Variability Modeling in the Real: A Perspective from the Operating Systems Domain

Thorsten Berger
University of Leipzig
berger@informatik.uni-leipzig.de

Steven She
University of Waterloo
shshe@gsd.uwaterloo.ca

Rafael Lotufo
University of Waterloo
rlotufo@gsd.uwaterloo.ca

Andrzej Wąsowski
IT University of Copenhagen
wasowski@itu.dk

Krzysztof Czarnecki
University of Waterloo
kczarnec@gsd.uwaterloo.ca

ABSTRACT

Variability models represent the common and variable features of products in a product line. Several variability modeling languages have been proposed in academia and industry; however, little is known about the practical use of such languages. We study and compare the constructs, semantics, usage and tools of two variability modeling languages, Kconfig and CDL. We provide empirical evidence for the real-world use of the concepts known from variability modeling research. Since variability models provide basis for automated tools (feature dependency checkers and product configurators), we believe that our findings will be of interest to variability modeling language and tool designers.

Categories and Subject Descriptors

D.2.2 [**Software Engineering**]: Design Tools and Techniques; D.2.8 [**Software Engineering**]: Metrics

General Terms

Design, Languages, Measurement

Keywords

Configuration, empirical software engineering, feature models, product line architectures, variability modeling

1. INTRODUCTION

Variability models represent the common and variable characteristics, or *features*, of products in a product line. Product line developers use them to manage the addition and evolution of features and their dependencies. Product line users derive concrete products from variability models. A range of automated tools support these activities: analyzers verify model consistency or detect dead features [1] and graphical configuration tools (*configurators* for short) support intelligent choice propagation and model completion [6, 23, 9]. 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. Recognizing the interest, the OMG currently seeks proposals for a Common Variability Language (CVL) standard [14].

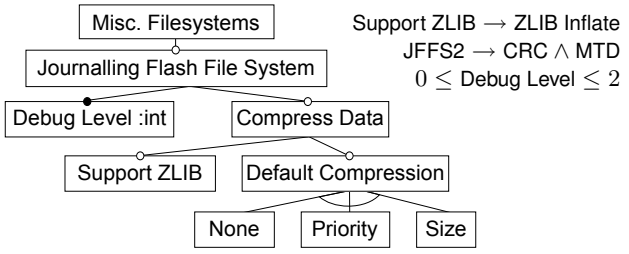

Figure 1: Feature model interpretation of JFFS2

Although variability modeling languages have been designed both in academia [10, 5, 3] and industry (pure::variants, Gears), little is known on their practical use. A recent survey [8] lists many research contributions on feature models but no empirical studies of industrial practice of feature modeling. Our work addresses this gap. We study and compare two variability modeling languages and their use: Kconfig [24] and Component Description Language (CDL) [22]. Both were developed as part of open-source operating systems (OSs). Kconfig is used to describe the variability of the Linux kernel. CDL is part of eCos, a real-time (RT) operating system for embedded devices. Both Linux and eCos have vast configuration spaces with thousands of features, which explains their need for variability management.

We compare the constructs, the semantics, and the usage of Kconfig and CDL, while using the well-researched concepts of *feature modeling* as a reference. Feature models were originally introduced as part of the Feature-Oriented Domain Analysis (FODA) [10]. They gained popularity with product-lines 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. Fig. 1 presents a sample feature model in the FODA notation, which illustrates the core concepts shared by many feature modeling languages. The sample model shows the

ASE'10, September 20–24, 2010, Antwerp, Belgium.

variability of the Journalling Flash File System—one of the numerous files systems supported in both Linux and eCos. 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 the XOR group constraint—exactly one of the three choices has to be selected. Finally, textual cross-tree constraints are listed to the right.

Our goal is (1) to provide quantitative and qualitative empirical evidence whether the concepts researched in feature modeling are used in real-world modeling languages and product lines and, if so, how they are used; (2) to widen the understanding of the design space for the studied concepts.

We analyzed the two languages *and* the models expressed in them, the Linux kernel model and the eCos model. We have instrumented the native tools supporting the languages to build an infrastructure for collecting quantitative data about the models. Guided by the collected statistics, we inspected large parts of the models to study actual usage patterns. In order to also compare the languages directly (as opposed to via models) we also compared their semantics. Many semantic differences turned out to be subtle and not immediately obvious from the syntax.

With respect to the first objective, our study shows that the core concepts of FODA feature modeling are supported by both Kconfig and CDL and are used in both the Linux and the eCos models. These include Boolean (optional), integer and string features, a hierarchy, group constraints, and cross-tree constraints. Interestingly, both languages and models use concepts that are beyond FODA and have not been as widely studied as the core concepts:

- *Visibility*: Both languages allow controlling the visibility of features in the user interface (UI) of the configuration tool via *visibility conditions*.
- *Computed defaults*: They both support computing default values of features, using values of other features.
- *Binding modes*: Kconfig uses three-valued logics to specify whether a feature implementation is linked statically, built for dynamic linking, or absent. FODA uses much more space, if representing this in Boolean logic.
- *Domain-specific vocabulary*: Both languages provide specialized vocabularies for various kinds of features, including architectural terms (CDL), such as components and interfaces, or terms related to the configurator UI (Kconfig), such as menus. The vocabularies are specific to the two projects and likely improve the understandability of the models within the communities.

For the second objective, the study reveals interesting differences in how Kconfig and CDL provide the above concepts:

- *Feature representation*: Kconfig treats each feature as a variable, either of Boolean, integer or String type; CDL, in addition, also supports composite features that contain both a Boolean component encoding feature presence and a numeric or string value.
- *Feature hierarchy*: For both languages the hierarchy shown to users in a configurator largely follows the syntactic nesting in the corresponding model; still, both languages offer some mechanisms to control the configuration hierarchy separately from syntactic nesting.

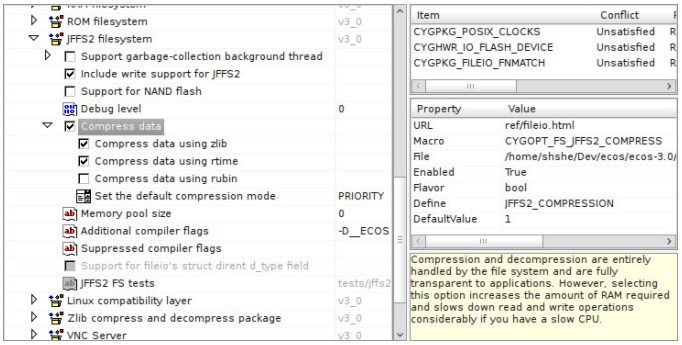

Figure 2: The eCos Configtool GUI

- *Group constraints*: Kconfig supports feature grouping constructs with group cardinalities limited to XOR, MUTEX, and OR. CDL is more flexible, allowing integer intervals as group cardinalities; however, the studied models only use XOR, MUTEX, and OR cardinalities. In feature modeling (and Kconfig), only parents can impose group constraints on their children; in CDL features can impose group constraints on any other features and eCos makes use of it.
- *Visibility and computed defaults*: Both languages support two types of conditions on a feature: conditions defining only configuration dependencies and conditions defining both such dependencies and visibility. Further, visibility and computed defaults are used together to provide *derived features*—automatically computed features, not shown to the user in the UI. Derived features are used to simplify constraints or define implementation features used in the build system.
- *Constraint language*: Both languages support arbitrary Boolean constraints. Kconfig supports also equality tests on integer and string values. CDL adds various arithmetic and string operations, and a few built-in functions. The models reveal that arithmetic operations are likely needed for embedded software, as in eCos; whereas string operations could be dealt with in the build system outside of the models, as in Linux.

We observed limitations in the configurators for Kconfig and CDL (see Fig. 2 for the UI of the eCos tool). Particularly the Kconfig configurator lacks reasoning procedures to support choice propagation. To mitigate this, the Kconfig language includes an imperative construct for specifying choice propagation, which delegates this task to model developers; however, both the Kconfig user manual and many developer comments in the Linux revision history [11] acknowledge that using the construct is very error-prone. The eCos configurator is far more intelligent thanks to an inference engine; however, the engine offers incomplete reasoning and may propose configuration choices that would not be desirable for the user. Interestingly, both configurators follow a *reconfiguration* paradigm: any configuration task starts with an initial, possibly default, configuration. However, scalable reasoning to support conflict resolution in reconfiguration for rich languages such as Kconfig and CDL remains an interesting open problem.

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 these large projects (8M SLOC for Linux and 0.9M SLOC for eCos). The size of the models (6320 features for Linux and 1244 features for eCos) witnesses the scalability of the respective modeling approaches.

Since both Linux and eCos are open source, their usage can be studied openly, and researchers can independently validate and replicate such studies. Both languages support quite different systems: the kernel of a general purpose OS (Linux) vs. the entire real-time OS for embedded applications (eCos). They were developed independently from each other, and independently from the feature modeling languages with research origin. Since they share many similar concepts, they can confirm the importance of the modeling constructs discussed in the literature.

Although our analysis is limited to variability languages and models from the OS domain, we believe that other projects such as RT embedded systems that require static configuration will likely have similar requirements. Thus, our findings are of interest to a growing audience of variability modeling language and tool designers, especially in efforts such as the development of OMG's CVL standard.

2. THE SYSTEMS

The implementation of the Kconfig language is distributed together with the Linux kernel source. Kconfig has been used to specify build-time dependencies of the kernel since 2002. The Linux configurator reads the Kconfig model and allows the user to select features in a graphical UI closely resembling the CDL configurator of Fig. 2. It outputs a set of feature-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.

CDL was designed for the purposes of the configurable embedded operating system eCos (ecos.sourceware.org). Unlike Kconfig, which is a standalone language, CDL is embedded in Tcl, a dynamic and highly extensible scripting language. CDL inherits characteristics from Tcl, such as syntactic nesting of blocks, dynamic typing of values, and a rich set of operators in constraint expressions. CDL's configurator (Fig. 2) offers an inference engine for conflict resolution.

The studied version 3.0 of eCos supports 116 hardware architectures, called *targets*, and comprises almost a million lines of code. The code base is divided into packages, each one containing the source code and a set of CDL files declaring the configurability of the package. Each target defines a set of packages specific to the architecture. So-called *templates* aggregate packages with more cross-architecture 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 scope our analysis to the x86 architecture in Linux and i386PC target with the all template in eCos. We extended the configurators to export the configuration tree together with all the information necessary for our analysis.

3. THE LANGUAGES

To compare Kconfig and CDL, we reverse-engineered formal semantics specification for each of them [18, 2], by analyzing user manuals, testing the tools on examples, and inspecting tool implementations. This step allowed us to discover many subtle differences and connections.

Here we summarize the key similarities and differences between the languages using 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 in fact extracted from the original Linux and eCos models. They define the features of the Journalling Flash File System, version 2 (JFFS2), supported by both OSs. In fact, eCos's JFFS2 implementation was ported from Linux. JFFS2 is one of 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 underlined; we also left out unnecessary parts of the corresponding sources. Our discussion will follow Table 1 as an outline. The table summarizes the similarities and differences among Kconfig, CDL, and FODA-based feature modeling and provides citations for concepts that go beyond the original FODA notation.

Features. A feature is a label that 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;
2. *Grouping feature*: a label grouping a set of other related features, such as a menu;
3. *Implementation feature*: a configuration option accessed by the build system or a generator; and
4. *Derived feature*: a configuration option automatically computed via constraints.

Kconfig and CDL are domain-specific languages, providing specialized keywords for various *kinds* of features (Tbl. 1, row 1). Feature kinds in Kconfig reflect their appearance in the configurator UI: *menus* are pure grouping features; *menuconfigs* are menus that can be enabled and disabled by clicking; *choices* are like menus or menuconfigs except that they also impose grouping constraints on their children; and *configs* are individual options. The menuconfig MISC_FILESYSTEMS (Fig. 3 Line k-1) corresponds to the root in Fig. 1. It contains the 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 in `if` (k-4) and `endif` (k-49).

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 simple configuration options (leaves). Several possibly exclusive features can provide equivalent functionality required elsewhere. *Interfaces* are abstractions allowing imposing cardinality constraints in such cases. Line c-9 states that CYGPKG_FS_JFFS2 implements the interface CYGINT_IO_FILEIO (not shown).

	concept	Kconfig	CDL	feature models [10]
feature kinds	Grouping	menu, menuconfig, choice	package, component, interface	feature
	Individual	config	option	feature
feature representation	Composition	single value	bool. value w/opt. data value	bool. value w/opt. attribute
	Feature type			
	Switch	bool, tristate	bool, booldata	(optional)
	Data	hex, int, string	booldata, data	integer, string
	None	(menu)	none	(mandatory)
hierarchy	Specification	syntactic and computed	syntactic and reparenting	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 [4]
	Or [1..$*$]	mandatory tristate choice	interface constraint, INT ≥ 1	OR group [4]
	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 [16]
feature constraints	Configuration	select	requires, active_if	cross-tree constraint
	Value restrictions	range	legal_values	cross-tree constraint
	Derived features	non-prompt default	calculated, interface	rare [5]
	Defaults	prompt default	default_value	rare [4]
	Visibility conditions	prompt condition	active_if	rare [5]
	Expression operators	&&, ||, !, =, !=	also inequality, arithm. and str. ops.	unspecified
	Binding modes	three-value logic	N/A	rare [4]
other	Textual content	prompt, help	display, description	description
	Modularization	textual inclusion	dynamic loading/unloading	rare [3]
	Build symbols	one-to-one	one-to-many	unspecified
	Code mappings	no, uses KBuild (m:n)	yes (1:n), and build specifications	N/A

Table 1: Mapping of concepts between Kconfig, CDL and feature modeling

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).

Kconfig and CDL differ in the ways they represent configurations (see Tbl. 1, row 2). 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:

$$\sigma\colon F \mapsto \text{Val} \qquad \text{and if } \sigma(f) = v \text{ then } v \in \text{type-of}(f)$$

Table 1 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.

The Kconfig type `bool` has two values, `y` and `n`, internally represented by 2 and 0; 0 denotes feature absence, while 2 means that the feature's implementation is compiled statically into the kernel. `Tristate` resembles `bool`, except for the additional `m` value, represented internally by 1, which denotes that the feature should be compiled as a dynamically loadable module. For example JFFS2_ZLIB (k-32) has type `bool` and JFFS2_FS (k-6) is `tristate`. 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 has two values: an *enabled state* and a *data value*. The enabled state 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:

$$\sigma\colon F \mapsto \{0,1\} \times \text{Val} \quad \text{and if } \sigma(f) = (e,d) \text{ then } d \in \text{type-of}(f)$$

CDL refers to the type of a feature as a *flavor*. The available flavors map neatly 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

More precisely, features with the flavors `none` and `data` can be made *optional* by specifying a configuration constraint (explained later). Still, an optional feature with flavor `none` or `data` differs from its respective `bool` or `booldata` counterpart: the latter two are shown as user-selectable checkboxes in the configurator, whereas the former two have no checkboxes since their presence is controlled via visibility conditions. Figure 3 includes features assuming numeric values (CYGOPT_FS_JFFS2_DEBUG), Bool values (CYGOPT_JFFS2_NAND), or strings (CYGOPT_FS_JFFS2_COMPRESS_CMODE).

Hierarchy. Typical modeling languages organize features into hierarchies (Tbl. 1, row 3). We distinguish between syntactic and configuration hierarchy. The former is given by the syntactic nesting of features, such as the nesting of configs in menus or choices in Kconfig, or options and components in other components and packages in CDL. The configuration hierarchy is shown to the user in a configurator (cf. Fig. 2). In the notation of Fig. 1, the diagrammatic tree represents both the intended configuration hierarchy and the syntactic nesting. In Kconfig and CDL the configuration hierarchy can deviate from the syntactic one.

In Kconfig, syntactic nesting within menuconfigs and choices will be reflected in the configuration hierarchy; however, configs can also appear as children of other configs, even though

```
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
```

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

they cannot be nested syntactically. 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 configuration hierarchy mostly follows the syntactic nesting of features unless declared otherwise. *Reparenting* is a mechanism to explicitly specify a parent from a different syntactic scope (see Line c-10). It allows adjusting the developer-oriented structure of the model to make it more user-oriented before it is shown in the configurator.

An important property of the configuration 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 configuration hierarchy in CDL has this property too. In contrast, the configuration hierarchy in Kconfig only enforces the child-to-parent implications for the visibility conditions. So the parent of a feature that is visible in the configurator must be visible. However, if the parent is not selected, a feature can still be selected automatically, unlike in other known feature modeling languages.

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.

Group constraints. In feature modeling, group constraints restrict the number of sibling features to be selected if their parent is selected (Tbl. 1, 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, a *choice* groups a set of features and imposes a group constraint on them. Choices are either `bool` or `tristate` with a *mandatory* or *optional* modifier flag. If not specified otherwise, a choice is `bool` and mandatory; thus, the choice in line k-38 is an XOR group. 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.

CDL *interfaces* are a more expressive construct for restricting cardinality of a set of features. The value of an interface counts the number of its selected implementations (concrete features). Restricting this value introduces a cardinality constraint. In contrast to FODA-like languages, CDL does not require that all implementing features are

siblings—the feature activating the group constraint need not be a parent of the constrained features.

Feature Constraints. CDL and Kconfig support three types of constraints (Tbl. 1, 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 constraint Support ZLIB→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` statement 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 valid string values (c-42). Default values are specified using `default` in Kconfig (k-16) and `default_value` in CDL (c-16). If no default value is specified, Kconfig assumes 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.

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. We refer to such invisible features with calculated values as *derived features.* JFFS_FS_WRITEBUFFER in line k-22 is derived since it has no prompt declared, thus, has a false visibility condition, but has a default that determines its 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 *mod* corresponds to the unknown state. The equality test is defined only between constants (i.e. `tristate`, `int`, `hex` and `string`) and features state. It evaluates to `y` (2) if the values match, and to `n` (0) otherwise.

Textual content. Both Kconfig and CDL allow providing natural language descriptions for features (Tbl. 1, 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. These symbols and their values are referenced in presence conditions inside the KBuild system and control the inclusion of particular source files from the Linux codebase. Although these presence conditions can be of any form, they are a disjunction or conjunction of symbols in most cases.

In CDL, feature names do not always correspond directly to symbols for the build and the preprocessor. Instead, a more fine-grained control over symbols is supported, such as suppressing symbols, defining additional ones, or changing their formatting. Line c-18 shows an example of a feature defining a build symbol, CONFIG_JFFS2_FS_DEBUG, which actually appears in the code ported from Linux to eCos.

	Linux	#ftrs	%ftrs	eCos	#ftrs	%ftrs
switch	bool type	2313	37	bool flavor	455	37
	tristate type	3692	58	booldata flavor[1]	192	15
		6005	95		647	52
data	int type	175	2.8			
	hex type	32	0.5	data flavor	489	39
	string type	28	0.4	booldata flavor[1]	192	15
		235	3.7		681	55
none	menu (no type)	80	1.3	component (flavor none)	108	9
	Total	**6320**	100	Total	**1244**	100

[1] Repeated as booldata is both switch and data feature type.

Table 2: Representation statistics (cf. Tbl. 1 row 2)

	Linux	#ftrs	%ftrs	eCos	#ftrs	%ftrs
grouping	menu	80	1.3	components	265	21
	menuconfig	175	2.8	packages	56	4.5
		255	4		321	26
grouping with constraints	XOR	39	0.6	XOR	11	0.9
	OR	3	0.05	OR	0	0
	MUTEX	0	0	MUTEX	1	0.08
		42	0.7		12	1

Table 3: Grouping statistics (cf. Tbl. 1 row 4)

4. THE MODELS

Let us turn from the languages to their use. We now discuss the *models* of Linux and eCos, expressed in Kconfig and CDL respectively. Again we will use Table 1 as an outline.

Features and grouping. The Linux model has 6320 features; the eCos model is one fifth of this size. Table 2 shows the breakdown of features by type. The majority of features (95%) in the Linux model are bool or tristate, with only 3.7% having integer or string types. In contrast, more than half of the features in eCos are data features; this is interesting since the majority of the examples found in the literature have few or no such features [19]. Note that we listed `booldata` features both as switch and data features to reflect their dual nature; the percentages are given with respect to the total number of features, which counts them only once. We established the following reasons for the large percentage of data features in eCos. First, some feature kinds contain data values by default, even though they are not intended to be directly set by the user: interfaces carry the count of the number of implementing features selected in a configuration and packages always have the type `booldata`, with the data part representing the package version as a string.

There are 130 data or `booldata` interfaces and 56 `booldata` packages in eCos. Further, 24 of `data` or `booldata` features represent enumerations, similar to the last option in Fig. 3. Interestingly, they accept 117 legal values in total, which is the number of additional `bool` features that we would need to express these enumerations in Kconfig, as shown in Fig. 3. There are also 72 data or `booldata` features representing compiler flags, 4 representing linker flags, and 40 holding names of files with test code. The remaining 355 data or `booldata` features, or 28% of all features, represent diverse configuration constants such as priorities, buffer sizes, and supported IO ports. Many of these constants are specific to a RTOS and would either be set dynamically or not be configurable in Linux.

As many as a quarter of all eCos features are explicit grouping features (Table 3), as opposed to Linux's 4%. This is unexpected given that the percentages of non-leaf features in both models are comparable: 16% for Linux and 24% for eCos (some components in eCos are leaves). The reason is that Linux also allows nesting of configs, so configs can also take a grouping role. Whereas menus and menuconfigs create a separate menu structure requiring explicit drill-down by the user, config hierarchies are shown by indentation and are, thus, more lightweight to navigate.

Less than 1% of features in eCos and Linux impose group constraints on their children. Let us see how group constraints are used in practice. The three OR groups in Linux are motivated by binding time: the OR constraint in the model 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 represents three alternative random number generators. There are no MUTEX groups in Linux; a possible reason is the need to define a build symbol even when no group member is selected, cf. JFFS_CMODE_NONE in Fig. 3.

Recall that CDL interfaces generalize group cardinality constraints. This generality is not exploited in practice, though. We did not find a single instance of a group cardinality constraint, which is a proper (m, n)-interval, as opposed to intervals with lower bound being 0 or 1 and upper bound being 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, the interface and the implementing features are usually far apart, i.e., 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. In total, we had 81 such constraints in eCos.

Hierarchy. Both Linux and eCos have shallow configuration hierarchies, with an average depth of 4 for Linux and 3 for eCos and maximum depth of 8 and 6 respectively. The number of features with a given number of children decreases sharply with the increase of the number of children: the majority of features are leaves (5316 and 947 respectively); the second-largest class are single-child parents (452 and 76), followed by two-child parents (161 and 72). Nevertheless the maximum number of children (branching) is as much 158 and 29 respectively. This indicates a need to develop modeling interfaces that support high variation in branching from very limited to very wide.

Recall that, unlike 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. We verified with a SAT solver applied to a derived boolean semantics of the Linux model that 300 features do not imply their parents (like JFFS2_ZLIB in Fig. 3).

We found 39 (3%) re-parented features in eCos. Most

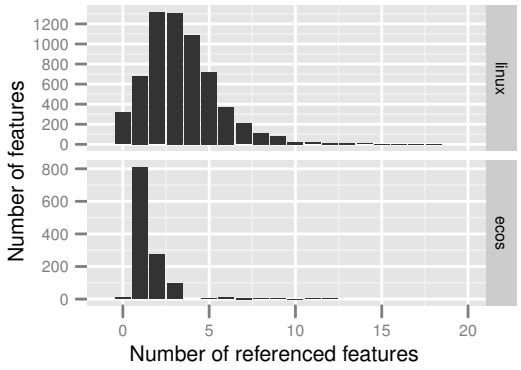

Figure 4: Feature dependencies per feature

re-parentings move packages in the hierarchy; however, 10 options and 2 components were re-parented as well. For example, the CYGBLD_GLOBAL_OPTIONS component from CYGPKG_HAL_I386_PC package was promoted to the top-level and, in addition to its syntactic children, two new options were re-parented under this component. Still, since relatively few features (3% in eCos, 5% in Linux) violate hierarchical rules of feature modeling, we observe that practitioners find hierarchical organization of dependencies natural.

Constraints. The vast majority of features (surprisingly 86% both for Linux and eCos) declare constraints of some sort (configuration, visibility, or defaults). Fig. 4 shows histograms with *dependencies* per feature, defined as the number of features referenced in constraints of a given feature. In Linux, most features refer to 2-4 other features; this range is much lower in eCos, with typically 1-2 dependencies. Some features declare a large number of dependencies; the maximum is 56 in Linux and 21 in eCos.

Table 4 summarizes the use of visibility conditions and defaults. Both Linux and eCos models use visibility conditions. In Linux, 3% of features have an explicitly specified prompt condition (like JFFS2_COMPRESS), rather than just via `depends on`, and 10% of features in eCos use `active_if`. Further, 15% of Linux features specify explicit defaults; eCos makes heavy use of explicit defaults (69% of features). Only a small part of features is computed via expressions: 2% for Linux and 7% for eCos; the remaining defaults are specified as literals. Recall that Linux supports conditionally-derived features, i.e., features that are derived or user-changeable with a default value, depending on a condition; 3% of Linux features belong into this category. Finally, 12% (Linux) and 18% (eCos) of features are (unconditionally) derived.

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)
```

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, outside of the model.

Summary. Let us now summarize the main lessons learnt:

- Core FODA concepts (Tbl. 1) are used in both models.
- Boolean features are the basic and most common type; the constraint language should support arbitrary Boolean constraints, including mutual exclusion.
- Linux uses heavily the three-state logics for controlling binding mode; more than half features are tristate.
- The languages benefit from being domain-specific. Domain vocabulary increases understandability.
- Integer features are important for embedded systems; eCos uses arithmetic operators and comparisons.
- Strings are mostly used for file names; string operations other than equality tests seem essential if the build system lacks appropriate support.
- Group constraints dependent on a remote feature improve modularity in the eCos model. Since only basic cardinalities are used, the interfaces appear overly general. It suffices to include n-ary XOR, OR and MUTEX operators in the constraint language.
- Separating configuration hierarchy from syntactic hierarchy helps maintain modularity of the developer view separately from the user view.
- Like in feature modeling, child-to-parent implications are enforced in CDL and in most of the Linux model.
- Default values (also computed) are used a lot in practice, saving the user unnecessary configuration work.
- Visibility control is essential in both models. Two constructs are useful: a pure configuration condition (like `requires`) and a combined configuration-and-visibility condition (like `active_if`). Configuration independent of visibility leads to intricate semantics.
- 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 one model.

5. THE CONFIGURATORS

Kconfig and CDL are supported by 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 reaches a valid configuration. The Kconfig configurator prevents the user from modifications that violate constraints; the eCos 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

Concept	Linux		eCos	
Visibility conditions	200	(3%)	123	(10%)
Explicit defaults	944	(15%)	857	(69%)
Computed (expressions)	104	(2%)	82	(7%)
Literals	632	(10%)	775	(62%)
Conditionally derived	198	(3%)	N/A	
Unconditionally derived	736	(12%)	218	(18%)

Table 4: Visibility and default statistics

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 [11]. Still, the Linux model is full of `select` statements, as this is the only way to obtain (limited) propagation in the configurator.

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.

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 resolution, by using the number of required changes and source of the values being changed, e.g. 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 3 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 on the mailing list (sourceware.org/ml/ecos-discuss/2001-11/msg00161.html) 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. Tools employing complete reasoners do exist for package configuration involving simple use dependencies and version ranges. For example p2 in Eclipse is using a SAT solver. However, scalable reasoning to support conflict resolution for rich languages such as Kconfig and CDL remains an interesting open problem.

6. THREATS TO VALIDITY

The main threat to the external validity of our findings is that they are based on two languages and two operating systems only. On the other hand, both are large independently developed real-world projects, with different objectives: Linux is a general purpose kernel and eCos is an entire specialized RTOS for embedded systems. We believe that other related domains, especially embedded RT such as automotive and avionic control software, will share many characteristics with the studied systems. Further, comparison to other feature modeling languages, shows that both are representative of the space of feature modeling.

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.

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 future work. We only examined one architecture per OS; however, both architectures represent large and mature portions of the systems: Linux's x86 architecture covers 61% of the total of 10415 features and 67% of the total of 8M SLOC; the eCos's i386PC covers 44% of the total of 2859 features and 33% of the total of 0.9M SLOC.

An internal threat is that our statistics are incorrect. To reduce this risk, we instrumented the native tools to gather the statistics rather than building our own parsers. We thoroughly tested our 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. We manually confirmed that all of them are indeed dead, either because they depended on features from another architecture or they were intentionally deactivated.

7. RELATED WORK

Semantics of academic variability modeling languages were studied before [17, 5]. We focus on languages originating from practice. A survey on the use of feature models [8] identified only five papers reporting practical experience. References [14,16,17] in [8] are experiences from researchers applying feature modeling to sample problems from industry. References [31,37] therein are self-reported industry experiences: the first on using 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. A notable exception is the report on the industrial use of Dopler for variability modeling and product derivation [7]; Sadly, neither the models nor data are available.

We reported early findings on the Linux model in a previous workshop paper [19]; however, the present paper differs significantly. The previous work was to extract a FODA feature model from Linux and compare it with feature models from research papers. The present work compares two languages and models in their full richness (beyond FODA), including their formal semantics and a different set of statistics. The resulting findings (Sections 2–4) are new. We also studied the evolution of the Linux model [11], showing that the number of dependencies has grown proportionally to the number of features over the last five years.

Table 1 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 [3] and pure::variants do not support them. Computed defaults were proposed by researchers [4], 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* [5], it shares many characteristics with feature modeling. The connection between Kconfig and feature modeling was made in [20]. 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 [15]. A variety of reasoners have been used to create feature model analyzers and configurators, including CSP solvers [23], SAT solvers [21, 13], and BDD packages [12]. 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.

8. CONCLUSION

Our study provides empirical evidence for the use of variability modeling in real-world large-scale systems. The study confirms that feature modeling concepts from FODA are used in practice; however, it shows that more advanced concepts, such as visibility conditions, derived features, and binding mode are also needed. Our language comparison showed intricate semantic interactions among the advanced concepts, deepening our understanding of such languages. We also identified significant limitations of existing configurators—a call to arms for future research. We believe our findings will be of interest to variability modeling language and tool designers.

9. REFERENCES

[1] D. Benavides, S. Segura, and A. Ruiz-Cortes. Automated analysis of feature models 20 years later: a literature review. *Information Systems*, 2010.

[2] T. Berger and S. She. Formal semantics of the CDL language. Technical Note. Available at www.informatik.uni-leipzig.de/~berger/cdl_semantics.pdf.

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

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

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

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

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

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

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

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

[11] R. Lotufo, S. She, T. Berger, K. Czarnecki, and A. Wąsowski. Evolution of the Linux kernel variability model. In *SPLC*, 2010.

[12] M. Mendonca, A. Wąsowski, K. Czarnecki, and D. D. Cowan. Efficient compilation techniques for large scale feature models. In *GPCE*, 2008.

[13] M. Mendonça, A. Wąsowski, and K. Czarnecki. SAT-based analysis of feature models is easy. In *SPLC*, 2009.

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

[15] 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*, 52(3), 2010.

[16] M. Riebisch, K. Böllert, D. Streitferdt, and I. Philippow. Extending feature diagrams with UML multiplicities.

[17] P.-Y. Schobbens, P. Heymans, J.-C. Trigaux, and Y. Bontemps. Generic semantics of feature diagrams. *Comput. Netw.*, 51(2):456–479, 2007.

[18] S. She and T. Berger. Formal semantics of the Kconfig language. Technical Note. Available at eng.uwaterloo.ca/~shshe/kconfig_semantics.pdf.

[19] S. She, R. Lotufo, T. Berger, A. Wąsowski, and K. Czarnecki. The variability model of the Linux kernel. In *VaMoS*, 2010.

[20] J. Sincero and W. Schröder-Preikschat. The Linux kernel configurator as a feature modeling tool. In *SPLC-ASPL*, 2008.

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

[22] 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.

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

[24] R. Zippel and contributors. kconfig-language.txt. available in the kernel tree at kernel.org, seen 2009-11/23.