The Variability Model of The Linux Kernel

Steven She*, Rafael Lotufo*, Thorsten Berger†*, Andrzej Wąsowski‡, Krzysztof Czarnecki*
*University of Waterloo, Canada, {shshe, rlotufo, kczarnec}@gsd.uwaterloo.ca
†University of Leipzig, Germany, berger@informatik.uni-leipzig.de
‡IT University of Copenhagen, Denmark, wasowski@itu.dk

***Abstract*—Lack of realistic benchmarks hinders efficient design and evaluation of analysis techniques for feature models. We extract a variability model from the code base of the Linux kernel, obtaining a model larger by an order of magnitude than the largest publicly available feature model so far. We analyze properties of this model, compare it with previously available benchmarks, and emphasize the differences from published academic examples. As a result, we broaden our understanding of what a feature model is, hopefully challenging tool designers by providing an interesting benchmark, giving input to design of random model generators, and last but not least, inspiring designers of variability modeling languages.**

I. Introduction

Reports from tool vendors and users in the series of proceedings of the Software Products Lines conference witness a broad industrial interest and experience in using variability modeling. Nevertheless, many researchers feel that realistic benchmarks for evaluating variability modeling tools are inaccessible [1]. Many variability models *are* in fact available already (see www.splot-research.org, fm.gsdlab.org), however very few of them originate from realistic processes; most are small examples from research publications or outcomes of student run case studies. Given the lack of realistic large-scale models, many authors resort to using randomly generated models [2], [3], [4].

In order to help the community mitigate this limitation, we bring a large and realistic variability model, extracted from the build system of the Linux kernel. Linux has an explicit variability specification expressed in the Kconfig language [5]—a language developed specifically for this purpose by the kernel developers. The Kconfig model so closely resembles feature models [6], [7] that it can be naturally interpreted as one [8].

We present the Kconfig model—available online in a feature modeling friendly format (see fm.gsdlab.org). We detail the model transformation from the Kconfig language to feature model. Foremostly, we extensively study the properties of the Kconfig model, contrasting it to metrics of a corpus of published feature models.

We intend to attract the attention of designers of feature modeling tools to our result, so they can use the Kconfig model as a particularly tough benchmark and its characteristics as a source of requirements on tools. Some would also be interested to know that randomly generated models used in previous works are likely much easier to analyze than the Linux model. Last but not the least, the Kconfig notation, together with the size and structure of the Linux model, should inspire designers of variability modeling languages, in particular when it comes to support for modularity and user interface aspects like visibility of features.

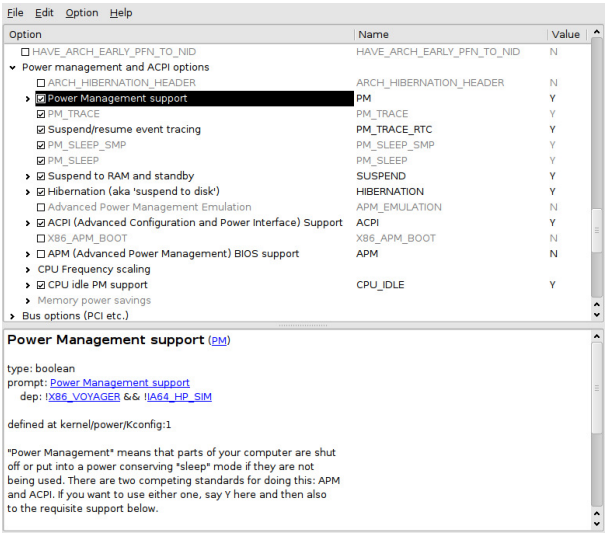


Fig. 1. The xconfig configurator GUI

II. The Kconfig Language

Configuration options are known as *configs* in Linux. Kconfig is the language used to specify the available configs and dependencies among them: configs can be nested under other configs; they can also be grouped under *menus* and *choice groups*. The kernel configurator, xconfig, renders the Kconfig model as a tree of options, which users select to specify configurations to be built. Cross-tree dependencies, if any, are indicated in the bottom part of xconfig's GUI (Fig. 1).

Figure 2 shows a fragment of the Linux variability model in the Kconfig language. The fragment contains a menu with four Boolean configs as children.

Configs are named parameters with a specified value type: Boolean, tristate, integer (**`int`** or **`hex`**), or string. Boolean configs represent options that can be switched on (`y`) or off (`n`). Tristate configs are similar, except that they have two 'on' states: `y` indicates that the code implementing the option should be linked into the kernel statically, whereas `m` indicates that it should be compiled as a dynamically loadable module. Thus, tristate is a simple form of a *binding mode* specification [7]. We refer to Boolean and tristate configs collectively as *switch configs*, since they appear in xconfig as switches (e.g., checkboxes). Integer configs are used to specify

```
menu "Power management and ACPI options"
  depends on !X86_VOYAGER
  config PM
    bool "Power Management support"
    depends on !IA64_HP_SIM
    ---help---
        "Power Management" means that ...
  config PM_DEBUG
    bool "Power Management Debug Support"
    depends on PM
  config CPU_IDLE
    bool "CPU idle PM support"
    default ACPI
  config PM_SLEEP
    bool
    depends on SUSPEND || HIBERNATION ||
               XEN_SAVE_RESTORE
    default y
...
endmenu
```

Fig. 2. Fragment of a Kconfig model

numerical options such as buffer sizes. String configs are used to specify names of, for example, files or disk partitions. We refer to integer and string configs as *entry-field configs*, since the configuration tool shows them as editable fields.

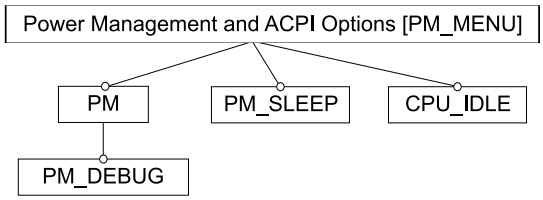
Config definitions can include other elements besides type. If the type is followed by a *prompt*, i.e., a short explanation text shown to the user in xconfig, the config is *user-selectable*; otherwise it is not. A config can have a *visibility condition* directly following the prompt (not shown in the example). A longer *help* text can also be provided (see the PM entry in Figure 2).

A *depends-on* clause introduces a dependency that must be satisfied when selecting the config. In the example, PM can only be selected if IA64_HP_SIM is not selected. Reversely, a *select* clause (not shown) enforces immediate selection of another config when this config is selected by the user. Depends-on is also used to specify nesting, when referencing its previous config: for example PM_DEBUG is nested under PM.

A *default* clause has a two-fold effect. First, if the config is user-selectable, *default* is used to provide an initial value, which can still be overridden by the user. For example, CPU_IDLE takes the same value as ACPI by default. If the config is not user-selectable, then the default enforces a value for the feature, effectively defining a cross-tree constraint. For example, PM_SLEEP is non-user-selectable and set to y if its depends-on condition holds; otherwise it is set to n.

Menus are used for grouping. Kconfig provides a conditional visibility mechanism for menus. We call conditionally visible menus simply *conditional menus*; if their condition, a cross-tree constraint, is false, they and their children are grayed out in the configurator.

Finally, *choices* (not shown) group configs into alternatives, which we call *choice configs*. Choices themselves can be Boolean or tristate. When the choice state is y, the choice configs underneath are constrained with XOR; when the value



$$PM\_MENU \rightarrow \neg X86\_VOYAGER$$
$$PM \rightarrow \neg IA64\_HP\_SIM$$
$$PM\_SLEEP \leftrightarrow SUSPEND \,||\, HIBERNATION \,||\, \ldots$$

Fig. 3. Feature model for the example of Fig. 2

is m, the configs are constrained with OR. A choice marked as *optional* can be set to n and no choice config needs to be selected. Choices without the mark are *mandatory* and cannot be set to n. Tristate choices reflect a common binding variation: static binding requires an exclusive module to be linked statically; dynamic binding allows multiple alternative modules to be compiled—a single module is loaded at runtime.

Kconfig as a Feature Modeling Notation. We will interpret the hierarchy of configs, menus, and choices as the *Linux feature model*. Table I maps Kconfig concepts to feature modeling concepts. Figure 3 shows the feature model for the Kconfig example of Figure 2 generated by this mapping.

Switch configs map to optional features (Table I); each feature for a tristate config additionally has an attribute of type bmode, defined as enum {y,m}. An entry-field config maps to a mandatory feature with an attribute of an appropriate type, integer or string.

We map conditional menus to optional features; a more faithful translation would require extending the feature modeling notation with visibility conditions. Unconditional menus map to mandatory features.

We map a choice to a feature with a group underneath containing the grouped features representing the choice configs. A mandatory choice maps to a mandatory feature with an XOR-

TABLE I
MAPPING KCONFIG MODELS TO FEATURE MODELS

Kconfig concepts		Feature modeling concepts
Switch config		Optional feature
Entry-field config		Mandatory feature
Conditional menu		Optional feature
Unconditional menu		Mandatory feature
Choice		
Mandatory	➠	Mandatory feat. + XOR-group
Optional		Optional feat. + XOR-group
Mandatory tristate		Mandatory feat. + OR-group
Optional tristate		Optional feat. + OR-group
Choice config		Grouped feature
Config, menu or choice nesting	➠	Sub-feature relation
Visibility conditions Selects Constraining defaults	➠	Cross-tree constraint

group. An optional choice maps to an optional feature with an XOR-group. In a slightly lossy manner, we map a tristate choice to a feature with the less restrictive OR-group.

III. The Linux Variability Study

We analyzed aspects of features, hierarchy, constraints, and natural-language content in the Linux model both quantitatively and qualitatively. When applicable, we applied the same analysis to a corpus of published models and compared the results with Linux.

Statistics gathering for Linux. Our statistics were gathered for the 2.6.28.6 version of the kernel. The Kconfig model was extracted from a normalized form using the kernel configurator code. Prior to gathering the statistics, we apply a post-processing stage to account for some special cases in the Kconfig model (e.g. there were 11 multiply-defined features in the hierarchy).

Corpus of published models. The corpus contains 32 models available from the SPLOT website [9] (see Appendix A). Most of these models are academic examples; from workshop and conference publications and MSc and PhD theses. They span many domains, including insurance, entertainment, web applications, home automation, search engines, and databases. Nineteen models represent software product lines; eight represent other types of product lines, e.g., hardware or business; and five represent entire domains, e.g., electronic commerce systems. Only five models describe real, existing software systems, however even these appears to be results of research efforts as opposed to regular industrial engineering practice.

To analyze the published models, we created a simple tool, reusing Mendonca's parser for SPLOT's XML-based feature model format (http://www.splot-research.org/sxfm.html).

A. Characterization of Features

Feature and group statistics. Table II gives statistics for both Kconfig concepts and their feature modeling counterparts. The Linux model has 5426 features, which is an order of magnitude more than *Electronic Shopping*, the largest of the published models, with 287 features. The vast majority (4744 or 89%) are user-selectable. Only about about 3% (188) of features have integer and String attributes. Note that 71

TABLE II
Linux feature and group statistics

Kconfig Concept	Features	Mand.	Grouped	XOR + OR
Config	5323	0	146	0
Non / User-Sel.	547 + 4744			
Boolean	2005	0	136	0
Tristate	3130	0	10	0
Int	132	132	0	0
Hex	29	29	0	0
String	27	27	0	0
Menu	71	38	0	0
Choice	32	31	0	30 + 2
Total	**5426**	257	146	30 + 2

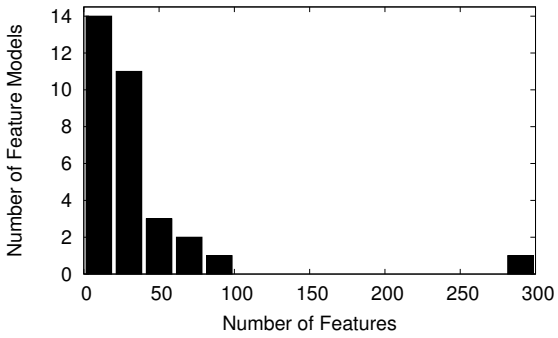


Fig. 4. Sizes of the published models

features correspond to menus, and 38 of these features are mandatory. Choices contributed 32 features and 32 groups; 30 of them are XOR-groups; the two OR-groups were contributed by two tristate choices.

Compared to published models (Figure 4, Table III) the Linux model contains very small percentages of mandatory features (5%), grouped features (3%), and groups (1%; relative to the number of features). Thus, the Linux features are mostly optional (92%).

Code-granularity of features. In order to assess the code-granularity of an average feature in Linux, we computed the set of source files included in the *allno* and in the *allyes* configurations. *Allno* tries to approximate the smallest possible configuration of Linux kernel, while *allyes* tries to approximate the largest configuration (both are included as targets in the build system and are based on a very simple algorithm). Table IV reports that a mere 61 user-selectable features are included in the former, and 3448 in the latter (which is about 73% of all user-selectable features). Moreover, all features except one of *allno* are also features of *allyes*. We used the difference between the sizes of the two configurations to compute an average feature size. An average feature spans 2.76

TABLE III
Published models vs. Linux

% relative to no. features	**Published Models** median, min - max	**Linux**
mandatory features	25%, 0% - 66%	4.74%
grouped features	44%, 0% - 75%	2.69%
groups	16%, 0% - 35%	0.59%
XOR	9%, 0% - 30%	0.55%
OR	6%, 0% - 16%	0.04%
cross-tree constraint ratio	19%, 0% - 62%	82%

TABLE IV
Configuration statistics

Metric	**allno**	**allyes**	Δ	θ
Features	61	3,448	3387	1
Files	973	10,326	9,353	2.76
SLOC	210,302	4,266,171	4,055,869	1,197.48

$\Delta_i = \textbf{allyes}_i - \textbf{allno}_i; \theta_i = \Delta_i / \Delta_1$

source files, and roughly a thousand non-blank non-comment lines of code (although surprisingly small, this number is still an over-approximation of the actual average, as it ignores the fact that lines not belonging to *allyes* are removed by the preprocessor).

Qualitative characteristics. To understand types of features and options in the Linux kernel, we performed a subjective categorization of 180 randomly selected features. The selected categories characterize the granularity of features from the user's point of view—whether a feature enables support for a device or its option—as well as their type—whether the feature is related to a driver, protocol, API, etc.

We categorized features based on the help descriptions provided in the Kconfig files and by querying the web when needed. We left features with no description information in Kconfig uncategorized. As this type of classification is subjective, we performed a sanity check by cross checking the categories for 18 features. We found a discrepancy of only 8% and so believe that the categorization is sound and relevant. We used the following user-based granularity categories:

- *Menu:* grouping features, e.g., IO_SCHEDULERS, which groups read/write schedulers for block devices;
- *Support:* features that support certain devices or protocols, e.g., HID_SAMSUNG, which enables support for Samsung's InfraRed remote control;
- *Option:* features enabling/disabling a specific kernel or driver capability, e.g., DASD, which enables DASD block devices using S/390s channel commands;
- *Debug:* features enabling tracing and other debug functions for developers, like BOOT_TRACER, which activates collection of run-time informations to assist boot time optimization; or MEMSTICK_DEBUG, which activates debug info for memory stick devices.

Additionally, we categorized features into the following *types*:

- *API:* features that provide an API for programming, e.g., CRYPTO_CTR, which enables API for the block cipher algorithm, required for IPSec;
- *Driver:* features related to drivers, e.g., SND_ADLIB, that enables support for AdLib cards;
- *Kernel:* change kernel behaviour, e.g., FAILSLAB, which enables fault-injection capability for kmalloc;

TABLE V
USER-BASED GRANULARITY CATEGORIES OF LINUX FEATURES

Menu	Support	Option	Debug	No Info
1	97	46	13	23

TABLE VI
TYPES OF FEATURES IN LINUX KERNEL

API	Driver	Kernel	Protocol	Subsystem	No Info
5	120	15	14	1	25

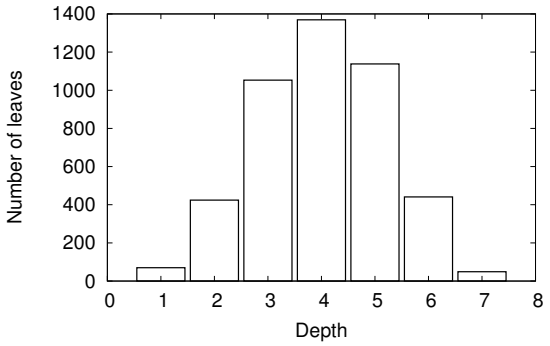


Fig. 5. Linux leaf depth

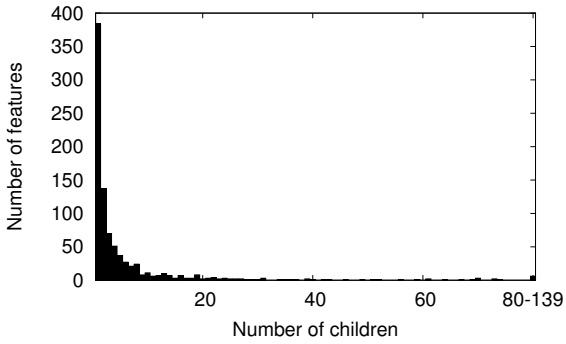


Fig. 6. Linux branching factor

- *Protocol:* features that implement protocols, e.g., LLC2, which enables support for PF_LLC sockets;
- *Subsystem:* features that enable whole subsystems, e.g., Bluetooth, BT, which enables the Bluetooth subsystem, comprising of several layers of software.

We found a very high correlation between support features and drivers, meaning that roughly 50% of features in the Linux kernel are drivers that support certain devices and protocols, greatly outnumbering features that enable smaller-grained functionality.

Interesting is also the relatively high number of the developer-oriented debug features.

B. Model Hierarchy

Figure 5 shows the number of leaves with a given depth for Linux. The maximum leaf depth in the Linux model is 7 (we assume level 0 for root). The maximum depth for published models is 10. The shapes of the leaf-depth histogram for the publish models vary significantly; however, the shape for the largest published model, *Electronic Shopping*, has closest resemblance to that of Linux. An interesting observation is that the Linux model is relatively shallow (average depth of 4).

Figure 6 shows the number of features for a given *branching factor*, i.e., number of children, for Linux. The vast majority of features are leaves (4544; not shown in the histogram). Surprisingly, as many as 384 features are single-child parents;

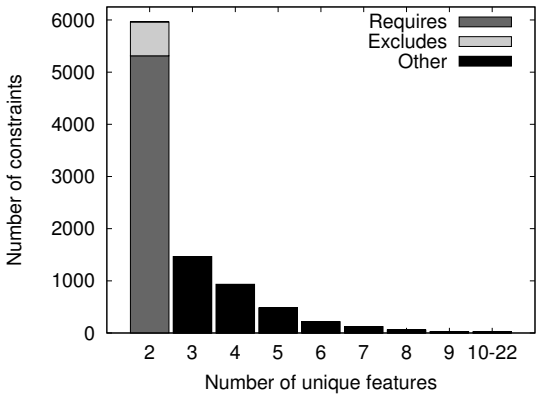


Fig. 7. Constraints categorized by the number of features they reference

these include features representing choices and singular options. The histogram has a long tail: 58 features have between 20 and 139 children. In contrast, the published models have maximum branching of 11, which is vastly smaller than for Linux. Interestingly, when generating models, White et al. [2] assume a branching factor of at most 5, which appears very low in the face of our data.

As before, histogram shapes of the published models vary significantly, with the shape for Electronic Shopping resembling most closely that of Linux. A notable difference, when compared to Linux, is the relatively low number of single-child parents in Electronic Shopping, where it is far lower than the number of two-child parents.

C. Constraints

A *constraint* restricts legal combinations of features. Obviously, hierarchical dependencies, discussed in the previous, section are constraints. Not all dependencies can be expressed as hierarchical dependencies, however. Additional *cross-tree constraints*, specified in a suitable logic, are typically added besides the nesting structure. The Kconfig language includes a language of boolean expressions extended with equality predicates on non-boolean values for this purpose.

We determined three sources of cross-tree constraints in the Kconfig model: *visibility conditions*, *select clauses*, and *constraining defaults* (see Section II). The excerpt from Fig. 2 maps to the cross-tree constraints shown under the diagram in Fig. 3. The constraint on PM_DEBUG is not a cross-tree constraint—it belongs to the hierarchy. Also, the default in CPU_IDLE does not give rise to any cross-tree constraints, since it is not constraining itself, but merely proposes a default value that can be overridden.

As Mendonca [4] points out, the hardness of analysis of feature models lies in the complexity of their cross-tree constraints. At the same time, it is not uncommon that researchers believe that cross-tree constraints are rare and not crucial to feature modeling. For example in our reference corpus, eleven models do not have *any* cross tree constraints, and a further eleven only have one or two constraints. Only 37% of models in our sample have a significant amount of cross-tree dependencies. In the Kconfig model, a total of 4186 (77%) of features declare a constraint in their definitions, sometimes more than one, giving rise to a total of 9291 constraints in the feature-modeling sense (top-level conjuncts).

It is interesting to see how these conjuncts distribute over more standard types of constraints (see Figure 7). 5313 (89%) of them are positive implications (also known as *requires* constraints), and 649 (11%) are *excludes* constraints (i.e. constraints of the form $f \rightarrow \neg g$). 3324 constraints represent more complex relations involving more than two features. The most complex of these includes 22 distinct literals in one constraint:

$$(\text{MWINCHIP3D} \vee \text{MCRUSOE} \vee \text{MEFFICEON} \vee \text{MCYRIXIII} \vee \text{MK7} \vee \text{MK6} \vee \text{MPENTIUM4} \vee \text{MPENTIUMM} \vee \text{MPENTIUMIII} \vee \text{MPENTIUMII} \vee \text{M686} \vee \text{M586MMX} \vee \text{M586TSC} \vee \text{MK8} \vee \text{MVIAC3\_2} \vee \text{MVIAC7} \vee \text{MGEODEGX1} \vee \text{MGEODE\_LX} \vee \text{MCORE2}) \wedge \neg\text{X86\_NUMAQ}) \vee \text{X86\_64}) \rightarrow \text{X86\_TSC} = \text{y}$$

In contrast, published models almost exclusively contain binary constraints. Only 4 (12%) models in the sample contain a constraint relating 3 features, and none contained larger constraints. We should mention here that for analysis tools, binary constraints are easy. Consistency checking of a 2-CNF formula can be done in polynomial time, while consistency checking for a formula with ternary clauses is NP-complete. Thus the Linux model sets a new challenge for analysis tools.

Not only does the Linux kernel model contains a large number of constraints, but these constraints also involve unusually many features. Mendonca [4], [10] introduces *cross-tree constraints ratio* (CTCR)—a normalized measure comparing the number of features participating in cross-tree constraints (more precisely a percentage of features in the model that are referenced in constraints). As we can see in Table III, CTCR for published models varies between 0 and 60%, with 19% being a typical value. In the Linux model, all 82% of features participate in cross tree constraints. Effectively the Linux hierarchy says relatively little about the combinatorial dependencies between features, indicating a certain limitation of hierarchical models for describing dependencies in very complex software projects.

D. Natural Language Properties

Natural language processing techniques gain popularity in software engineering tools, including feature modeling [11], [12]. It is thus relevant to investigate the main properties of texts available in the Kconfig model. The Linux model has three kinds of textual attributes: feature identifiers (like PM), prompt texts (Power management support), and descriptions (`"Power Management" means that ...`).

Available Textual Information. We have counted the length of feature identifiers (considering strings separated by underscores as separate words), of prompts, and of descriptions. Table VII summarizes the findings. The majority of identifiers are 13 characters long, but there do exist some of length 2 (such as MD, SX, VT), and some of length 43

(SECURITY_SELINUX_POLICYDB_VERSION_MAX_VALUE). The majority of identifiers contains two or three words, with some approaching as many as nine. Most prompts include 3–5 words, with some reaching up to 13.

The help descriptions exhibit a similar pattern. The majority is around 30 words long, but some reach as much as 392 words. At the same time 823 features have no descriptions at all. Out of these, 540 are non-user-selectable, and 102 are menus or choices. As many as 181 of user-selectable configs contain no descriptions.

These numbers demonstrate a considerable effort of kernel developers to make features descriptions valuable for users. Still, about 20% of features have poor data like unhelpful identifiers or descriptions under 20 words. Most short descriptions are not very explanatory, containing texts such as *Say Y here* or *If unsure say N* or just the full name of the supported device. Longer descriptions have detailed explanations, such as when to enable the given feature and suggestions of other related features to enable or disable.

Vocabulary. We analyzed the most frequent domain specific terms in the Linux model. We consider any word not included in the *aspell* (0.60.5) English dictionary to be a domain term. Table VIII shows the most frequent domain words for the text attributes. Most common terms clearly relate to popular hardware kinds or to kernel subsystems.

Moreover, identifiers of 3601 features share common words with identifiers of their immediate parents (not necessarily domain specific words). For example: CRYPTO_DEV_HIFN_795X_RNG is a child of CRYPTO_DEV_HIFN_795X, or DVB_USB_DIBUSB_MB_FAULTY is a child of DVB_USB_DIBUSB_MB. Consequently, for about half of the features, it should be possible to automatically determine their immediate parents using a string similarity metric.

IV. Threats to Validity

External Validity. Our baseline corpus is comprised of 32 models selected from published sources. Due to their academic origin, these models are not realistic. However, they are a suitable baseline here, as they support our main point that there exist realistic models that do not share characteristics with published models.

Studying the Linux kernel as a single subject raises a doubt whether the reported values of metrics are representative. However, we do not make any general claims about feature models based on these metrics. Rather we postulate that this model, which describes a realistic and wide-spread software system, should be included in benchmark sets of feature models. While a single model cannot be claimed to be representative, it *does* witness a departure from expected characteristics. Moreover, as Tartler [13] suggests, it is unlikely that Linux is an exception among operating systems, since other have similar qualities.

Arguably, the Kconfig model has not been created as a feature model but rather as a specification of a configuration system. Our interpretation of this model as a feature model can be seen as a violation of the original intention. Nevertheless Kconfig shares structural characteristics with feature models, and its main purpose (modeling a range of configuration choices) is consistent with the main objective of feature modeling. Unlike typical feature models, which are created during domain analysis, the Kconfig model has been grown bottom up—by adding features during evolution of the system. While these two processes, domain analysis and evolution—*are* different—the evolutionary bottom up construction is a realistic scenario, resembling the processes during evolution of mature product lines.

Internal Validity. The qualitative classification of Linux features has been done manually by one member of the project team for a randomly selected subset of 180 features. In order to ensure the representativeness of this result we have selected the 180 feature sample uniformly; in the sense that every feature in the kernel had equal probability to be selected. For a subsample of uniformly selected 18 features (out of the 180) we have independently verified the classification using another member of the project.

No formal definition of the Kconfig language is available, besides the only existing implementation itself. Thus, we could not verify whether our transformation of the Kconfig model to the feature modeling notation is semantics preserving. In order to decrease the chance of misinterpretations, we have used the original Linux configurator code to perform the first phase of that translation consisting of normalizing the representation and removing any syntactic sugar. Thus, we are confident that the first phase of the translation is in agreement with whatever the kernel developers have intended. For the remaining part of translation usual best practice measures were applied, such like investigating test cases using our translator against the Linux configurator tool.

Because there is no single accepted canonical syntactic representation for constraint systems, counting constraints is always subject to applied translations. In our case, however, it is very clear that regardless of which of the reasonable translations was used, the complexity of constraints in the Linux model is much higher than any constraints in the

TABLE VII
Size of textual artifacts in the Kconfig model

artifact	no. of characters			no. of words		
	median	min	max	median	min	max
identifiers	13	2	58	2	1	9
prompts	27	2	82	4	1	13
description	-	-	-	29	2	392

TABLE VIII
Top domain terms in the Kconfig model

Text source	Most frequent domain terms					
Identifier	*usb*	*snd*	*md*	*serial*	*fb*	*debug*
Prompt	*usb*	*ethernet*	*pci*	*intel*	*scsi*	*pcmcia*
Description	*usb*	*linux*	*scsi*	*ethernet*	*pci*	*howto*

published models.

V. Related Work

The connection between the Kconfig language and feature modeling was previously described by Sincero and Schröder-Preikschat [14]. In their paper, they describe feature modeling concepts in terms of Kconfig constructs. Our work differs in that we investigate the inverse mapping—from Kconfig to feature model.

Tartler et al. [13] propose an architecture for a tool to detect dead features in the Linux Kernel. They make the same assumption as we do, i.e. that Kconfig can be naturally and meaningfully interpreted as a feature model. However, they are not interested in the properties of the model itself, except for existence of dead features. They extend a classic feature model analysis (dead feature detection) to the mapping of features to code to diagnose quality errors in the Linux codebase. It will be interesting to see if their analysis actually scales to the model we have described here.

Segura and Ruiz–Cortés [1] complain about the lack of realistic benchmarks for feature model analysis tools, and postulate creating a common standardized benchmark set with a repository of realistic models. Our contribution can be seen as one step towards addressing their concern, by providing a large and realistic feature model, which is now available online.

Randomly generated benchmarks have been applied in evaluation of analyzes tools for feature models [3], [2], [4], [10]. The generators applied in these projects have been tuned to generate models that resemble published models. However these characteristics turn out not to be true for large real life models, as exemplified by the Linux model described in this paper. Our results can serve as an input to design a statistically significant generator of random benchmarks for tools.

The European Project AMPLE has investigated use of automatic information retrieval techniques for extraction of feature models from requirement documents [11], [12]. The applicability and tuning of such algorithms to practical models heavily depends on properties of documentation available such as the size of provided descriptions and the amount of domain specific terms in them. The statistics of Section III-D are hopefully inspiring for this purpose.

Earlier we have presented reverse engineering algorithms that can be used to obtain examples of feature models from systems, where no explicit model is embedded—directly from compatibility constraints [15], or from sets of legal configurations [16].

VI. Conclusion

The Linux model is an excellent example of a large-scale variability model used in practice. We have shown that it challenges many of the long-held assumptions in the product-line community. It offers a wealth of information and acts as an excellent benchmark for the evaluation of automated analysis tools.

References

[1] S. Segura and A. Ruiz-Cortés, "Benchmarking on the automated analyses of feature models: A preliminary roadmap," in *VaMoS*, 2009, pp. 9–17.

[2] J. White, D. C. Schmidt, D. Benavides, P. Trinidad, and A. R. Cortés, "Automated diagnosis of product-line configuration errors in feature models," in *SPLC*. IEEE Computer Society, 2008, pp. 225–234.

[3] P. Trinidad, D. Benavides, A. R. Cortés, S. Segura, and A. Jimenez, "Fama framework," in *SPLC*. IEEE Computer Society, 2008, p. 359.

[4] M. Mendonca, A. Wąsowski, and K. Czarnecki, "Sat-based analysis of feature models is easy," in *SPLC'09*. IEEE, 2009.

[5] R. Zippel and numerous contributors, "kconfig-language.txt," seen 2009-11-23.

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

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

[8] J. Sincero, H. Schirmeier, W. Schröder-Preikschat, and O. Spinczyk, "Is The Linux Kernel a Software Product Line?" in *Workshop SPLC-OSSPL 2007*, 2007.

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

[10] M. Mendonca, A. Wasowski, K. Czarnecki, and D. D. Cowan, "Efficient compilation techniques for large scale feature models," in *GPCE'08*, 2008, pp. 13–22.

[11] N. Weston, R. Chitchyan, and A. Rashid, "A framework for constructing semantically composeable feature models from natural language requirements," in *SPLC*. IEEE Computer Society, 2009.

[12] V. Alves, C. Schwanninger, L. Barbosa, A. Rashid, P. Sawyer, P. Rayson, C. Pohl, and A. Rummler, "An exploratory study of information retrieval techniques in domain analysis," in *SPLC*. IEEE Computer Society, 2008, pp. 67–76.

[13] R. Tartler, J. Sincero, W. Schröder-Preikschat, and D. Lohmann, "Dead or alive: Finding zombie features in the linux kernel," in *FOSD*, 2009, pp. 81–86.

[14] J. Sincero and W. Schröder-Preikschat, "The linux kernel configurator as a feature modeling tool," in *Proceedings of the Workshop on Analyses of Software Product Lines (ASPL)*, 2008, pp. 257–260.

[15] K. Czarnecki and A. Wąsowski, "Feature models and logics: There and back again," in *SPLC '07*. IEEE, 2007.

[16] K. Czarnecki, S. She, and A. Wąsowski, "Sample spaces and feature models: There and back again," in *SPLC'08*. IEEE, 2008.

[17] *Software Product Lines, 12th International Conference, SPLC 2008, Limerick, Ireland, September 8-12, 2008, Proceedings*. IEEE Computer Society, 2008.

Appendix A
Corpus of published models

The corpus includes 32 models downloaded from the SPLOT website [9] on 16 November 2009:

Aircraft PL, Arcade Game PL, Car PL, Cellphone, CFDP Library, Connector PL, Digital Video System, Documentation_Generation, Electronic Shopping, FAME-DBMS, Graph, Graph Product Line, HIS, Insurance Policy, Insurance_Product, Inventory, James, JPlug, Key_Word_In_Context_index_systems, Model_Transformation, Monitor_Engine_System, MoviesApp PL, SAL, Search_Engine_PL, Smart Home, Stack PL, Telecommunication_System, Text_Editor, Thread, Virtual_Office_of_the_Future, Weather Station, Web_Portal