Efficient Extraction and Analysis of Preprocessor-Based Variability *

Julio Sincero Reinhard Tartler Daniel Lohmann Wolfgang Schröder-Preikschat
Friedrich-Alexander-University Erlangen-Nuremberg, Department of Computer Science 4
{sincero, tartler, lohmann, wosch}@cs.fau.de

Abstract

The C Preprocessor (CPP) is the tool of choice for the *implementation* of variability in many large-scale configurable software projects. Linux, probably the most-configurable piece of software ever, employs more than 10,000 preprocessor variables for this purpose. However, this *de-facto* variability tends to be "hidden in the code"; which on the long term leads to variability defects, such as dead code or inconsistencies with respect to the intended (modeled) variability of the software. This calls for tool support for the *efficient* extraction of (and reasoning over) CPP-based variability.

We suggest a novel approach to extract CPP-based variability. Our tool transforms CPP-based variability in O(n) complexity into a propositional formula that "mimics" all valid effects of conditional compilation and can be analyzed with standard SAT or BDD packages.

Our evaluation results demonstrate the scalability and practicability of the approach. A dead-block-analysis on the complete Linux source tree takes less than 30 minutes; we thereby have revealed 60 dead blocks, 2 of which meanwhile have been confirmed as new (and long-lasting) bugs; the rest is still under investigation.

Categories and Subject Descriptors D.3.4 [*PROGRAMMING LANGUAGES*]: Processors—Preprocessors

General Terms Languages, Measurement, Experimentation

Keywords Linux, Conditional Compilation, Variability

1. Introduction

Even though frequently criticized [26], the C Preprocessor (CPP) is a tool that provides an intuitive mechanism to implement variability at source code level. Its simplicity and flexibility attracts many practitioners [16]. Also, projects with strict requirements on the overhead resulting from the composition process of their variants (e.g., operating systems), rely on the overhead-free mechanisms offered by the C preprocessor.

Like many other Software Product Lines (SPLs), the Linux kernel employs the CPP language to handle variability at source code level [16]. As of this writing, the Linux kernel has around 3 million lines of code, about 20 thousand lines of code are changed each day, and there are around a thousand developers involved in each release. The daily changes include refactoring, addition and removal of features. This high volume of code changes makes keeping the consistency of the variability over all artifacts very challenging.

The Linux kernel manages its variability by using an idiomatic definition of configuration flags (features). Its variability model (the *Kconfig* [24] model) defines a set of features and their interdependencies. Later, the configuration process results in a valid selection of features expressed as CPP flags. These flags are used in the source code—with the addition of the prefix[1] CONFIG—to realize the model variability at code level. The problem is that both the CPP and the *Kconfig* tools share a common set of configuration flags, but they operate completely independent of each other: While *Kconfig* declares dependencies and other constraints on different configuration flags, CPP controls which conditional slices of code are selected for compilation. Since both tools induce different models that influence each other, this scenario easily leads to inconsistencies if not checked accordingly. Consider the patch below[2], which fixes a typo in a Linux configuration flag regarding the *hot CPU plugging* feature.

```
1 diff --git a/kernel/smp.c b/kernel/smp.c
2 index ad63d85..94188b8 100644
3 --- a/kernel/smp.c
4 +++ b/kernel/smp.c
5
6  -#ifdef CONFIG_CPU_HOTPLUG
7  +#ifdef CONFIG_HOTPLUG_CPU
```

This patch fixes a typical inconsistency problem between the variability model and the realization of implementation variability. The feature specified in the *Kconfig* model was named HOTPLUG_CPU and the developer used the flag CPU_HOTPLUG in the code; such a simple mistake went unnoticed for almost six months in the Linux kernel mainline tree. This shows that these two views onto the same system (variability *model* vs. variability *implementation*) are a nightmare with respect to maintenance and evolution. Without sophisticated tool support, they quickly lead to *dead code*, that is, code that cannot be selected by any configuration, and *zombie features*, that is, annotated code that is present in all configurations. Our initial analysis [27] of just the referential integrity between both

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

GPCE'10, October 10–13, 2010, Eindhoven, The Netherlands.

[1] By adding this prefix it is clear to the developer which ones are the configuration-controlled flags (that should not have their values changed in the code because their values originate from the configuration tool) and which are not.

[2] Git revision control commit identifier: 69dd647f

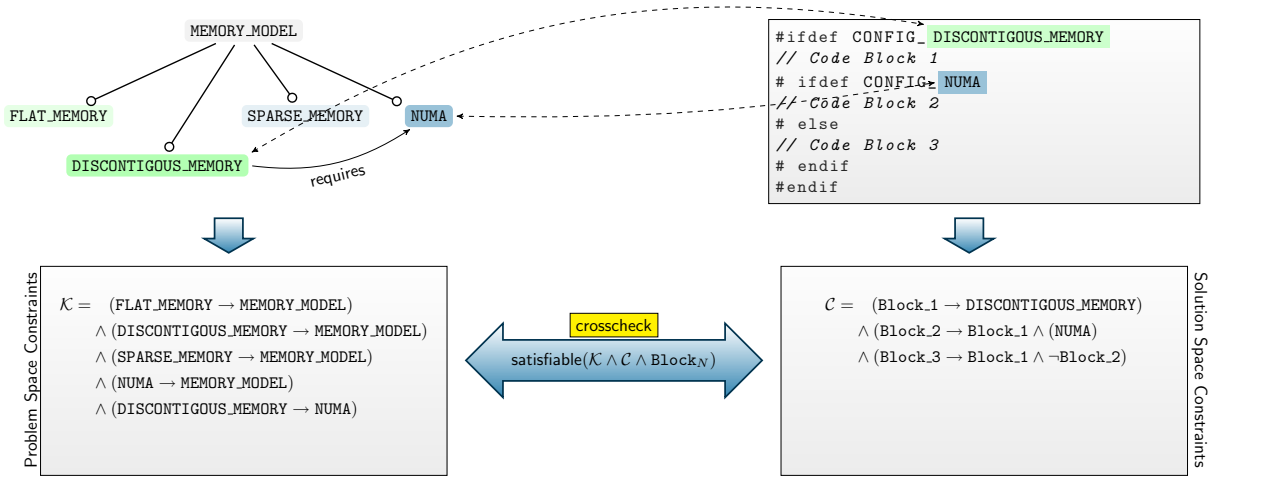

Figure 1. Inconsistency in Linux v2.6.33 in `arch/x86/include/asm/mmzone_32.h` found by our approach.

worlds in Linux has uncovered around 400 inconsistencies—and with this, we have just touched the tip of the iceberg!

In [27], we have shown that there is reason to expect various kinds of inconsistencies in the Linux implementation. In order to detect both *referential* (referenced configuration items in the *Kconfig* model that are not found in the source code and vice versa) and *semantic* inconsistencies (code that cannot be enabled or disabled by any valid configuration derived from the *Kconfig* model), we have sketched a suitable infrastructure to address this problem; in this work we present a concrete approach that is a fundamental part for the solution of this problem. While various approaches [7, 13, 14, 21, 28] compare *feature models* with *implementation variability models*, all of them either require the implementation model to be created and maintained separately or have trivial (1 to 1) mapping from model to solution. However, in many cases such models are simply not available (e.g., because of the use of legacy code, 3rd party software, etc.) and manual reengineering based on the implementation is not feasible. Instead, we use the model that is hidden in the source code as expressed by the means of `CPP` directives.

Our ultimate goal is to exploit both the configuration variability from the feature model as well as the implementation variability model from `CPP` directives, as shown in Figure 1. Nevertheless, in this work we concentrate on the right hand side of Figure 1; that is, how to extract the boolean formula from the preprocessor-based source artifacts in order to enable the crosschecking between the *variability model* and the *implementation*. Figure 1 shows a real example of inconsistency between the variability model and the implementation variability that was revealed by our approach. The `#else` block of the code snippet on the right hand side will never be selected by any valid configuration of the model shown in the left hand side. This is because the feature `DISCONTIGOUS_MEMORY` requires the feature `NUMA` as described in the variability model, and, in the code, the conditional block that depends on `NUMA` is nested in the block that depends on `DISCONTIGOUS_MEMORY`, therefore, when the outer block is selected, the first inner block will always be. Consequently, the `#else` block will never be enabled. The problem satisfiable($\mathcal{K} \wedge \mathcal{C} \wedge \text{Block}_3$), which conjugates the model constraints ($\mathcal{K}$), the implementation constraints described as `CPP` statements ($\mathcal{C}$) and a specific `CPP` block (Block_3) is *unsatisfiable*. By solving this formula we can assure that this block of code is dead.

As we will show in the remainder of this document, the sole model derived from the source artifacts not only enables the mentioned crosschecking, by it can also reveal inherent inconsistencies and defects in the source code. In this work, we present a fundamental building block for our approach to crosscheck model and implementation variability; the extraction of the implementation variability into a boolean formula enables several kinds of reasonings in order to better integrate `CPP`-based artifacts into the SPL development.

1.1 Challenges

We present an approach to extract a variability model directly from source code that uses the `CPP`, which can then be used for further analysis. This makes most sense in projects with a considerable code size; in smaller projects the implementation variability is rather clear. In Linux, we are facing a project with nearly 30,000 source files that contain over 60,000 *configuration-dependent* conditional blocks. In order to make this technique useful for both (1) developers during the regular development phase (where few files of interest), and (2) variability studies (where the whole code base is of interest), this process needs to be fast. The best technique proposed so far is the use of symbolic execution techniques on preprocessor statements [10, 15]. While these techniques are precise, they have scalability issues. We propose an approach that is both precise and fast. We envision the integration of our techniques in development environments so that variability analysis, like finding configuration defects, redundant or dead code, can be seamlessly integrated to the build process.

1.2 Contributions

In summary, we make the following contributions:

1. We *formalize* the `CPP` directives using propositional logic.
2. We present an *approach* to build the boolean formula that represents the variability of compilation units.
3. We provide a *tool chain* that implements our theoretical concepts.
4. We present a *feasibility study* by applying our tool to the Linux kernel, which revealed **4** inconsistencies in its code base (without considering additional constraints from *Kconfig*).

2. Extracting Variability from Source Code

In this section, we formalize the mechanisms for *conditional compilation*. We strictly follow the `CPP` language specification [11], Section §10.6.1, and employ propositional logic as means of ab-

Directive	Description
`#if EXPR`	conditional inclusion on the following block if `EXPR` evaluates to *true*
`defined IDENT`	Inside `EXPR`, the operator `defined` checks if the `CPP` flag `IDENT` has been defined
`#ifdef IDENT`	abbreviated form for `#if defined IDENT`
`#ifndef IDENT`	abbreviated form for `#if !defined IDENT`
`#else`	alternative block if the preceding block is not included
`#elif EXPR`	conditional inclusion on the following block if `EXPR` evaluates to *true* **and** the preceding block is not included
`#else`	alternative block if the preceding block is not included
`#endif`	terminates an conditional included block

Table 1. C Preprocessor directives related with conditional compilation

straction. We introduce several definitions that basically represent conditional-compilation concepts in terms of logic constructs. With these definitions, we build the *boolean formulas* that represent the *implementation variability model* of `CPP`-based compilation units.

The resulting variability model mimics the semantics of the `CPP` that are relevant for conditional compilation. On this basis, advanced reasoning —similar to work published for feature models [4]— can be done without an explicit variability description.

2.1 Background: Conditional Compilation

Conditional compilation relies on source code annotations that follow the `CPP` language. The semantic rules that arise from the `CPP` language specification are pretty complex. Fortunately, for a quantitative analysis of the variability induced by `CPP`, it is sufficient to focus on the rules that control *conditional compilation*.

The C Preprocessor is controlled via special preprocessor directives that are specified directly in the source code. Not every preprocessor directive is useful for implementing variability. For the purposes of this paper, we focus on the subset of the `CPP` language that includes directives shown in Table 1. The directives `#if`, `#elif`, `#ifdef` and `#ifndef` are used to declare *conditional blocks*, which are skipped or copied to the output stream depending on the result of the evaluation of their argument. This argument can either be a single `CPP` flag (such as the `IDENT` parameter of the `#ifdef` and `#ifndef` directives) or a logical expression (such as the `EXPR` parameter of the `#if` and `#elif` directives). Additionally, conditional blocks can be *nested*.

In this context the C Preprocessor is used to insert slices of source code into its output stream according to the set of defined `CPP` flags. Technically, these flags are defined at compilation time either as command-line parameter or by using `#define` directives. In this paper, we discuss boolean configuration flags only. In practice, configuration flags can also be integers that are compared and checked during preprocessing.

The following listing shows the most trivial example of a conditional block:

```
#ifdef CONFIG_SMP
//block 1
#endif
```

The number of variants that can be composed from this code snippet is two: Either the item `CONFIG_SMP` is set, then `block 1` is selected, or if `CONFIG_SMP` is not set, then `block 1` is skipped. The next listing shows alternative blocks:

```
#ifdef CONFIG_SMP
//block 1
#else
//block 2
#endif
```

Here, we have again in total two variants that can be composed. Depending on the definition of `CONFIG_SMP`, either `block 1` or `block 2` is selected. Another use of if-groups can be seen in the following listing:

```
#ifdef CONFIG_SMP
//block 1
#elif defined CONFIG_APIC
//block 2
#endif
```

In this example, there are three variants that can be composed: If both items are unset, then none of the blocks are selected. `Block 2` is selected only if `CONFIG_SMP` is unset and `CONFIG_APIC` is set. If both flags are set, then `block 1` gets precedence.

Important for the understanding of our approach is the concept of *block*. It represents the snippet of code that is controlled by conditional compilation. The variability of a `CPP`-based compilation unit is basically the set of all possible block combinations that the `CPP` can generate for the compilation unit.

2.2 Basic Definitions

DEFINITION 1. *[Configuration] Given n boolean configuration flags found in a compilation unit u, a **configuration** is the vector $\vec{f} = f_1, \ldots, f_n$, where f_i is a boolean variable representing the assignment of the i-th configuration flag.*

Configuration flags control the selection of conditional blocks in a compilation unit. Each member of the vector $\vec{f}$ represents the flag assignments as seen by `CPP`. For example, the file shown in Listing 1 makes use of three different flags; therefore its configuration vector is defined as $\vec{f} = \{A, B, C\}$. If this file is compiled with only the flag B is defined, according to our definition, the configuration vector is set as $\vec{f} = \{false, true, false\}$.

DEFINITION 2. *[Block Selection] Given a compilation unit u with m conditional blocks, the **block selection** is the vector $\vec{b_u} = b_1, \ldots, b_m$, where b_i is a boolean variable that represents the presence of the i-th conditional block.*

```
#if (defined A || defined B || defined C)
//block 1
# if defined A
  //block 2
# elif defined B
  //block 3
# else
  //block 4
# endif
#endif
```

Listing 1. Example of Conditional Blocks

Each variable in this vector represents a conditional block in the compilation unit u. The block selection represents the selection as done by the CPP; that is, the members of the vector $\vec{b_u}$ that represent selected blocks are set to true and members representing skipped blocks are set to false. For example, the block selection vector of the file shown in Listing 1 is defined as $\vec{b_u} = \{b_1, b_2, b_3, b_4\}$. If this file is compiled with only flags A and B set, that is, using the configuration vector $\vec{f} = \{true, true, false\}$, the resulting preprocessed file will contain blocks b_1 and b_2, consequently, the block configuration vector will be $\vec{b_u} = \{true, true, false, false\}$

Using Definition 1 and Definition 2, the process of applying a set of configuration flags $\vec{f}$ to a compilation unit u that contains m conditional blocks $\vec{b_u}$ can be expressed by the following function (which is compilation unit dependent, therefore, the index u) $\mathcal{P}_u$:

$$\mathcal{P}_u(\vec{f}) \mapsto \vec{b_u} \tag{1}$$

This function represents the *mapping* from a given *configuration* $\vec{f}$ to a specific *block selection* $\vec{b_u}$ if preprocessing is performed by the CPP tool. Since the basic nature of propositional formulas is to work on binary decisions, our approach does not attempt to calculate the function $\mathcal{P}_u$ directly. Instead, a helper function $\mathcal{C}_u$ that *checks* the behavior of CPP for a given configuration is built using propositional logic. Our approach allows deducing the variability indirectly with the following checker function:

DEFINITION 3. *[Checker Function] Given a function $\mathcal{P}_u$ that represents the conditional compilation semantics of the CPP language, a configuration vector $\vec{f}$ and a compilation unit u with the block selection vector $\vec{b_u}$, the* ***checker function*** *$\mathcal{C}_u$ is defined as:*

$$\mathcal{C}_u(\vec{f}, \vec{b_u}) \rightarrow \begin{cases} \text{true} & \iff \quad \mathcal{P}_u(\vec{f}) = \vec{b_u} \\ \text{false} & \iff \quad \mathcal{P}_u(\vec{f}) \neq \vec{b_u} \end{cases} \tag{2}$$

In order to extract the *variability model* from the compilation unit u, the function $\mathcal{C}_u$ must be constructed. According to the specification of the CPP language there are three preconditions for the inclusion of a conditional block during preprocessing:

1. The expression that controls conditional inclusion must evaluate to true, otherwise the block is completely skipped [11, §6.10.1, 1].
2. A nested block can be selected if and only if its parent is also selected [11, §6.10.1, 5].
3. For if-groups (groups that contain the directive `#elif` or `#else`), the blocks are evaluated in declaration order. The first that evaluates to true is selected, all others are skipped. The `#else` conditional is selected when none of the predecessor blocks of the if-group evaluates to true [11, §6.10.1, 5].

These three preconditions control the *presence condition* of conditional blocks. Together, they are *necessary* and *sufficient*, which means that if they are valid for a specific block, this block will necessarily be selected by the CPP. They can be directly translated to the following helper functions:

expression(b_i) Given a block b_i, the function *expression*(b_i) returns the logical expression as specified in the block declaration. Example: For the first block in Listing 1, the function *expression*(b_1) returns: $A \vee B \vee C$.

parent(b_i) Given a block b_i, the function *parent*(b_i) returns the logical variable that represents the selection of its parent. If the block is not nested in any other block, then the result is always *true*. Example: For the third block in Listing 1, the function returns: b_1.

noPredecessors(b_i) Given a block b_i, the function *noPredecessors*(b_i) returns the negation of the disjunction of all its predecessors (logical variables representing blocks) in an if-group. Example: For the fourth block of Listing 1 (*noPredecessors*(b_4)), the function returns: $\neg(b_2 \vee b_3)$.

Note that the helper functions do not return boolean values, they actually return[3] logic expressions containing logic symbols and operators. Using these functions, the *presence condition* for conditional blocks is constructed as follows:

DEFINITION 4. *[Presence Condition] A conditional block b_i is selected by CPP if and only if the following conjunction holds:*

$$\mathcal{PC}(b_i) = expression(b_i) \wedge noPredecessors(b_i) \wedge parent(b_i) \tag{3}$$

The presence condition $\mathcal{PC}(b_i)$ allows us to build a boolean formula that can be used to check if a specific block is to be selected for a configuration $\vec{f}$. Combining all presence conditions of a compilation unit allow us to build the checker function:

$$\mathcal{C}_u(\vec{f}, \vec{b_u}) = \bigwedge_{i=1..m} b_i \leftrightarrow \mathcal{PC}(b_i) \tag{4}$$

The checker function returns a boolean formula containing $n + m$ variables, where n is the size of the vector $\vec{f}$ and m is the size of the vector $\vec{b_u}$. The function is simply the conjunction of the presence condition of all blocks of a specific compilation unit. It is important to note that the vector $\vec{f}$ does not explicitly appear on the right hand side of the function because it will appear implicitly a result from function $expression(b_i)$ when calculating the presence condition of a block. Also important, is the biimplication between a block and its presence condition. When a block has its presence met, it is not only *allowed* to be enabled, but it will *necessarily* be enabled by the CPP, it means that not only the block implies its presence condition, but the presence condition also implies the block, therefore the biimplication.

This function is satisfiable for the variable assignments that correspond to valid behaviors of the CPP. The checker function is able to verify if a specific block selection $\vec{b_u}$ represents the resulting preprocessing when given a configuration $\vec{f}$ and a compilation unit u. Consequently, the term *variability* is defined as follows:

DEFINITION 5. *[Variability] Given a compilation unit u and the checker function $\mathcal{C}_u$, the* ***variability*** *$\mathcal{V}$ of a compilation unit u is the set of all block selections $\vec{b_u}$ for which there exists a selection $\vec{f}$ such that $\mathcal{C}_u$ is satisfied:*

$$\mathcal{V} = \{\vec{b_u} \mid \exists \vec{f} : \mathcal{C}_u(\vec{f}, \vec{b_u})\} \tag{5}$$

2.3 Calculating Variability with *Checker Function*

The *variability* $\mathcal{V}$ (Formula (5)) is therefore the set of vectors $\vec{b_u}$ that can be mapped by the function $\mathcal{P}_u(\vec{f})$ (Formula (1)) with at least one *input configuration* $\vec{f}$. This set can be calculated by generating

[3] For some function calls, the returned expression is trivial. For example, the call of *parent()* for a top-level block, *expression()* for an else block, or *noPredecessors()* for a block that does not belong to a if-group. In all cases, we can return the constant `true` and (optionally) avoid its inclusion in the resulting presence condition.

```
1  #if defined A || defined  B || defined C
2  # if defined ( A && B )
3  # elif defined ( A && C )
4  # elif defined ( B && C )
5  #   ifdef C
6  #   else
7  #   endif
8  # else
9  # endif
10 #endif
```

Listing 2. Source Code Example with all language features for Conditional Compilation.

the Checker Function $\mathcal{C}_u$ (Formula (4)), and solving the satisfiability problem that follows from Formula (2). The complexity to *produce* these constraints therefore effectively scales linearly (O(n)) with the number of conditional blocks. *Solving* the resulting formulas of course remains NP complete.

The result of this method is therefore a propositional formula that represents the implementation variability. It serves as a building block for further reasoning techniques that can be integrated into software engineering tools. The formula that we generate is not a visual or editable model; it is a logic representation of the variability as described by the CPP annotations, similar to other approaches [4, 6, 7, 20] that translate visual models (e.g., feature models) into boolean formulas.

Concrete Example

In order to explain how the construction of the *checker function* works in practice, we show the results for a combined example. For additional examples, please refer to [25]. In order to make the understanding easier, we use the following annotations:

- The symbol ⟲ at the end of each line specifies in which iteration the presence condition of the corresponding conditional block is generated.
- Each clause has been marked with an overbrace ($\overbrace{\ }^{\blacktriangleright f()}$) in order to indicate which helper function provided each part of the resulting presence condition.

The example shown in Listing 2 uses all semantic elements of the CPP language that are relevant for conditional compilation. The complexity of the example is typical for real world code. From this source code, we construct the *checker function* as shown in Figure 2.

The first iteration generates the clause for the top level block b_1. The clause ▶ *expression()* generates a biimplication from the block b_1 on its expression, i.e., the block is selected if and only if its expression holds. This clause corresponds to the helper function *expression(b_i)* as described in Section 2.2.

The biimplication for block b_2 (lines 2-3) is generated in iteration 2. The first clause (marked with ▶*parent()*) ensures that if block b_2 is selected, then b_1 must be as well. This clause corresponds to the helper function *parent(b_i)*. Clause ▶*expression()* is constructed in the same way as for block b_1.

For block b_3 (lines 3-4) all three helper functions contribute to its presence condition. Because this block is a successor of block b_2 and must not be considered by CPP in case block b_2 is selected, the clause ▶*noPredecessor()* ensures that b_3 can only be selected if b_2 is not.

Blocks b_5 (line 5) and b_6 (line 6) form a group of blocks nested inside block b_4 (line 4-8). Therefore, iteration 5 and 6 generate the clauses ▶ *parent()*, ▶ *expression()* and ▶ *noPredecessor()* for

$$\mathcal{C}_u(\{A,B,C\},\{b_1,\ldots,b_7\}) = \bigwedge_{i=1..7} b_i \leftrightarrow \mathcal{PC}(b_i) =$$

$$\bigwedge_{i=1..7} b_i \leftrightarrow expression(b_i) \wedge noPredecessors(b_i) \wedge parent(b_i) =$$

$$\Big(b_1 \leftrightarrow (\ \overbrace{A \vee B \vee C}^{\blacktriangleright expression(b_1)}\ )\Big) \quad (⟲\ 1)$$

$$\wedge\ \Big(b_2 \leftrightarrow (\ \overbrace{(A \wedge B)}^{\blacktriangleright expression(b_2)}\ \wedge\ \overbrace{(b_1)}^{\blacktriangleright parent(b_2)}\ )\Big) \quad (⟲\ 2)$$

$$\wedge\ \Big(b_3 \leftrightarrow (\ \overbrace{(A \wedge C)}^{\blacktriangleright expression(b_3)}\ \wedge\ \overbrace{\neg(b_2)}^{\blacktriangleright noPredecessor(b_3)}\ \wedge\ \overbrace{(b_1)}^{\blacktriangleright parent(b_3)}\ )\Big) \quad (⟲\ 3)$$

$$\wedge\ \Big(b_4 \leftrightarrow (\ \overbrace{(B\ \wedge\ C)}^{\blacktriangleright expression(b_4)}\ \wedge\ \overbrace{\neg(b_2 \vee b_3)}^{\blacktriangleright noPredecessor(b_4)}\ \wedge\ \overbrace{(b_1)}^{\blacktriangleright parent(b_4)}\ )\Big) \quad (⟲\ 4)$$

$$\wedge\ \Big(b_5 \leftrightarrow (\ \overbrace{(C)}^{\blacktriangleright expression(b_5)}\ \wedge\ \overbrace{(b_4)}^{\blacktriangleright parent(b_5)}\ )\Big) \quad (⟲\ 5)$$

$$\wedge\ \Big(b_6 \leftrightarrow (\ \overbrace{\neg(b_5)}^{\blacktriangleright noPredecessor(b_6)}\ \wedge\ \overbrace{(b_4)}^{\blacktriangleright parent(b_6)}\ )\Big) \quad (⟲\ 6)$$

$$\wedge\ \Big(b_7 \leftrightarrow (\ \overbrace{\neg(b_2 \vee b_3 \vee b_4)}^{\blacktriangleright noPredecessor(b_7)}\ \wedge\ \overbrace{(b_1)}^{\blacktriangleright parent(b_7)}\ )\Big) \quad (⟲\ 7)$$

Figure 2. Checker function for Listing 2.

both blocks with the corresponding expressions and using the same parent (b_4).

Block b_7 (line 6) is treated in the last iteration. Since this is an #else block without an expression of its own, only the clauses ▶*parent()*, and ▶*noPredecessor()* are generated.

We identify the following solutions that satisfy this *checker function*:

$$\Big\{\{\emptyset,\emptyset\},\{C,b_1,b_7\},\{B,b_1,b_7\},$$
$$\{B,C,b_1,b_4,b_5\},\{A,b_1,b_7\},\{A,C,b_1,b_3\},$$
$$\{A,B,b_1,b_2\},\{A,B,C,b_1,b_2\}\Big\}$$

Note that the tuples are in the form $(\vec{f}, \vec{b})$, but for an easier-to-read representation we show only the vector members that are set to *true*, therefore, the empty set $\emptyset$ represents a vector where all member are set to *false*. From these solutions, we identify as *variability*:

$$\mathcal{V} = \Big\{\{b_1,b_7\},\{b_1,b_4,b_5\},\{b_1,b_7\},\{b_1,b_3,\}\{b_1,b_2\}\Big\} \quad (6)$$

The set $\mathcal{V}$ corresponds to all possible configurations (different block selections) that the CPP can generate for the file shown in Listing 2. This is possible to be calculated due to the characteristics of the boolean formula shown in Figure 2, which is built taking into consideration the structure of the input file and the semantics of the CPP language, as a result, it is a compact boolean formula that mimics the CPP preprocessor for a specific source file.

2.4 Reasoning on Implementation Variability Models

The formula explained in Section 2.3 represents the *implementation variability* model of a compilation unit. Based on this model, similar reasoning techniques like already proposed for feature models [4, 6, 29] can be performed as well.

Number of variants: using the formula $\mathcal{C}_u$ built by our approach, we are able to calculate the number of all possible different configurations a compilation unit can be translated to, for example *sat_count($\mathcal{C}_u$)*.

Calculating all variants: using $\mathcal{C}_u$ we can also calculate all valid variable assignments, for example *all_sat($\mathcal{C}_u$)*. Using this, we can classify which assignments lead to the same configuration, and, as a result, the unique set of valid block configurations.

Validation: using $\mathcal{C}_u$, we check for internal consistency, that is, checking for each block of a compilation unit if it is selectable by at least one valid configuration, for example *satisfiable($\mathcal{C}_u \wedge b_i$)*. External consistency (e.g., implementation model versus feature model) can also be performed to check if each block is selectable by at least one valid configuration from the feature model. For example, using *FM* as the boolean formula representing a feature model, we can calculate *satisfiable($\mathcal{C}_u \wedge b_i \wedge FM$)*.

Reasoning about edits: the algorithms for reasoning about edits to feature models presented by Thum [29]—for finding out how a refactoring on a feature model (in our case on the `CPP` directives) impacts the model variability—can also be applied to the formula $\mathcal{C}_u$.

Filtering and partial configurations: reducing the number of variation points is an effective means to help understanding the implementation. By preconfiguring a subset of the available configuration flags (assignments to some variables of $\vec{f}$) to constrain the $\mathcal{C}_u$ formula, both simplified `#ifdef` expression of conditional blocks as well as unselectable blocks can be queried and used to provide such a *partial* view

3. Analyzing Implementation Variability

With the theoretical insight on how to extract a variability model from `CPP` statements from the previous section, in this section we discuss and evaluate how to apply our approach to SPLs.

3.1 Implementation Overview

Our approach is organized in a frontend and a backend part such that it can be applied to any kind of software using `CPP` annotations. The frontend analyzes and extracts the expressions and structure of `CPP` statements in an *implementation variability database*. On this basis, the backend generates the formulas as presented in Section 2.3.

As frontend, we have written the tool **source2rsf**, which extracts `CPP` directives from source files. It is based on the existing preprocessor implementation found in the *sparse* [18] analyzer. The tool captures the details of block declarations such as `#ifdef` expressions, nesting, successors, predecessors, etc. This data is saved in text files and contains all the required input information for building the constraints of the checker function from Section 2.3. The backend is implemented with our tool **undertaker**, which uses the concepts from Section 2.2 to construct the *checker function* described in Section 2.3. It produces propositional formulas that can be further processed with standard Binary Decision Diagram (BDD) packages like buddy [5] or SAT solvers like limmat [17]. As already studied [4, 20], the reasoning techniques described in Section 2.4 have drastic performance fluctuations depending on the used realization technique. For example, calculating the *number of variants* can be efficiently done by BDDs, whereas SAT solvers are more suitable for *validation* checking. For this reason, we provide both variants of our backend.

3.2 Analyzing Source Artifacts vs. Compilation Units

The frontend can be employed in two distinct modes, which have a large impact on the results.

In **compilation-unit mode**, the frontend is used together with or in place of compiler invocations. In this mode, the frontend includes

```
104 #ifdef CONFIG_PROC_FS
105 #include <linux/proc_fs.h>
106 #include <linux/seq_file.h>
107 #endif
```

Listing 3. Excerpt of `net/ipv4/igmp.c`: The headers `proc_fs.h` and `seq_file.h` are processed the (additional) condition that the configuration option `CONFIG_PROC_FS` is set.

referenced headers by processing `#include` directives in order to detect variability defects "across" source artifacts, an approach that also is commonly used by static analysis tools, such as the proposed approaches to analyze CPP-induced variability by means of symbolic execution [10, 15]. While analyzing variability "as the compiler sees it" clearly has its merits, this approach also has serious practical limitations. Firstly, the tools need to be integrated in the build scripts of the software to be analyzed, which can be a tedious task. Secondly, the coverage is often unclear, as in nontrivial SPLs (such as Linux) not only the preprocessor, but also the build scripts themselves are employed to implement variability; the set of source artifacts that actually gets compiled depends on the configuration. Thirdly, in large SPLs the expansion of `#incude` files into many compilation units can lead to a *significant* growth of the code lines to be analyzed, with consequences for the analysis computation times. We shall see the results for Linux in Section 3.5.

For this reasons, we also support a **source-file mode**. In this mode, the frontend simply processes each and every source artifact. This does not require integration into the build scripts, maximizes the analysis coverage and avoids unnecessary processing of headers that are included by several other implementation artifacts. The limitations of this approach are that it (a) misses potential changes to `CPP` identifiers induced by the build system itself and furthermore (b) misses configuration defects that spread across multiple source artifacts.

Listing 3 shows a typical example how (b) manifests in Linux. The constraints that are in effect in lines 105 and 106 of `igmp.c`, which use the `#include` directive to textually include other source artifacts, are taken into account only in *compilation-unit mode*, but missed in *source-file mode* – the 'imported" variability from the header files `proc_fs.h` and `seq_file.h` is not taken into account. For reasoning techniques like *reasoning on edits* or *calculating partial configurations* (cf. Section 2.4), this fact may not matter, but in general, this can skew the results. We shall discuss the effects for analyzing Linux in Section 3.5.

3.3 Efficient Analysis for Very Large Variability Models

Our approach calculates a logical formula that needs to be solved. In this formula, each conditional block and each configuration flag is represented by a logical variable. As solving this formula is essentially a satisfiability problem, which is NP-complete, a very important question to be answered is how this approach scales for very large SPLs. In order to handle the problem size, we have applied two optimizations.

The first optimization is to tailor the analysis to only those `#ifdef` statements that are actually controlled by the configuration process. Essentially, only these statements constitute the *interesting* variation points for our analysis. The exact set of configurability-related `CPP`-identifiers depends on the configuration tool, but generally can be easily determined. In the case of Linux this is even prescribed by a coding guideline: configuration-related preprocessor flags are always prefixed with `CONFIG_`. Therefore, only `#ifdef` expressions that contain one or more `CONFIG_`-prefixed identifiers mark *interesting* variation points. Our tool uses an *optional* matching

```
1  #ifdef CONFIG_SCHEDSTATS
2  // code ommitted
3  #ifdef CONFIG_SMP
4  // code ommitted
5  #endif
6  // code ommitted
7  #else /* !CONFIG_SCHEDSTATS */
8  // code ommitted
9  #endif
10
11 #ifdef CONFIG_SCHED_SMT
12 // code ommitted
13 #else
14 // code ommitted
15 #endif
```

Listing 4. Excerpt of `kernel/sched.c`, lines 307ff: Two `#ifdef`-clouds, which can be processed independently.

expression to automatically detect—and skip—conditional blocks that are not of interest.

The second optimization is to split the variability model of a compilation unit into smaller models that can be validated *independently*. The smaller models contain only blocks that are related. A block is related to the block it is nested in, and to its alternative blocks (in an if-group). We call these groups of blocks `#ifdef`-*clouds*. Consider Listing 4, which is taken from the source code of the Linux scheduler. The conditional blocks that start with `CONFIG_SCHEDSTATS` (lines 1 to 9) and `CONFIG_SCHED_SMT` (lines 11 to 15) are independent, because the `CPP`-statements from the first group cannot take effect on the number of variants that can be produced from the second group—and vice versa. This means that we can calculate and solve the logical formulas for both clouds independently, which effectively reduces the problem size. For example, when we consider the file shown in Listing 4 as a whole, the resulting boolean formula contains 8 variables (5 blocks + 3 identifiers). In contrast, when we analyze the clouds independently, the first will produce a formula with 5 variables (3 blocks + 2 identifiers) and the second with 3 (2 blocks + 1 identifier). For very large SPLs (such as Linux) this optimization has a dramatic effect (and actually makes the approach feasible at all). Recall that to check the validity of blocks we have to solve the formula *satisfiable*($\mathcal{C}_u \wedge b_i$) for every block of the file (or in the optimized version for every block of a cloud). The fewer variables in the formula $\mathcal{C}_u$ the faster we can solve the satisfiability problem.

3.4 Case Study #1: Variability Analysis of the Graph Product Line

In order to evaluate how well the various reasonings presented in Section 2.4 can be performed we conduct two case studies. The first study case is performed with the well-known Graph Product Line (GPL) [19]. We used the `CPP`-based version of this project that was generated with the CIDE [13] tool. In this evaluation, we implement the reasoning *number of variants* and *calculating all variants* in order to get insights into the implementation variability and compare it with the feature model. These two reasonings require the BDD variant of our backend implementation.

The feature model of the GPL contains **20** features, of which 140 valid variants can be created. Using our approach we now analyze the variability of the source code. For this, we analyze all **10** java files with our tools. Table 2 summarizes the number of variants, blocks and `#ifdef` clouds per implementation file. Interestingly, the variability is very concentrated in two *hot spots*, namely the files

Implementation Variability

Source File:	V	C	B
`Neighbor.java`	1	0	0
`Edge.java`	2	4	4
`WorkSpace.java`	2	1	1
`Main.java`	3	1	2
`NumberWorkSpace.java`	3	1	2
`RegionWorkSpace.java`	4	1	3
`CycleWorkSpace.java`	4	1	4
`WorkSpaceTranspose.java`	6	1	5
`Vertex.java`	2,064	26	79
`Graph.java`	5,760	43	121
`Total`	10,367	79	221

Table 2. Implemenation Variability Analysis of the Graph Product Line. Where **V** is the number of variants, **C** the number of `#ifdef` clouds, and **B** the number of blocks.

Mode:	Source File	Compilation Unit
`#ifdef` clouds	25,844	$3.67 \cdot 10^6$
Coverage	100%	77%
t(frontend)	$>$ 20 min	$\approx$ 5.5h
t(Validation w/ SAT)	$>$ 6 min	$\approx$ 6h
t(Validation w/ BDD)	$>$ 275 min	N/A

Table 3. Evaluation Results for Linux Kernel Version 2.6.33. The columns compare the two different approaches *scanning all source files* and *scanning all compilation units*.

`Vertex.java` and `Graph.java`. By combining all possible variants of all implementation files we are able to calculate the total number of variants that is described by the code base: 10,367. This represents the total number of variants that can be *generated* with the `CPP` if all 20 `CPP` identifiers could be selected and deselected in an arbitrary fashion.

Even though this number is way below the 2^{20} combinations that are theoretically possible, it is also way above the 140 variants that are specified by the feature model. In fact, the variability described by the feature model covers less than **2%** of the variability described in the source code. This dramatically demonstrates the "semantic poorness" of `CPP`-based variability. How the `CPP` is employed to implement variability in some source code does not teach us much about the actually *intended* variability. This also indicates that an automatic extraction of feature models from the implementation (an idea that frequently pops up) is probably a very challenging task.

Our conclusion is that both models of variability, feature model and implementation, have very different characteristics and need to be investigated together. Our approach facilitates this for SPLs that employ the `CPP` for the implementation of variability. For example, we can use $\mathcal{C}_u$ to check if all different variants (from the feature model) will actually differ in the generated code . The importance of operations that search for inconsistencies between model and implementation (short: *crosscheckings*) has been confirmed by several authors [7, 21, 28]. Our approach contributes to this body of work by enabling—through automatic extraction—the use of the implementation variability in form of `CPP` directives in the reasoning processes.

3.5 Case Study #2: Finding Dead Code in Linux

To learn how well our approach performs on real software projects, we apply it to a very large SPL [23]. According to [16], the Linux kernel code is one of the largest configurable software projects that is available for analysis, which makes this code base an ideal test

bed for our approach. The evaluation is done on a modern Intel Quad Core workstation with 2.83 GHz and 8 GB RAM. Table 3 summarizes the results.

In order to get insight about the practical differences of the two modes of operation as defined in Section 3.2, we apply both approaches to Linux version **2.6.33**. The frontend extracts preprocessor statements from the source code and stores them in a way usable for the backend.

In *source-file mode*, the frontend processes all 26,765 source files that match the pattern `*.[chS]`, which results in 25,844 `#ifdef` clouds. With this approach, we gain full source-tree coverage and can create all logical formulas in about half an hour.

In *compilation-unit mode*, the frontend extracts preprocessor statements from expanded compilation units. We compile a kernel with the assumably "largest configuration", which is *allmodconfig* on the `x86` architecture. During this analysis, ony 77% of all source files are accessed, the remaining 23% source files of the Linux tree belong to other configurations only. Note that while the coverage is lower, the number of `#ifdef` clouds is orders of magnitudes higher (3.6 million). The reason for this is that a compilation unit in general consists of more than one source file, and many (if not most) header files are included textually in many different compilation units. This leads to the fact that conditional blocks in the same header files have to be processed several times.

On this basis, we implement a consistency check that *validates* the extracted implementation variability model. It essentially detects conditional blocks that are not selected under *any* possible input configuration, which we call *dead blocks* [27]. Since this check is a satisfiability problem, the SAT variant of the *undertaker* tool is sufficient to perform this reasoning. However, we also use the BDD variant *in addition* to compare the performance of both backends. It turns out that both the frontend as well as the backend of our toolchain take their time: about half an hour when employing source-file mode and over 12 hours in compilation-unit mode. Note that these numbers describe the times necessary to analyze the whole code base, however, the source files are analyzed independently of each other. This means that if our approach is employed during implementation, the developer can find problems in specific files in less then one second. Nevertheless, these numbers also show that with our approach this kind of complete analysis becomes feasible even with very large scale SPLs.

Naturally, the BDD and SAT variants of the backend find the same dead blocks. We reveal **60** dead blocks in compilation-unit mode and **4** in source-file mode, two of which *meanwhile* have been confirmed as **new** bugs (the other two are still under investigation) by the upstream Linux developers, who have accepted our corresponding patches[4]. These bugs have remained undetected in the Linux source code for more than **5** years!

Analyzing Compilation Units Revisited

The variability model extraction in compilation-unit mode requires a considerably larger amount of time than processing all source files individually. However, as the two modes of operation produce different results *only* [5], for compilation units that use conditional `#include` directives (as shown in Listing 3), we have analyzed how many compilation units actually use this construction. It turns out that from 7,081 processed compilation units, less than 5% actually use conditional `#include` directives. This means that, a both fast and complete analysis can be achieved by first processing all source file individually, then determining which compilation units use conditional `#include` directives, and finally reprocess only these ones in expanded compilation mode.

[4] `http://lkml.org/lkml/2010/4/26/90`, `http://lkml.org/lkml/2010/4/26/87`

[5] Note that the 4 dead blocks found with the source-file mode were also found by the compilation-unit mode, the other 56 dead blocks result from specific combinations of source files through conditional inclusion.

4. Discussion

The *implementation variability model* that we automatically extract from `CPP`-based source files can be used in a variety of ways to improve the SPL development. We have shown that our implementation can be used to analyze the variability of an SPL implementation as well to reveal inconsistencies. Nevertheless, our model can also be combined with several state-of-the-art techniques to support different areas of the SPL development. This section is to be understood as an analogy to various related approaches that apply reasoning and similar techniques to *feature models*. Future work include picking up ideas sketched in this section for the development and/or integration of new tools to support further analysis and bug finding techniques.

4.1 Understanding the Variability of Software Product Lines

The management of variability is crucial for SPLs. However, understanding the variability spread over many artifacts can be very complicated, and, without appropriate tool support, it can be even hopeless. In order to answer questions like *Is the code base flexible enough to build all variants allowed by the feature model?* an appropriate representation of the *implementation variability model* is absolutely required. Such an implementation variability model enables SPLs tools to be aware whether the feature model is expressive enough, or efficiently check if any two valid and distinct variants produce the same configured source tree without actually running the `CPP`. Both cases are an indicative of bugs; either in the feature model or in the implementation.

Another important matter for the understanding of SPLs is *feature interaction*. Features that have no relationship in the feature model often interact with each other in the implementation. An implementation model that contains the information regarding how code depends on features can also help with the understanding of (static) feature interactions. Moreover, this information from the implementation model is an important element for metrics, for example, for determining the level of interaction of features (with how many other features it appears on block declarations, etc.), or for the level of scattering of a feature (in how many block declarations it appears), among others.

4.2 Maintenance and Evolution

In order to avoid the introduction bugs during refactoring, checking the consistency of the variability among different artifacts is very important. Therefore, tool support for assessing the *effects* of refactoring on the variability is indispensable.

We believe that our model can support maintenance and evolution of SPLs by automatically detecting several kinds of inconsistencies in different ways. For example, while working on new code or changing the feature model, our techniques can help the developer to check if his changes introduce any consistency problems. Moreover, our tool can also be used periodically by an SPL specialist to check for inconsistencies or simply to assess the current state of the SPL variability in order to plan the SPL's evolution steps, for example addition of new feature, removal, refactoring, etc.

4.3 Testing

Techniques for testing SPLs can also take advantage of our approach. For example, with a specific configuration (e.g., from the feature model) at hand it is possible to assess how many (and which) conditional blocks of a specific compilation unit will be covered by a given test case. This information can be used to determine if the configuration for the test is adequate (if it covers the required

blocks to be tested) or not. The other way around, that is, discovering the number of different (and what they look like) configurations that are necessary for covering all blocks of a specific compilation unit can also be automatically determined using the implementation variability model.

4.4 Limitations

The algorithm we have presented in this article has two obvious limitations. First, the subset of the CPP language contains conditional compilation with *boolean configuration flags only*. As far as conditional compilation is concerned, the decision if a block is selected or not is intrinsically boolean. Therefore, each expression that involves nonboolean flags and operators (e.g., ==, <, etc.) is replaced by a new free logic variable. For example, the expression `#if CONFIG_BUFFER > 1024` is rewritten to `#if defined CONFIG_COMPARATOR_1`.

The CPP language allows *undefinition and redefinition of flags* in any line of the source code. This language detail is not currently considered by the presented algorithm, but could be supported by adding extra conjunctions to the *checker function*, more specifically, to the *expression(b_i)* function. Here, for each expression, the function would need to take undefinitions and redefinitions of the undefined or redefined CPP flags into account.

While both cases are rather straightforward to implement, supported by experience with the Linux kernel presented in this article, we believe that even the current implementation of the algorithm is already well suited for many of the use cases we outlined above.

5. Related Work

Analyzing large scale software projects with transformation systems are related to our approach; DMS [2] proposed by Baxter is probably the most renowned one. In context of this framework, an approach has been published [3] that aims at simplifying CPP statements by detecting dead code and simplifying CPP expressions. Unlike our approach, this work uses concrete configurations to evaluate the expressions partially [12]. In contrast to that, our work does not require concrete values for CPP identifiers, but produces a logical model in form of a propositional formula that can either be evaluated directly or can be combined with further constraints like the ones extracted from the feature model. We believe that our approach would fit great in the DMS framework.

Analyzing conditional compilation with symbolic execution has been proposed by Hu et al. [10] in 2000. This approach maps conditional compilation to execution steps: Inclusion of headers map to *calls*, alternative blocks to branches in a control flow graph (CFG), which is then processed with traditional symbolic execution techniques. Lattendresse [15] improves this technique by using *rewrite systems* in order to find presence conditions for every line of code. Similarly to our approach the presence conditions of all conditional blocks are calculated during the process as well. However, symbolic evaluation does not actually calculate the variability as per definition 5, which is required for the reasoning techniques that we present in section 2.4. While it would be possible to use the presence conditions learned with symbolic execution and construct the checker function (Formula 4), we present an lightweight approach that does scale (the algorithm for generating the boolean formula grows linearly with respect to the number of blocks) to code bases of millions of code, as presented by our evaluation on the Linux kernel in Section 3.5.

Other approaches extend the grammar of the C/C++ parser by CPP directives. Badros et al. [1] propose the PCp^3 framework to integrate *hooks* into the parser in order to perform many common software-engineering analyses. This technique is technically similar to our integration into the tool *sparse* [18]. However, the focus is on mapping between unprocessed and preprocessed code, whereas our work aims at mapping towards higher level models. Garrido [9] extends the parser with the concept of conditional abstract syntax tree (AST)s, which enables preprocessor-aware refactorings on CPP-based source files. We believe that this work can be combined with our approach to further assist software engineering tools.

The crosschecking between different variability models is also related to our work. Metzger et al. [21] present several formalizations that allow for consistency checks between variability models. Czarnecki et al. [7] present an approach to checking the consistency of feature-based model templates, that is, checking consistency of feature models and model templates. Thaker et al. [28] present an approach to the safe composition of product lines, it is much related to our idea of crosschecking. However, their approach exploits the structure of the code, which is based on the *feature-oriented software development*, therefore, it can not be used in projects that rely on the CPP like the Linux kernel. We believe our technique is complementary to these approaches; as they also use propositional logic, we could provide the implementation model directly as a formula, avoiding its representation in a specific model format.

The evaluation of variability from models is also related to our work. Czarnecki et al. [8] present an approach to transform logic formulas into *feature models*. This technique could be combined with ours in order to automatically generate models from the boolean formula that is generated with our algorithm. Benavides et al. [4] present several techniques for *reasoning* on feature models after transforming them into boolean formulas. We have shown how the same kind of reasoning can be applied to the resulting implementation variability model of our approach. She et al. [22] presents an approach to convert the Linux Kconfig model into feature models. On the one hand, complementary to our work, it could be used in combination with our algorithm to provide the crosscheckings. On the other hand, contrary to our work, the source code files—consequently the CPP information— are not taken into consideration.

6. Conclusions and Future Work

This article has presented an efficient approach to the automatic extraction and analysis of the implementation variability described as CPP directives. Our novel *extraction process* scales linearly with respect to the number of conditional blocks of a source file, which allow us to generate the boolean formulas for all source files of the Linux kernel in **less than 30** minutes.

Our *analysis process* uses well-established formalisms—BDDs and SAT solvers—that allow us to provide different reasonings on the implementation variability. In order to assess the analysis process in real software we have conducted two case studies: In the first, using the Graph Product Line, we have analyzed and compared the feature model and the implementation variability. This has given us insights that both worlds show remarkably different characteristics and, therefore, should always be studied together. In the second case study, we have analyzed the entire code base of the Linux kernel for dead code blocks. This study has shown the feasibility of our approach to large SPLs and revealed interesting aspects of the analysis of CPP directives in real-word large-scale software projects.

We have explained the characteristics of the two different analysis strategies: source file and compilation unit modes. In order to perform a precise analysis (very) efficiently, compilation-unit mode should only be applied to compilation units that actually make use of conditional `#include` statements, which reduces the analysis time drastically. We have also presented highly effective further optimizations during the dead block analysis in Linux: When checking the satisfiability of a specific block, the whole model can be split into smaller models. With this, our *analysis process* took around 6 minutes for the actual search of dead blocks in all source files of

the Linux kernel. The study has revealed several inconsistencies in the code base, our fixes for two of them have in the meantime been accept in the Linux mainline tree.

Ideas for future work include (1) applying our approach to several other software projects (2) cross-checking the implementation variability model of the Linux kernel with the *Kbuild* model in order to reveal the *semantic* inconsistencies and after that (3) a detailed analysis and corresponding classification of configuration problems found in the Linux kernel.

References

[1] Greg J Badros and David Notkin. A framework for preprocessor-aware c source code analyses. *Software: Practice and Experience*, 30(8):907–924, 2000.

[2] Ira D. Baxter. DMS: program transformations for practical scalable software evolution. In *5th Int. W'shop on Principles of Software Evolution (IWPSE'02)*, pages 48–51, New York, NY, USA, 2002. ACM.

[3] Ira D. Baxter and Michael Mehlich. Preprocessor conditional removal by simple partial evaluation. In *8th Conf. on Reverse Engineering (WCRE '01)*, pages 281–, Washington, DC, USA, 2001. IEEE.

[4] D. Benavides, A. Ruiz-Cortés, and P. Trinidad. Automated reasoning on feature models. In *17th Int. Conf. on Advanced Information Systems Engineering (CAISE '05)*, volume 3520, pages 491–503, Heidelberg, Germany, 2005. Springer.

[5] BuDDy project. `http://sourceforge.net/projects/buddy`, 2009.

[6] Andreas Classen, Arnaud Hubaux, and Patrick Heymans. A formal semantics for multi-level staged configuration. In *3th Int. W'shop on Variability Modelling of Software-intensive Systems (VAMOS '09)*, pages 51–60, 2009.

[7] Krzysztof Czarnecki and Krzysztof Pietroszek. Verifying feature-based model templates against well-formedness OCL constraints. In *6th Int. Conf. on Generative Programming and Component Engineering (GPCE '06)*, pages 211–220, New York, NY, USA, 2006. ACM.

[8] Krzysztof Czarnecki and Andrzej Wasowski. Feature diagrams and logics: There and back again. In *11th Software Product Line Conf. (SPLC '07)*, pages 23–34. IEEE, Sept. 2007.

[9] Alejandra Garrido. *Program refactoring in the presence of preprocessor directives*. PhD thesis, University of Illinois at Urbana-Champaign, Champaign, IL, USA, 2005. Adviser-Johnson, Ralph.

[10] Ying Hu, Ettore Merlo, Michel Dagenais, and Bruno Lagüe. C/C++ conditional compilation analysis using symbolic execution. In *16th IEEE Int. Conf. on Software Maintainance (ICSM'00)*, page 196, Washington, DC, USA, 2000. IEEE.

[11] International Organization for Standardization. *ISO/IEC 9899:TC2: Programming languages — C*. International Organization for Standardization, Geneva, Switzerland, 2005.

[12] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial evaluation and automatic program generation*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1993.

[13] Christian Kästner, Sven Apel, and Martin Kuhlemann. Granularity in software product lines. In *30th Int. Conf. on Software Engineering (ICSE '08)*, pages 311–320, New York, NY, USA, 2008. ACM.

[14] Christian Kästner, Sven Apel, Salvador Trujillo, Martin Kuhlemann, and Don 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.

[15] Mario Latendresse. Rewrite systems for symbolic evaluation of c-like preprocessing. In *CSMR '04: Proceedings of the Eighth Euromicro Working Conference on Software Maintenance and Reengineering (CSMR'04)*, page 165, Washington, DC, USA, 2004. IEEE Computer Society.

[16] Jörg Liebig, Sven Apel, Christian Lengauer, Christian Kästner, and Michael Schulze. An analysis of the variability in forty preprocessor-based software product lines. In *32nd Int. Conf. on Software Engineering (ICSE '10)*, New York, NY, USA, 2010. ACM.

[17] The Limmat SAT solver. `http://fmv.jku.at/limmat/`, 2002.

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

[19] Roberto E. Lopez-Herrejon and Don Batory. A standard problem for evaluating product-line methodologies. In *3rd Int. Conf. on Generative and Component-Based Software Engineering (GCSE '01)*, volume 2186, pages 10–24, Heidelberg, Germany, 2001. Springer.

[20] Marcílio Mendonça, Andrzej Wasowski, Krzysztof Czarnecki, and Donald D. Cowan. Efficient compilation techniques for large scale feature models. In *5th Int. Conf. on Generative Programming and Component Engineering (GPCE '08)*, pages 13–22, New York, NY, USA, 2008. ACM.

[21] Andreas Metzger, Patrick Heymans, Klaus Pohl, Pierre-Yves Schobbens, and Germain Saval. Disambiguating the documentation of variability in software product lines. In *15th IEEE Int. Conf. on Requirements Engineering (RE'07)*, pages 243–253, Washington, DC, USA, 2007. IEEE Computer Society.

[22] Steven She, Rafael Lotufo, Thorsten Berger, Andrzej Wasowski, and Krzysztof Czarnecki. The variability model of the linux kernel. In *4th Int. W'shop on Variability Modelling of Software-intensive Systems (VAMOS '10)*, Linz, Austria, January 2010.

[23] Julio Sincero, Horst Schirmeier, Wolfgang Schröder-Preikschat, and Olaf Spinczyk. Is the linux kernel a software product line? In Frank van der Linden and Björn Lundell, editors, *International Workshop on Open Source Software and Product Lines (SPLC-OSSPL 2007)*, Kyoto, Japan, 2007.

[24] Julio Sincero and Wolfgang Schröder-Preikschat. The linux kernel configurator as a feature modeling tool. In Steffen Thiel and Klaus Pohl, editors, *12th Software Product Line Conf. (SPLC '08), Second Volume*, pages 257–260. Lero Int. Science Centre, University of Limerick, Ireland, 2008.

[25] Julio Sincero, Reinhard Tartler, and Daniel Lohmann. An algorithm for quantifying the program variability induced by conditional compilation. Technical Report CS-2010-02, University of Erlangen, Dept. of Computer Science, January 2010.

[26] Henry Spencer and Gehoff Collyer. #ifdef considered harmful, or portability experience with C News. In *1992 USENIX ATC*, Berkeley, CA, USA, June 1992. USENIX.

[27] Reinhard Tartler, Julio Sincero, Wolfgang Schröder-Preikschat, and Daniel Lohmann. Dead or alive: Finding zombie features in the Linux kernel. In *1st W'shop on Feature-Oriented Software Development (FOSD '09)*, pages 81–86. ACM, 2009.

[28] Sahil Thaker, Don Batory, David Kitchin, and William Cook. Safe composition of product lines. In *7th Int. Conf. on Generative Programming and Component Engineering (GPCE '07)*, pages 95–104, New York, NY, USA, 2007. ACM.

[29] Thomas Thum, Don Batory, and Christian Kästner. Reasoning about edits to feature models. In *31st Int. Conf. on Software Engineering (ICSE '09)*, pages 254–264, Washington, DC, USA, 2009. IEEE.