowski, "A study of non-boolean constraints in variability models of an embedded operating system," in *FOSD*, 2011.

[71] J. Sincero, H. Schirmeier, W. Schröder-Preikschat, and O. Spinczyk, "Is the Linux kernel a software product line?" in *SPLC-OSSPL*, 2007.

[72] T. Berger, S. She, K. Czarnecki, and A. Wąsowski, "Feature-to-Code mapping in two large product lines," http://informatik.uni-leipzig.de/~berger/tr/2010-berger.pdf, University of Leipzig, Tech. Rep., 2010.

[73] R. Tartler, D. Lohmann, J. Sincero, and W. Schröder-Preikschat, "Feature consistency in compile-time-configurable system software: facing the Linux 10,000 feature problem," in *EuroSys*, 2011.

[74] C. Roy and J. Cordy, "A survey on software clone detection research," School of Computing, Queen's University, Tech. Rep. 2007-541, 2007.

[75] W. Koleilat and N. Shaft, "Extracting executable skeletons," Cheriton School of Computer Science, University of Waterloo, Tech. Rep., 2007.

[76] L. F. Friedrich, J. Stankovic, M. Humphrey, M. Marley, and J. Haskins, "A survey of configurable, component-based operating systems for embedded applications," *IEEE Micro*, vol. 21, no. 3, pp. 54–68, May 2001.

[77] D. Lohmann, F. Scheler, R. Tartler, O. Spinczyk, and W. Schröder-Preikschat, "A quantitative analysis of aspects in the eCos kernel," in *EuroSys*, 2006.

[78] K. Schmid, "Variability modeling for distributed development - a comparison with established practice," in *SPLC*, 2010.

[79] R. D. Cosmo and S. Zacchiroli, "Feature diagrams as package dependencies," in *SPLC*, 2010.

[80] J. A. Galindo, D. Benavides, and S. Segura, “Debian packages repositories as Software Product Line models. Towards automated analysis,” in *ACoTA*, 2010.
[81] J. Sincero and W. Schröder-Preikschat, “The Linux kernel configurator as a feature modeling tool,” in *SPLC-ASPL*, 2008.
[82] R. Rabiser, P. Grünbacher, and D. Dhungana, “Requirements for product derivation support: Results from a systematic literature review and an expert survey,” *Information and Software Technology*, vol. 52, no. 3, 2010.
[83] M. Stumptner, “An overview of knowledge-based configuration,” *AI Communications*, vol. 10, no. 2, pp. 111–125, Apr. 1997.
[84] A. Günter and C. Kühn, “Knowledge-based configuration-survey and future directions,” in *XPS-99: Knowledge-Based Systems. Survey and Future Directions*, ser. Lecture Notes in Computer Science, F. Puppe, Ed. Springer Berlin / Heidelberg, 1999, vol. 1570, pp. 47–66.
[85] T. Asikainen, T. Männistö, and T. Soininen, “Using a configurator for modelling and configuring software product lines based on feature models,” in *SPLC-DWS*, 2004.
[86] A. Hubaux, D. Jannach, C. Drescher, L. Murta, T. Männistö, K. Czarnecki, P. Heymans, T. Nguyen, and M. Zanker, “Unifying software and product configuration: A research roadmap,” in *ConfWS*, 2012.
[87] E. Abbasi, A. Hubaux, M. Acher, Q. Boucher, P. Heymans, A. Heymans, F. FSR, and W. Region, “What’s in a web configurator? empirical results from 111 cases,” PReCISE - FUNDP, University of Namur, Tech. Rep. P-CS-TR CONF-000001, 2012.
[88] R. Rabiser, P. Grünbacher, and M. Lehofer, “A qualitative study on user guidance capabilities in product configuration tools,” in *ASE*, 2012.

APPENDIX
MODEL PLOTS

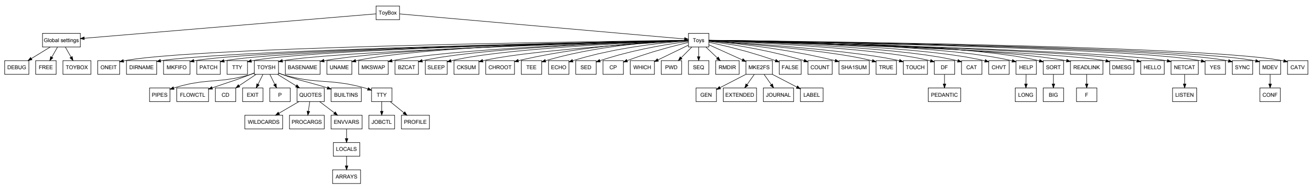


Fig. 17. ToyBox hierarchy (71 features).

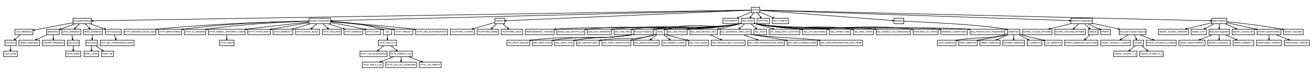


Fig. 18. axTLS hierarchy (108 features).

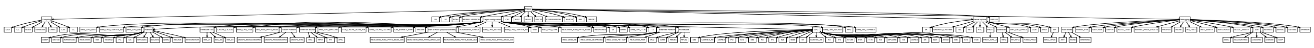


Fig. 19. Fiasco hierarchy (171 features).