Configuration Coverage in the Analysis of Large-Scale System Software*

Reinhard Tartler Daniel Lohmann
Christian Dietrich Christoph Egger Julio Sincero
{tartler,lohmann,qy03fugy,siccegge,sincero}@cs.fau.de
Friedrich-Alexander University Erlangen-Nuremberg, Germany

ABSTRACT

System software, especially operating systems, tends to be highly configurable. Like every complex piece of software, a considerable amount of bugs in the implementation has to be expected. In order to improve the general code quality, tools for static analysis provide means to check for source code defects without having to run actual test cases on real hardware. Still, for proper type checking a specific configuration is required so that all header include paths are available and all types are properly resolved.

In order to find as many bugs as possible, usually a "full configuration" is used for the check. However, mainly because of alternative blocks in form of `#else`-blocks, a single configuration is insufficient to achieve full coverage. In this paper, we present a metric for configuration coverage (CC) and explain the challenges for (properly) calculating it. Furthermore, we present an efficient approach for determining a sufficiently small set of configurations that achieve (nearly) full coverage and evaluate it on a recent Linux kernel version.

1. INTRODUCTION

Much system software employs compile time configuration as a simple and widely used technique for configuration management, which allows tailoring with respect to a broad range of supported hardware architectures and application domains. A prominent example is the Linux kernel, which provides more than 11,000 configurable features.

Technically, static configurability is mostly implemented by means of the C Preprocessor (`CPP`) [9], despite all the disadvantages ("`#ifdef` hell") this approach is known for [5, 8]. In the case of Linux (2.6.35), the result are more than 84,000 `#ifdef`-blocks spread over 28,000+ source code artefacts. These numbers are subject to constant growth; they have practically doubled over the last five years [11].

From the maintenance point of view, compile-time configurability imposes big challenges with respect to configuration coverage (CC). With this term we denote the coverage of software quality measures (such as unit tests or the use of static bug-finding tools) with respect to the *configuration-conditional* parts of the source code.

1.1 Problem Statement

One often hears statements like "we have tested this with Linux, it works." (Yes, but which configuration?) "Our tool has found 47 bugs in Linux!" (That is impressive, but how much of Linux did you actually analyze?). Even though static configurability by conditional compilation is omnipresent in system software, the community seems to be somewhat agnostic of the CC issue:

1. There is a plethora of literature and tool support (such as GCOV) to implement, for instance, unit tests that reach a certain coverage criteria (such as *decision coverage*: every edge in the control flow graph is taken at least once). However, the available methods and tools generally understand coverage from the *compiler's* perspective – they (implicitly) assume that preprocessing (thus, static configuration) has already happened. The problem of CC is completely ignored.

2. Many great papers (e.g., [2, 4, 10]) have been published about applying static bug-finding approaches to Linux and other pieces of system software. In all cases the authors could find (and eventually fix) a significant number of bugs. It is, however, remarkable, that the issue of CC is not mentioned at all in these papers; in most cases the authors do not even *state* the configuration(s?) they have analyzed.

(1.) has practical implications on the every-day work of software developers: A Linux developer, for instance, who has modified a bunch of files for some maintaining task would probably want to make sure that every edited line of code does actually compile and has been tested before submitting a patch. However, deducing a set of configurations that covers all relevant compilation-conditional parts of the source is a sometimes difficult, but always tedious and error-prone task. Our initial findings show that some files (such as `kernel/sched_fair.c`) require seven different configurations to cover all conditional parts – and this is probably just a lower bound, as we do not yet include header files in our analysis.

With respect to (2.): Private communication with some of the authors revealed that with Linux the common approach is to use a standard configuration (x86 with either `defconfig` or `allyesconfig`). This is perfectly acceptable – their goal was to find bugs and they have found bugs [1]. However,

*This work was supported by the German Research Council (DFG) under grant no SCHR 603/7-1.

how many additional bugs could possibly be found with full coverage? Given that `allyesconfig` is the de-facto standard for testing things with Linux: What is its actual coverage?

1.2 About this Paper

We think that configurability as a system property is a significant (and widely underestimated) concern for the complete software development cycle. In the VAMOS[1] project we aim to mitigate this situation by developing new methods and tools to deal with configurability-induced complexity. In a recent paper [11] we have shown that by better tool support, many configurability-related source-code defects (such as dead `#ifdef`-code) and bugs can be found upfront. Our findings led to fixes for 20 new bugs and the removal of 5,000+ superfluous lines of `#ifdef`-code in Linux 2.6.36. The work presented in this paper continues this line of research: We describe an approach to measure the effective CC of Linux configurations. We have extended our UNDERTAKER tool to automatically generate Linux configurations for a given source file in order to achieve full CC. In our preliminary results we thereby achieve a CC of 94 percent, whereas `allyesconfig` reaches a CC of only 78 percent.

2. HOW TO MEASURE CONFIGURATION COVERAGE IN LINUX

Technically, the `CPP`-statements of a C program describe a meta-program that is executed by the C Preprocessor before the actual compilation by the C compiler takes place. In this meta-program, the `CPP` expressions (such as `#ifdef` – `#else` – `#endif`) correspond to the conditions in the edges of a loop-free[2] control flow graph (CFG); the thereby controlled fragments of C-code (i. e., the bodies of `#ifdef`-blocks) are the statement nodes (Figure 2). On this CFG, established metrics, such as statement coverage or path coverage, can be applied. In this paper, we go for *statement coverage* and define configuration coverage (CC) as the fraction of *selected* configuration-conditional blocks divided by the number of *available* configuration-conditional blocks. (In the longer term we aim for *path coverage*; this is more thoroughly discussed in Section 5.)

Not every `CPP` identifier that is used in some `#ifdef`-expression does actually describe a configuration-conditional block. For instance, the identifiers used for `#include` guards do not contribute to configurability. In order to get a sound analysis of the configuration-controlled `CPP`-based variability in Linux, we restrict the analysis to those flags that can be controlled by KCONFIG (the configuration front-end of Linux). Fortunately, this is relatively easy, as it is a convention in Linux to prefix all configuration-related `CPP` identifiers with the string `CONFIG_`. We therefore limit our analysis to blocks with `CPP` expressions that reference at least one `CONFIG_xxx` identifier. We furthermore restrict our initial analysis to one architecture, namely `arch-x86`. Figure 1 shows the progression of configuration-conditional blocks over the last 6 years of Linux development for this architecture.

Linux ships with a number of predefined default configurations in the source tree. As pointed out in Section 1.1, we assume that tools for static analysis have generally be applied on one of these reference configurations of Linux

[1]VAriability Management in Operating Systems

[2]Leaving aside “insane” `CPP` meta-programming techniques based on recursive `#include`

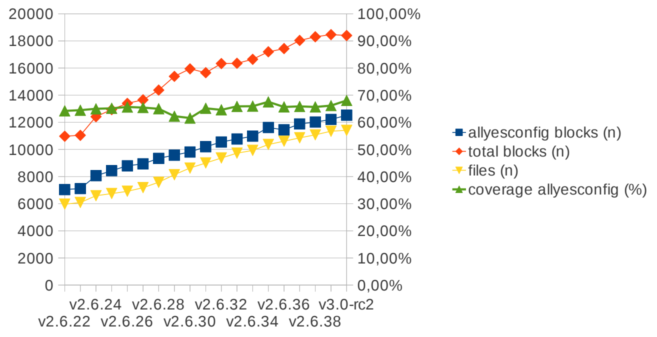


Figure 1: Progression of user controlled variability in Linux (arch-x86) in the last six years

(`allyesconfig` in the best case), ignoring any user-induced variability.

Figure 1 presents the CC of `allyesconfig` over the last six years of Linux kernel development. Because of alternative features (which manifest as `#else-blocks` in the code) generally there is no configuration that selects all configuration-conditional blocks. However, depending on the kernel version, the CC of `allyesconfig` has always been in the range of 60 to 70 percent: This is not too bad – with more and more features being added to Linux, we had expected this number to decrease over time. We also had expected it to be somewhat lower in general. Still, when assuming that most Linux kernels are tested with a configuration based on `allyesconfig`, code that is not covered by this configuration is generally less tested and therefore is more likely to contain bugs.

The numbers presented so far, however, do not represent the full picture in two respects:

Firstly, as we have shown in previous work [11], not every seemingly configuration-conditional block is actually configuration-conditional. The Linux source code contains many (thousands!) *dead blocks* – conditional blocks that are never selected, as their presence condition contains a contradiction – either within the source code itself (such as an `#ifndef CONFIG_FOO` block nested into an `#ifdef CONFIG_FOO` block) or in conjunction with the constraints defined by the KCONFIG-models. These “technically unreachable” blocks should not be considered with respect to CC. Luckily, our UNDERTAKER tool [11] can be employed to detect global and architecture-specific dead blocks. When applying our tools to Linux 2.6.35, we observe that the actual CC of `allyesconfig` increases to nearly 78 percent.

Secondly, we do not yet analyze the variability brought in by the header files. The effects of this limitation are difficult to judge (we are working on this), however, it might be significant: There is a coding guide line that variability should be implemented as far as possible in the headers. Instead of polluting the implementation with repetitions of the same `#ifdef`-block again and again, developers are encouraged to put this code into a conditionally defined macro. If this approach is used for the implementation of many alternative features the actual CC of `allyesconfig` will be lower.

In any case, however, higher or even full CC is desirable. In the following section we discuss how to efficiently create a set of configurations in order to achieve full CC.

```
#ifdef CONFIG_DISCONTIGMEM                    Block 1
static inline int pfn_to_nid(unsigned long
    pfn)
{
#ifdef CONFIG_NUMA                            Block 2
   return((int) physnode_map[(pfn) /
       PAGES_PER_ELEMENT]);
#else                                         Block 3
   return 0;
#endif
}
#endif                                        Block 1
```

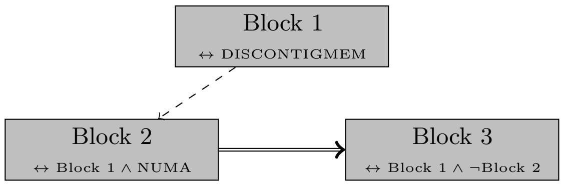


Figure 2: The internal structure of the conditional blocks are translated in a tree-like graph. The dashed edges represent nesting while double edges represent alternatives. Each node contains the presence condition $\mathcal{PC}$ of the block it represents.

3. APPROACH

We represent the structure of CPP blocks in a tree-like graph with two types of edges: nesting of blocks and #if-#elif cascades. Figure 2 illustrates this representation.

The goal of the approach is to find a set of configurations that maximizes the CC with respect to statement coverage. Since a thorough code test requires all blocks to be selected at least once, ideally, a minimal set of configurations is used in order to reduce testing effort. Note that the minimal set is not well-defined; there can be more than one minimal set of configurations for a given source file.

The minimal amount of configurations that when accumulated select all blocks could be calculated by solving the graph coloring problem on the "conflict graph": All nodes represent a conditional block. When two blocks cannot be selected at the same time (e .g., because of the CPP structure or constraints from the KCONFIG model), an edge between the blocks is added that represents the conflict. The minimal number of configurations is the number of colors required so that that two adjacent nodes have different colors. This is NP hard and, hence, unfeasible to be applied in a general-purpose development tool that scales to the size of Linux. Therefore we seek for heuristics to get a suboptimal, but yet sufficiently useful solution for real-world use-cases.

The structure of CPP blocks and the identifiers used in their expressions are translated into a propositional formula such that each CPP identifier and each conditional block is represented as a propositional variable. For each conditional block, we call the condition under which the block is selected the *presence condition* of the block. The presence condition is influenced by two factors: First, the structure and semantics of the CPP language impose constraints on the presence condition of each block. Second, extra constraints apply that come from the constraints as expressed by the KCONFIG models. In order to incorporate these extra constraints, we reuse the model extractors from [11]. In the following algorithms, the presence condition of a given block b is determined by the function $\mathcal{PC}(b)$. The function Φ_{file} is the formula that conjugates the *presence conditions* of all conditional blocks given as input. Conjugated with a block b, this formula determines a valid configuration of blocks and CONFIG_ identifiers that select the block b.

On this basis, we have developed two algorithms that calculate a set of configurations that (accumulated) selects all blocks:

The first algorithm is depicted in Figure 3. The general idea is to iterate over all blocks (Line 5) and use a SAT solver to generate a configuration that selects the current block (Line 7). As the most expensive operation is the number of SAT queries, covered blocks in already found configurations are skipped (Line 6). The set B collects the already covered blocks (Line 11). The resulting set R contains the found configurations. This algorithm therefore requires n SAT calls in the worst case.

As a further optimization, the SAT solver is tweaked to try to enable as many propositional variables as possible while looking for satisfying assignments for the formula, which increases the amount of selected blocks per configuration.

On the basis of this simple algorithm, Figure 4 shows an improved version of the approach that more aggressively minimizes the number of configurations and hence, improves the quality of the results. Here, the inner loop (Line 7) collects as many blocks as possible to the working set WS so that there is a configuration that covers all blocks in the working set. Blocks that obviously conflict with some other block of the working set, such as #else blocks of an already selected #if block, are skipped in Line 9. Line 10 verifies that there is configuration, such that all blocks of the current working set and the current block are selected. Otherwise the block is skipped. For the found working set of "compatible" blocks, a configuration is calculated similarly to the simple variant (Line 19f) and added to the resulting set R (Line 21). This results in n^2 SAT calls for the worst case.

```
1:  function NAIVECOVERAGE(IN)                 ▷ List of blocks
2:      $R \leftarrow$ empty set               ▷ Found Configurations
3:      $B \leftarrow$ empty set               ▷ Selected Blocks
4:      $\Phi_{\text{file}} \leftarrow \bigwedge_{IN}^{b} \mathcal{PC}(b)$   ▷ all presence conditions in file
5:      for all $b$ in IN do
6:          continue if $b \in B$              ▷ already processed
7:          r $\leftarrow$ sat($\Phi_{\text{file}} \wedge b$)
8:          if r.failed() then                 ▷ dead block
9:              $B \leftarrow B \cup \{b\}$    ▷ mark as processed
10:         else
11:             $B \leftarrow B \cup$ r.selected_blocks()
12:             $R \leftarrow R \cup \{$r.configuration()$\}$
13:         end if
14:     end for
15:     return $R$
16: end function
```

Figure 3: Naïve Variant

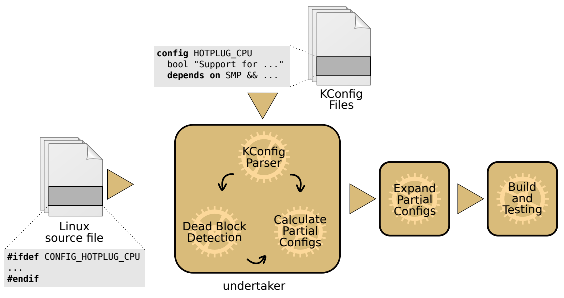
4. RESULTS

We have implemented both algorithms in the UNDERTAKER toolchain[3] as an additional command-line option. An enhanced version of the model extractor from [11] extracts the constraints from KCONFIG. As SAT solver, the PICOSAT

[3]http://vamos.informatik.uni-erlangen.de

```
function GreedyCoverage(IN)                              ▷ List of blocks
    $R \leftarrow$ empty set                                  ▷ Found Configurations
    $B \leftarrow$ empty set                                  ▷ Selected Blocks
    $\Phi_{\text{file}} \leftarrow \bigwedge_{IN}^{b} \mathcal{PC}(b)$     ▷ all presence conditions in file
    while IN.size() != B.size() do
        $WS \leftarrow$ empty set                             ▷ reset working set of blocks
        for all $b$ in IN do
            continue if $b \in B$                             ▷ already processed
            continue if $b$ conflicts $WS$
            r $\leftarrow$ sat($b \wedge WS \wedge \Phi_{\text{file}}$)
            if r.failed() then
                if $WS$.empty() then                          ▷ dead block
                    $B \leftarrow B \cup \{b\}$                     ▷ mark as processed
                end if
            else
                $WS \leftarrow WS \cup \{b\}$                     ▷ Add to working set
            end if
        end for
        r $\leftarrow$ sat($WS \wedge \Phi_{\text{file}}$)
        $B \leftarrow B \cup$ r.selected_blocks()
        $R \leftarrow R \cup \{$r.configuration()$\}$
    end while
    return $R$
end function
```

Figure 4: Greedy Variant



```
config HOTPLUG_CPU
  bool "Support for ..."
  depends on SMP && ...
```

```
#ifdef CONFIG_HOTPLUG_CPU
...
#endif
```

Figure 5: Principle of Operation

package[4] is used. Given a single source file as input, UNDERTAKER calculates (partial) configurations that maximize the configuration coverage for this specific file. This allows programmers participating in large projects, such as the Linux kernel, to focus on the specific parts of the kernel that they currently work on.

We run a preliminary prototype of our tool chain to analyze CC of the Linux v2.6.35 kernel. The whole experiment was done under consideration of the constraints of the x86 architecture. Table 1 summarizes the results. Out of 10,365 `.c` files, we identify 3,163 files that contain configuration-conditional blocks. On these files, our tool calculates in total 4,505 (naïve version) and 4,435 (greedy version) configurations. Generating the configurations took about 5 minutes minutes for the naïve version and about 15 minutes for the greedy version of the algorithm on a standard quadcore workstation (Intel Core 2, 2.83 Ghz). However, all times where measured in single-thread operation.

The generated configurations do not include selections for items without constraints on the current file. For full configurations, the resulting partial configurations need to be supplemented with the remaining items. The selection of the remaining items is irrelevant. For this expansion process, we reuse the KCONFIG infrastructure. In practice, this process is cumbersome as KCONFIG in some cases silently overwrites user choices instead of signaling incompatible item selections. Due to technicalities of our approximations of the KCONFIG model [13], our preliminary implementation fails to process some files ($< 7\%$). The problematic files tend to contain complicated `CPP` expressions on which the `allyesconfig` coverage resulted in a coverage of only 50 percent. In comparison, the CC of `allyesconfig` is 70 percent and uses 3163 compiler invokations. With our approach, the resulting partial configurations increase the CC to 84 percent and require 4530 compiler invocations. This means with investing 30 percent more static analysis runs the CC improves by 14 percent.

[4]`http://fmv.jku.at/picosat/`

Analyzed files	10,365
Files with variability	3,163
Rate of files with variability	30.52%
Sum of all (partial) configurations (naïve)	4,505
Sum of all (partial) configurations (greedy)	4,435
Approach worked on #files	2,970
Approach failed on #files	193
Configuration expansion success rate	93.90%
Sum of *configuration controlled* conditional blocks	16,444
Sum of blocks selected by `allyesconfig`	11,511
Sum of all blocks selected by `undertaker-coverage`	13,844
Coverage `allyesconfig` (non-dead-corrected)	70.00%
Coverage `undertaker` (non-dead-corrected)	84.19%
Dead blocks	1,778
Selectable blocks (excluding dead blocks)	14,666
Selected by `allyesconfig`	11,511
`allyesconfig` coverage	78.49%
Covered by `undertaker` ("greedy" algorithm)	13,844
`undertaker` coverage	94.40%
`undertaker` coverage / `allyesconfig` coverage	1.20

Table 1: CC-Analysis Results with Linux 2.6.35, arch-x86.

These numbers, however, do not take dead blocks into account: As described in Section 2, a considerable amount of blocks cannot be selected by any configuration, which skews the comparison. Therefore, we used the UNDERTAKER tool to detect such *dead* blocks and subtract them from the total number of selectable blocks. While this increases the CC in both cases, we see that in this comparison, our approach achieves 1.2 times better coverage (78.49% to 94.40%).

5. DISCUSSION

Header-induced Variability.

Even though our current implementation reaches a significantly higher CC than `allyesconfig`, it fails to cover a remaining six percent of valid configuration-conditional blocks. One reason for this is that our current implementation does not take constraints into account that derive from `#define` and `#undef` statements in header files. In order to integrate these extra constraints, a sound implementation of a partial preprocessor such as [3] or [12] would be necessary. We are currently looking into integrating such techniques in our approach. First quick experiments with [3] indicate that with a partial expansion of header-induced macros, our approach achieves up to 99 percent coverage and is 2.5 times better than `allyesconfig`.

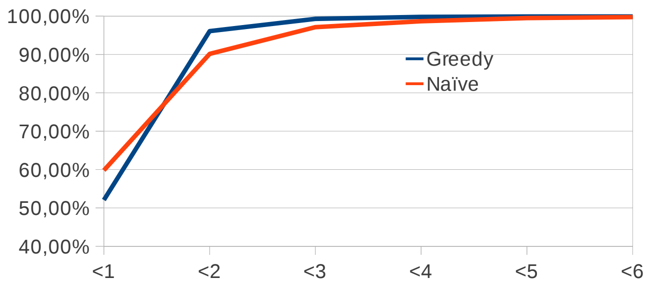


Figure 6: Relative amount of source files for which CC is reached with n configurations

Comparison of the Algorithms.

The histograms in Figure 6 show that for more than 90 percent of the files in Linux CC can be reached by at most two configurations. We can therefore conclude that the implemented heuristics yield reasonably small solutions. In direct comparison, the greedy algorithm produced about 1.5 percent fewer configurations than the naïve variant in total, but takes more than twice the run time, which is not worth the effort in the general case. However, whereas for the greedy variant, no file requires more than 7 configurations to reach CC, the naïve variant requires up to 16 configurations in a few cases. Hence, it would be worthwhile to always start with the naïve algorithm and fall back to the greedy algorithm only when the resulting number of configurations passes by a certain threshold (e. g. five).

Coverage Criteria and Related Work.

The approach presented in this paper generates a set of configuration that achieve *statement coverage*. While many syntactical and semantical problems can already be found by statement coverage, some issues might still be missed. For instance, consider two conditional blocks, one containing a function definition and the other a corresponding function call: The selection of the second block requires the presence of the first one. If the presence conditions of the two blocks do not imply this constraint, the configuration sets generated by the approach presented in this paper may or may not miss a configuration where this constraint is violated.

In order to cover such problems, stronger coverage conditions are necessary. In order to completely type-check all configurations, every possible path must be tested. However, full *path coverage*, would probably require too many configurations to be usable in practice. Since most type checking problems in `C` are some kind of "definition-use" violation, pairwise combinatorial testing, like presented in [6] for features of a software product line, might be a pragmatic alternative. This is a topic of further research.

Palix [7] tries to reproduce a ten year old analysis on Linux in order to investigate the evolutionary development of Linux across the last decade. As the old experiment misses to state the exact configuration that was used, the environment could only be approximated. Hereby, the paper indirectly discusses CC in the sense that the selected configuration can (and does) affect the results of static analysis tools considerably. We take this anecdote as call for further integration of configuration consistency checks into static analysis tools.

6. SUMMARY AND FUTURE WORK

We have introduced and discussed the problem of configuration coverage (CC) in preprocessor-configured system software. CC is defined as the fraction of conditional blocks that can be selected in a source file. An analysis on the Linux kernel shows that the generally applied `allyesconfig` configuration selects only 78 percent of all configuration-conditional `#ifdef` blocks.

As the remaining code is likely to be less tested, we propose an approach that efficiently calculates a set of configurations to reach (nearly) full coverage. In the future, these resulting configurations can be used to systematically apply static analysis tools on code that is impossible to analyze with a single configuration. We invite other researchers to suggest suitable tools to verify the hypothesis that such code tends to contain more bugs.

References

[1] Bessey, A., Block, K., Chelf, B., Chou, A., Fulton, B., Hallem, S., Henri-Gros, C., Kamsky, A., McPeak, S., and Engler, D. A few billion lines of code later: using static analysis to find bugs in the real world. *CACM 53* (February 2010), pp. 66–75.

[2] Engler, D., Chen, D. Y., Hallem, S., Chou, A., and Chelf, B. Bugs as deviant behavior: a general approach to inferring errors in systems code. In *18th ACM Symp. on OS Principles (SOSP '01)*, ACM, pp. 57–72.

[3] Kästner, C., Giarrusso, P. G., Rendel, T., Erdweg, S., Ostermann, K., and Berger, T. Variability-aware parsing in the presence of lexical macros and conditional compilation. In *26th ACM Conf. on OOP, Systems, Languages, and Applications (OOPSLA '11)*, ACM.

[4] Kremenek, T., Twohey, P., Back, G., Ng, A., and Engler, D. From uncertainty to belief: inferring the specification within. In *7th Symp. on OS Design and Implementation (OSDI '06)*, USENIX, pp. 161–176.

[5] Liebig, J., Apel, S., Lengauer, C., Kästner, C., and Schulze, M. An analysis of the variability in forty preprocessor-based software product lines. In *32nd Int. Conf. on Software Engineering (ICSE '10)*, ACM.

[6] Oster, S., Markert, F., and Ritter, P. Automated incremental pairwise testing of software product lines. In *14th Software Product Line Conf. (SPLC '10)* (2010), vol. 6287 of *LNCS*, Springer, pp. 196–210.

[7] Palix, N., Thomas, G., Saha, S., Calvès, C., Lawall, J. L., and Muller, G. Faults in Linux: Ten years later. In *16th Int. Conf. on Arch. Support for Programming Languages and Operating Systems (ASPLOS '11)* (2011), ACM, pp. 305–318.

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

[9] Spinellis, D. A tale of four kernels. In *30th Int. Conf. on Software Engineering (ICSE '08)*, W. Schäfer, M. B. Dwyer, and V. Gruhn, Eds., ACM, pp. 381–390.

[10] Tan, L., Yuan, D., Krishna, G., and Zhou, Y. /*icomment: Bugs or bad comments?*/. In *21st ACM Symp. on OS Principles (SOSP '07)*, ACM, pp. 145–158.

[11] Tartler, R., Lohmann, D., Sincero, J., and Schröder-Preikschat, W. Feature consistency in compile-time-configurable system software: Facing the Linux 10,000 feature problem. In *ACM SIGOPS/EuroSys Eur. Conf. on Computer Systems 2011 (EuroSys '11)* (Apr. 2011), ACM, pp. 47–60.

[12] Urban, M., Lohmann, D., and Spinczyk, O. The aspect-oriented design of the puma c/c++ parser framework. In *9th Int. Conf. on Aspect-Oriented Software Development (AOSD '10)*, ACM, pp. 217–221.

[13] Zengler, C., and Küchlin, W. Encoding the Linux kernel configuration in propositional logic. In *Proceedings of the 19th European Conference on Artificial Intelligence (ECAI 2010) Workshop on Configuration 2010* (2010), L. Hotz and A. Haselböck, Eds., pp. 51–56.