Dead or Alive: Finding Zombie Features in the Linux Kernel[*]

Reinhard Tartler, Julio Sincero, Wolfgang Schröder-Preikschat, Daniel Lohmann
{tartler, sincero, wosch, lohmann}@cs.fau.de

Friedrich-Alexander-University Erlangen-Nuremberg

ABSTRACT
Variability management in operating systems is an error-prone and tedious task. This is especially true for the Linux operating system, which provides a specialized tool called *Kconfig* for users to customize kernels from an impressive amount of selectable features. However, the lack of a dedicated tool for kernel developers leads to inconsistencies between the implementation and the variant model described by *Kconfig*. This results in real bugs like features that cannot be either enabled or disabled at all; the so called *zombie* features.

For both in the implementation and the variant model, these inconsistencies can be categorized in *referential* and *semantic* problems. We therefore propose a tool approach to check the variability described by conditional compilation in the implementation with the variant model for both kinds of consistency. Our analysis of the variation points show that our approach is feasible for the amount of variability found in the Linux kernel.

1. INTRODUCTION
Operating systems provide no business value of their own. Their sole purpose is to ease the development and execution of applications on some hardware platform, that is, to serve application developers and application users with a virtual machine layer that provides the "right" set of features for *their* particular problem domain. Of course, this pays off only if the operating-system itself is *reusable* for many different applications and hardware platforms. Historically, this led to the idea of configurable *system families*, in which each family member subsumes a particular set of features. In fact, much of the work of the software-engineering pioneer's from the 70s was motivated by practical problems that stem from the feature variability of configurable operating-system families [5, 15, 6].

Today, the number of configurable features offered by many operating systems is still an order of magnitude higher than the variability we typically find in software product lines and software families from other domains: The feature model of the (very small) PURE embedded operating system, for instance, already offers more than 250 configurable features [2]; eCos [13], which targets the same domain, provides more than 750 features. However, the Linux kernel, which is the subject of this paper, can be configured by even more than 8000 (!) configuration options [17].

It should be clear that both, kernel hackers and end users, have to be supported by extra means and tools for *variability management* to handle this impressive amount of features. With the graphical configuration editors build around the *Kconfig* tool, end-user support is already very reasonable. For kernel hackers, however, variability management is an error-prone and tedious task. The result are bugs and *zombie*-features that are still presented in *Kconfig*, but not in the code or vice versa.

1.1 Variability Management in Linux
The variability management techniques employed in the Linux kernel can be divided into three levels:

Model Level. The *Kconfig* tool set was especially written to support the modeling of features and interdependencies of the Linux kernel. It provides a language to describe a variant model consisting of features (referred to as *config options*) together with their constraints and dependencies. Modularization of the variant model is supported by an inclusion mechanism. In Linux kernel version `2.6.30`, a total of 534 `Kconfig` files are employed, consisting of 88,112 lines of code that describe 8063 features and their dependencies. In many respects, the resulting variant model can be compared to feature models known by the software product-line community [18].
The user configures a Linux kernel by selecting features from this model. During the selection process, the *Kconfig* configuration utility implicitly enforces all dependencies and constraints, so that the outcome is always the description of a valid variant. Technically, this description is given as a C-style header file that de-

[*]This work was partly supported by the German Research Council (DFG) under grants no. SCHR 603/7-1 and SCHR 603/4

fines a `CONFIG_xxx` preprocessor macro for every selected feature.

Generation Level. Coarse-grained variability is implemented on the generation level. The compilation process in Linux is controlled by a set of custom scripts called *Kbuild* that interpret a subset of the `CONFIG_xxx` flags and drive the compilation process by selecting which compilation units should be compiled into the kernel, compiled as a loadable module, or not compiled at all.

Source Code Level. Fine-grained variability is implemented by conditional compilation using the C preprocessor. The source code is annotated with preprocessor directives (like `#ifdef CONFIG_xxx` or `#if(CONFIG_xxx ...)`), which are evaluated in the compilation process. This is the major variability mechanism used in the Linux kernel.

Whereas in other domains the set of features is usually the outcome of a top-down domain analysis (*requirement-motivated features*), the majority of features we find in configurable operating systems are usually the result of a bottom-up design and implementation process, beginning on the level of hardware-abstractions up to the kernel APIs (*implementation-motivated features*). This is particularly true in Linux, which is a very source-code–centric project. New or improved features are implemented first; later on they are assigned a `CONFIG_xxx` symbol which is, together with their dependencies, integrated into or updated in the *Kconfig* model. This, however, is a manual and tedious task that nevertheless requires the skills of an experienced kernel hacker.

1.2 Problem Statement

It is not difficult to imagine that this process leads to inconsistencies between the *Kconfig* representation of features and their dependencies as seen by the user who configures a Linux kernel, and the implementation.

It is clear that the *Kconfig* tool cannot solve variability management problems satisfactorily for kernel developers, as it has no access to the *use* of its *config options* in the code base. Likewise, the C Preprocessor is not able to detect inconsistencies of *config options* during parsing because it has no capabilities for interpreting the *Kconfig* model.

Undoubtedly, the variability described in the *Kconfig* files and in the source code are conceptually interconnected and have to be kept *consistent*. However, there is a gap between the various tools that are involved during the configuration and compilation phase of the Linux kernel. This gap has to be closed in order to detect existing bugs and avoid new ones when new features are added or existing features are refactored. This is especially important because changes (new additions or refactoring) at both sides (*Kconfig* model or code base) may potentially break consistency.

1.3 Our Contribution

In this paper we introduce a set of *conditions* that have to be asserted between code base and the *Kconfig* model in order to preserve consistency. We also confirm that this is a real problem by analyzing the current code base of the Linux kernel and show real bugs. We also expound that such bugs are introduced due to inadequate tool support. Moreover, we sketch the requirements of a tool to detect such inconsistencies, and, therefore, is able to close the gap between the variability at *model level* and *source code level*.

2. PROBLEM ANALYSIS

While analyzing the Linux family model we discovered that the implementation in form of C source code and the family model described in *Kconfig* show obvious inconsistencies. In summary, we identify two dimensions of consistency.

Referential consistency is categorized by:

1. every reference to a configuration item in the source code is referenced in the *Kconfig* variant model
2. every *Kconfig* option from the variant model is referenced from the implementation in the source code

Furthermore, *semantic consistency* can be accounted by:

3. every single configuration item in the source code is selectable in the *Kconfig* variant model
4. the *Kconfig* variant model covers all `#if` branches for all conditional blocks that use more than one configuration item

Fragments in the source code or in *Kconfig* that violate one or more of these consistency conditions result in features that can never be enabled or disabled. We therefore call features, which are implemented by fragments that are either *always* dead or alive *zombie features*.

In this paper, we focus on variation points that are implemented by *configuration-controlled* conditional compilation. Configuration controlled means that only conditional blocks that are affected by the configuration selection are considered. In Linux that means preprocessor macros beginning with the string `CONFIG_`. In theory, these macros should only be set in generated, configuration-derived files. In practice, this rule is not strictly enforced, so that estimations need to be made carefully.

As of version 2.6.23, the Linux source tree contains a script `scripts/checkkconfigsymbols.sh` that is intended to test the source code against the *Kconfig* model with respect to *referential integrity* regarding `CONFIG_xxx` symbols. However, this script is obviously not employed by the Linux maintainers, as a simple test run reports over **760** unresolvable references in kernel version 2.6.30. We modified the script to avoid false positives by only considering unresolvable references that appear in preprocessor directives (i.e., only in lines that start with `#if` or `#elif`). Even though this is most probably too strict and we thereby miss some legitimate problems, the number of unresolvable references only went down to **360**. Further inspection of the Linux code base shows that **206** macros, which start with the `CONFIG_` prefix, are being defined or undefined with the preprocessor. From these macros, only **39** are defined within *Kconfig*, this means that we estimate at least **321** real issues that need to be investigated!

This number is still calculated conservatively. A first analysis of the findings reveal several obvious bugs, such as the misspelling of CONFIG_CPUMASKS_OFFSTACK in the file include/-linux/irq.h (most probably CONFIG_CPUMASK_OFFSTACK was meant), or the item CONFIG_CPU_HOTPLUG in the file kernel/-smp.c, which should probably read CONFIG_HOTPLUG_CPU.[1]

Even though these number of inconsistencies already sounds pretty alarmingly, they most probably cover only the tip of the iceberg. Besides additional inconsistencies with respect to *referential integrity* that arise from *Kconfig* items that are not present in the source code, we can also expect inconsistencies between the encoding of feature dependencies and feature interactions in *Kconfig* and in the source code.

3. SOLUTION OUTLINE

It is obvious that the simple approach taken by the scripts/-checkkconfigsymbols.sh does only cover checking for *referential consistency* from the implementation to the variant model. However, we require detecting violations of all four conditions (i.e. both *referential* and *semantic consistency*), with satisfying accuracy.

Conditional compilation is implemented by preprocessor directives that order conditional blocks in a defined order, and is straightforward to analyze. However quantifying *configuration-controlled* variability is more challenging. Consider the following source code excerpt taken from the file include/linux/init.h:

```
#ifndef _LINUX_INIT_H
#define _LINUX_INIT_H
[...]
#if defined(MODULE) || defined(CONFIG_HOTPLUG)
#define __devexit_p(x) x
#else
#define __devexit_p(x) NULL
#endif

[...]
#endif /* _LINUX_INIT_H */
```

This source code snippet shows a typical C header. The very first preprocessor statement in this file is a technique known as "#include-guard", so that multiple inclusions of this file do not cause the actual contents to be evaluated multiple times by the compiler. In any case, these kinds of blocks are clearly not *configuration controlled* and therefore, must not be counted. Next, this header defines a qualifier __devexit_p(x) whose exact definition depends on both a configuration dependent variable CONFIG_HOTPLUG as well as another macro named MODULE. This macro can be totally unrelated to the *Kconfig selection*. Therefore, our tool does consider this conditional block as such, but for analyzing the variability of the compilation unit, only the macro CONFIG_HOTPLUG is considered.

[1]We have reported these inconsistencies as potential bugs to the Linux community and are awaiting a confirmation.

However, not all conditional blocks contribute to *configuration-controlled* variability. Consider the following code snipped taken from the file kernel/printk.c:

```
static int __init console_setup(char *str)
{
[...]
#ifdef __sparc__
        if (!strcmp(str, "ttya"))
                strcpy(buf, "ttyS0");
        if (!strcmp(str, "ttyb"))
                strcpy(buf, "ttyS1");
#endif
[...]
}
```

The purpose of this conditional block is portability of the file to the Sparc architecture. While this is an important concern, it is not handled by the means of *Kconfig* and therefore cannot be controlled by the user.

The preprocessor is used in this source file to include the previously shown header textually before passing the composed text to the compiler. In order to calculate the variability of the expanded compilation unit, the #include directive needs to be expanded. This allows us to consider both cases: The variability of the source file – the developer's view – and the variability of the compilation unit – the view of the compiler.

Interestingly, in Linux the configuration selection is not referenced explicitly in any source file. Instead, the configuration is implicitly present with a *forced*-#include technique as implemented by the -include compiler command-line option of GCC. This technique is used in all compilation units used during the compilation phase of the Linux kernel. Tools for evaluating the Linux source code therefore have to adopt this technique as well.

In this example we identified two #ifdef-blocks that are configuration dependent. The #ifdef-statement in the main source file does reference an identifier with the substring CONFIG_. However, it does not follow the convention that configuration items in Linux must **begin** with that prefix. Moreover the #if-statement in the header also contains an #else-block, which we count as an extra block.

A tool that reliably detects these inconsistencies requires a *global view* on all variation points in both the source code (the declarations of *conditional blocks*) and the *Kconfig* family model. As a first step, our framework therefore builds an *Implementation Variability Database* by scanning the source code. After scanning the complete kernel tree, the database contains all configuration-dependent conditional-blocks from the *implementation* of Linux. This is depicted in the lower part of Figure 1.

This database is essentially a fact database. In order to query such a database efficiently, first-order logic provides an appropriate language to create queries that allow drawing further conclusions. We therefore use the crocopat tool for relational programming [3] for this task. crocopat uses the *rigi standard format* from the rigi [20] reverse-engineering suite as input format. With crocopat, we can trivially do prolog-like

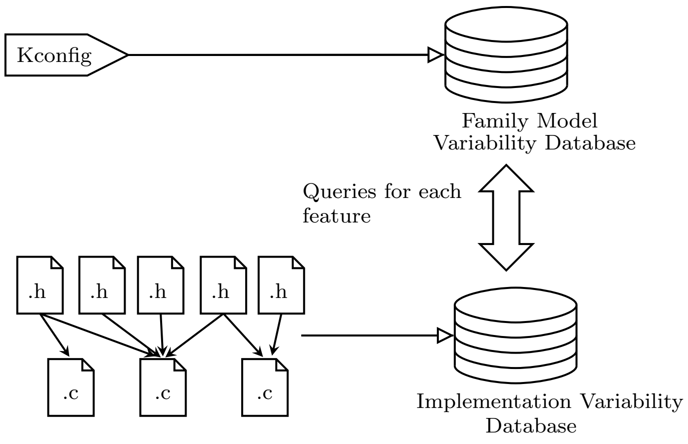
Kconfig
Family Model Variability Database
Queries for each feature
.h
.h
.h
.h
.h
.c
.c
.c
Implementation Variability Database

Figure 1: Our tool approach

queries on the *Implementation Variability Database* of any sort.

From the *Kconfig* files we extract the dependencies and constraints into a second database, the *Family Model Variability Database*. For this we need to parse the existing *Kconfig* files in Linux and analyze the dependencies and constraints. In order to be able to do queries and selections, we use the *rigi standard format* again so that `crocopat` can be reused. This is sketched in the upper part of figure 1.

In order to build the *Implementation Variability Database* we will calculate the *variability* of individual expanded compilation units. In this context *variability* means all possible different token streams (the input for syntactic parsing) that can be generated by the C Preprocessor (`CPP`) when preprocessing a compilation unit. When the expression of a `CPP` directive evaluates to true, the corresponding code will be read and inserted into the resulting token stream. If the expression evaluates to false, this code block is skipped and not included in the token stream. It means that in theory, a file with n different conditional directives (`#if`, `#elif`, etc.) might consist of 2^n different combinations.

Fortunately, many of these configurations cannot be composed in practice. For example, the conditional blocks defined by an `#if` directive and its corresponding `#else` will never be enabled at the same time, because of the exclusive semantic of `#if-#else` blocks. In order to calculate the exact variability of each compilation unit, our tool generates a formula like $f\colon (x_1, \dots, x_n) \to \{0, 1\}$ where $x_1, x_2, \dots, x_n$ are the *Kconfig* symbols used in `CPP` statements. This formula is basically the conjunction of the condition of each conditional block. These conditions are in fact the conjunction of the expression where the *Kconfig* symbols are used (e.g. `CONFIG_SMT && CONFIG_X86`) with the structural constraints like nested blocks (implications of the form ($child \Rightarrow parent$)). Another type of constraint is imposed by the semantics of `#if-#elif-#else` block groups, where for each block $(B_1, B_2, \dots, B_n)$ of such group a dependency of the form $B_1 \Rightarrow \neg(B_2 \vee B_3 \vee \dots \vee B_n)$ is required.

After constructing such a boolean function, it can be translated into a *binary decision diagram* (BDD) so that further analysis, like the calculation of the truth table, can be performed very efficiently. The lines in the truth table of such a boolean function that evaluate to true provide the set of valid configurations of a compilation unit. We can then use this set of valid configurations and crosscheck with the *Kconfig* dependencies in order to discover combinations of features that are allowed in the code base but not in the variant model, the so called, *zombie* features.

Having both databases available will allow us to check *referential integrity* of configuration items. In order to check *semantic integrity*, we transform the feature dependencies of the *Family Model Variability Databases* into BDDs that can be queried for satisfiability very efficiently. This way, checking satisfiability for each single configuration option is just a query against the BDD.

Checking semantic integrity with `#ifdef`-blocks that depend on more than one *configuration option* is more challenging. In a first step, we use the *Implementation Variability Database* to identify all conditional blocks with more than one configuration item and obtain the exact expression used in the conditional block. The challenge here is that conditional blocks may be influenced by configuration items both *explicitly* and *implicitly*. With *explicit influence* we mean that the *Kconfig symbol* appears literally in the expression of the preprocessor directive. *Implicit influence* happens either by nested `#ifdef` directives or when the `#define` preprocessor directive is used in a conditional block to define another configuration item. The framework must consider all configuration items (both implicit and explicit) for each conditional block and calculate on this basis the conditional-compilation *path-coverage*. Blocks that cannot be reached in any configuration are then in violation with the *semantic consistency* condition.

4. DISCUSSION

In the previous section, we have outlined our proposed tool that will be part of a greater framework to assist managing and verifying the variability in the Linux kernel. We hope that our tool will be adopted by the various Linux communities involved with variability management and issues that arise from it.

Especially targeted for driver developers, our framework would be able to show what configuration derived variability is actually used by a device driver. This would highlight the variability points introduced for example by driver developers, but also indicate interaction with other features that might have not been considered (yet) during the implementation of the driver. Depending on these additional variation points, this can indicate that additional test cases need to be considered.

Similarly, subsystem maintainers could use our tool during reviews and integrations. While inspecting the Linux Guidelines for patch submission and review[2] it turns out that **9** out of **24** points deal with *Kconfig* related issues. These issues are very hard to test and review; our framework can assist

[2]as found in the file `Documentation/SubmittingPatches`

here with visualizing and verifying the additional variation points.

While we are convinced that our framework will be useful for kernel maintainers, we need to consider if our approach scales with the amount of variability in Linux. With Linux, we are facing a variant model of about **8000** features. It is well known that the size of BDDs is very sensitive to the number and order of its variables, which may lead to insufficient memory problems. However, our preliminary results clearly show that the variability is not uniformly distributed across the Linux source code, but *variability hot spots* can be identified easily. Therefore, we will work on a per compilation-unit basis in order to keep the BDDs reasonably sized.

A first analysis with a self written tool based on sparse [10], a framework for static analysis written for the Linux kernel, shows that less than **10%** of all files of the Linux kernel use more then **2** different *Kconfig* symbols. When considering expanded compilation units, we see that more than **85%** of all compilation units have at most **350** different symbols in them. It is clear that we must also consider *Kconfig symbols* that are not only explicitly named in a compilation unit but come into effect indirectly. This can happen for example when a compilation unit *overrides* a configuration item with the `#define` preprocessor statement. Moreover, we need to compute a *global variable order* so that partial computation results can be reused. According to Mendonca et. al. [14], the largest feature models that can be handled today have about 2000 features and 400 extra constraints. Still, we expect that most compilation units in Linux will not exceed these limits, if any.

Is this a language problem?

Could the problem have been avoided in the first place? In many cases the usage of the C-Preprocessor is held responsible for maintenance problems in large software projects [19]. Would it be feasible to avoid the preprocessor in Linux? For Linux, the answer is *no*. Operating systems need modularization that is finer grained than provided by the plain C language. This fact was already known during the design and implementation of the C programming language [9], so the C Preprocessor became part of every implementation of C. With the conditional compilation feature, fragments of a program can be modularized at a *sub-statement* level. However, we identify two main issues with the approach taken by the preprocessor: a) conditional blocks cannot take any context into account, and b) the blocks are declared anonymously and cannot be referenced from anywhere.

The first point is necessary for any form of generic implementation. While conditional blocks cannot handle this directly, the common workaround for this limitation is to declare a preprocessor macro that takes parameters and declare multiple, alternate implementations of the macro. Later, in the C implementation, the macro is called like a function – however parameters are bound *by name* to the macro code. This workaround has limitations: First, the macro needs to be declared in a single line, although line continuations with the backslash (\) character are allowed. Second, most implementations do not allow common debugging facilities like setting a breakpoint or stepping through the macro code. Third and most importantly, no type checking is done at all during macro expansion. The lack of type support prevents the C compiler from printing helpful diagnostic messages in many cases of problems and thus leads to code that is hard to maintain.

Modularisation of program fragments

So in the end, the decision to use the C Preprocessor for modularizing these fragments is based on technical circumstances. While we envision compositional language approaches for their technical handling like feature oriented programming (FOP) [1], aspect oriented programming (AOP) [4] or comparable languages, we should also consider the motivation for introducing these parts in form of function fragments in the first place. Linux is a very implementation driven project for mainly two reasons. As indicated before, operating systems are inherently implementation driven. Moreover, Linux is a high traffic free-software project with a very active development community. For these reasons, many optional features of various kinds have been proposed and integrated into the Linux code base.

However, there are many cases where two or more optional features behave differently when they are selected at the same time, compared to the case that only a single one is selected. This problem has already been discussed by Batory et. al as the *optional feature problem* [8, 11]. The proposed solution is to modularize the features as *derivatives*, so that the variant management system can select these *derivative modules* according to the needs of the implementation in a given configuration.

In Linux these *derivatives* are not modularized at all, but scattered across the Linux source base using the C Preprocessor. Because nothing tracks the consistency of these modules to the family models, *unused* derivatives become easily orphaned but still end up in the resulting product (the bootable Linux kernel image). This can result in modules that can be identified by preprocessor statements but can never be enabled or disabled. For this reason, we call these *undead* modules *zombies*.

5. RELATED WORK

Lotufo [12] analyzes the complexity of maintaining the Kconfig files. An investigation of 29 stable versions of the Linux kernel configuration options is presented. He concludes that the complexity of the code for the configuration options increases consistently, as well as the complexity of the resulting model. Interestingly, as we point out in this work, he also suggests that reasoning capabilities should be added to *Kconfig*.

Post and Sinz [16] present a technique called lifting that converts all variants of a SPL into a meta program in order to facilitate the application of verification techniques like static analysis, or model checking. They evaluate their approach by applying it into to the `Kconfig` files of the Linux kernel: The `Kconfig` files were converted into C-code for analysis with a source code checker, which reveals two new bugs attached to uncommon configuration. This fact also supports the idea that the Linux kernel should introduce *reasoning* capabilities for its variant model.

Kästner et. al. [7] present an approach to check the syntactic correctness of all variants of a software product line. They present the tool CIDE which is able to analyze CPP-based code among other languages. The concept of finding bugs introduced by the use of CPP directives is similar to our work, however, while CIDE focuses on syntactic errors, our approach finds inconsistencies between the source code and the variant model directly.

6. CONCLUSIONS

The mapping between the implementation of the variability points in the source code and the family model is incomplete and inaccurate. Our investigation of the Linux kernel shows that the mapping between the implementation of the variability points in the source code and the family model shows obvious inconsistencies. Our first probably inaccurate, but conservative checks indicate over 300 real bugs that arise from conditional compilation blocks which use configuration options that are never defined in the *Kconfig* variant model.

We believe that this number is only the tip of the iceberg and expect that additional bugs from defined *Kconfig* items that are never used in the source code remain undetected. Besides these violations of *referential integrity*, we also believe that more inconsistencies can be detected by checking for *semantic integrity*, that is if the condition of a conditional block can be satisfied in the *Kconfig* variant model for all branches that is defined by the condition in the `#if` or `#ifdef` preprocessor statement.

It is clear that the impressive amount of variability cannot be checked by kernel developers manually in the source code. We therefore propose a tool that accompanies *Kconfig*, but represents the counterpart for kernel developers. This tool will help kernel developers to check all consistency conditions reliably. According to our estimations we are confident that our approach is feasible for the impressive amount of variability points in the Linux code base.

7. REFERENCES

[1] D. Batory. Feature-oriented programming and the AHEAD tool suite. In *26th (ICSE '04)*, pages 702–703, 2004.

[2] D. Beuche, A. Guerrouat, H. Papajewski, W. Schröder-Preikschat, O. Spinczyk, and U. Spinczyk. The PURE family of object-oriented operating systems for deeply embedded systems. In *2nd (ISORC '99)*, pages 45–53, St Malo, France, May 1999.

[3] D. Beyer. Relational programming with crocopat, 2006.

[4] K. Czarnecki and U. W. Eisenecker. *Generative Programming. Methods, Tools and Applications.* May 2000.

[5] E. W. Dijkstra. The structure of the THE-multiprogramming system. *Commun. ACM*, 11(5):341–346, May 1968.

[6] A. N. Habermann, L. Flon, and L. W. Cooprider. Modularization and hierarchy in a family of operating systems. *Commun. ACM*, 19(5):266–272, 1976.

[7] C. Kästner, S. Apel, S. Trujillo, M. Kuhlemann, and D. Batory. Guaranteeing syntactic correctness for all product line variants: A language-independent approach. In *Proceedings of the 47th International Conference Objects, Models, Components, Patterns (TOOLS EUROPE)*, volume 33 of *Lecture Notes in Business Information Processing*, pages 175–194. Springer Berlin Heidelberg, June 2009.

[8] C. Kästner, S. Apel, S. S. ur Rahman, M. Rosenmüller, D. Batory, , and G. Saake. On the impact of the optional feature problem: Analysis and case studies. 2009.

[9] B. W. Kernighan and D. M. Ritchie. *The C Programming Language.* 1978.

[10] Linus Torvalds. Sparse - a semantic parser for C. `http://www.kernel.org/pub/software/devel/sparse/`, 2003.

[11] J. Liu, D. Batory, and C. Lengauer. Feature oriented refactoring of legacy applications. In *10th (SPLC '06)*, pages 112–121, 2006.

[12] R. Lotufo. On the complexity of maintaining the linux kernel configuration. Technical report, 2009.

[13] A. Massa. *Embedded Software Development with eCos.* New Riders, 2002.

[14] M. Mendonça, A. Wasowski, K. Czarnecki, and D. D. Cowan. Efficient compilation techniques for large scale feature models. In *GPCE*, pages 13–22, 2008.

[15] D. L. Parnas. On the design and development of program families. *IEEE Trans. Softw. Eng.*, SE-2(1):1–9, Mar. 1976.

[16] H. Post and C. Sinz. Configuration lifting: Verification meets software configuration. In *ASE*, pages 347–350, 2008.

[17] J. Sincero, H. Schirmeier, W. Schröder-Preikschat, and O. Spinczyk. Is the linux kernel a software product line? In F. van der Linden and B. Lundell, editors, *International Workshop on Open Source Software and Product Lines (SPLC-OSSPL 2007)*, Kyoto, Japan, 2007.

[18] J. Sincero and W. Schröder-Preikschat. The linux kernel configurator as a feature modeling tool. In S. Thiel and K. Pohl, editors, *SPLC (2)*, pages 257–260. Lero Int. Science Centre, University of Limerick, Ireland, 2008.

[19] H. Spencer and G. Collyer. #ifdef considered harmful, or portability experience with C News. In *1992*, June 1992.

[20] M.-A. D. Storey, K. Wong, P. Fong, D. Hooper, K. Hopkins, and H. A. Müller. On designing an experiment to evaluate a reverse engineering tool. In *Proceedings of the 3rd Working Conference on Reverse Engineering, (WCRE'96), Monterey, California, USA, November 8-10, 1996*, pages 31–, November 1996.