TECHNISCHE FAKULTÄT

Department Informatik

Technical Report CS-2010-02

Julio Sincero, Reinhard Tartler, Daniel Lohmann

An Algorithm for Quantifying the Program Variability Induced by Conditional Compilation

January 2010

Please cite as:
Julio Sincero, Reinhard Tartler, Daniel Lohmann, “An Algorithm for Quantifying the Program Variability Induced by Conditional Compilation,” University of Erlangen, Dept. of Computer Science, Technical Report CS-2010-02, January 2010.

Friedrich-Alexander-Universität Erlangen-Nürnberg

Friedrich-Alexander-Universität Erlangen-Nürnberg
Department Informatik

Martensstr. 3 · 91058 Erlangen · Germany

www.informatik.uni-erlangen.de

An Algorithm for Quantifying the Program Variability Induced by Conditional Compilation

Julio Sincero, Reinhard Tartler and Daniel Lohmann
{sincero,tartler,lohmann}@cs.fau.de
FAU Erlangen–Nuremberg

Abstract

The C Preprocessor (CPP) provides the programmer with powerful means to implement variability. Therefore the tool CPP is an important standard asset in the development toolchain for classical system software written in C. Yet, nobody has seriously quantified the effects of implementing variability with conditional compilation. In this report we describe conditional compilation in a semi-formal way. On this basis, we present an algorithm that is able to calculate the resulting variability. With this, we believe that further analysis on variability caused by preprocessor based configuration can be supported. This is useful to detect bugs caused by inconsistencies between variability models in the actual implementation and to support exploration of actually implemented variability in source artifacts.

1 Introduction

Complex software often requires re-implementation of parts and modules for several product variants. Very often these parts are not consolidated into a single module, but are scattered across the code base. In order to integrate these variants, programmers need techniques to enable and disable code snippets according to the required functionality that aredefined by a specific *configuration*. Both *annotative* and *compositional* approaches (e.g., [14]) are means for the realization of variability in the code. The annotative approach provided by the C preprocessor language (CPP) is by far the most employed approach.

Several papers emphasize the problems of preprocessor-based customization of software [21, 10]. Nevertheless, the C preprocessor has to be considered as state of the art for the implementation of *overhead-free*, *fine-grained* customization [9]. Application examples from the domain of operating systems include all variants and flavors of Linux and BSD as well as the majority of embedded operating systems, such as FreeRTOS [11], ProOSEK [18], Contiki [7], or eCos [8].

Nevertheless, the flexibility and simplicity of the C preprocessor comes at a price: It is not powerful enough to manage every detail of variability management. As a result, it is employed only where it is strong – conditional compilation. Therefore, large projects have to use different tools to manage other facets of variability management like variant specification, selection of compilation units for the build process, etc. This leads to a set of tools that are semantically connected (they control the variation points), however, as we have previously analyzed [22] for the Linux project, the consistency between these tools are currently poorly guaranteed. This work aims at closing this gap.

We believe that an algorithm which calculates the variability that results from conditional compilation by means of CPP directives will provide the appropriate *interface* between the C preprocessor and the other variability management tools so that the consistency between them can be better analyzed. In order to achieve this goal we first examine the *variability* that can be expressed with

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

conditional compilation. Then we present an algorithm to quantify the variability of source files that use `CPP` directives.

This report is structured as follows: First we give a brief overview over the subset of the `CPP` language we use for our approach. Then, we revisit the related work. In Section 4 we explain the variability algorithm in detail. Section 5 explains the plausibility of the algorithm with selected examples. Finally, we discuss the potential of our approach when integrated in software projects.

2 Conditional Compilation

The semantic rules that arise from the `CPP` language specification are pretty complex. Fortunately, for a quantitative analysis we do not need to consider all of them, but only the rules that define *conditional compilation*. Still, identifying and quantifying the effects of *conditional blocks* in an automated manner is not trivial.

Normally, the C Preprocessor is called while compiling software through an compiler driver. For the GNU Compiler Collection this driver is called `gcc`, the Preprocessor is called `cpp`. The Preprocessor is used to assemble the input for the actual compiler run. This input is called the *expanded compilation unit*.

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 analysis, we only consider the subset of the `CPP` language that include directives shown in Table 1. The directives `#if`, `#elif`, `#ifdef` and `#ifndef` are used to declare *conditional blocks*, that means blocks that are being 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*. Nested blocks are only considered if the block in which they are nested is included in the expanded compilation unit.

In this context the C Preprocessor is used to insert slices of source code into its output stream according to the defined `CPP` flags. Technically, these flags are defined at compilation time either as command-line parameter or by using `#define` directives. We assume that the user is able to control these `CPP` flags either manually of by using a configuration tool.

In this paper we discuss boolean configuration flags only. In practice, configuration flags can also be strings and integers that can be compared and checked at compile time.

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

```
1 #ifdef CONFIG_A
2 #endif
```

The number of variants that can be composed with this block is two: Either the item `CONFIG_A` is selected, then the block is selected, or if `CONFIG_A` is not set, then the block is skipped. The next listing shows an alternative block:

```
1 #ifdef CONFIG_A
2 #else
3 #endif
```

Here, we have again in total two variants that can be composed. Depending on the definition of `CONFIG_A`, either the first or the second block is selected. A use of this type of block can be seen in the following listing:

```
1 #ifdef CONFIG_A
2 #elif defined CONFIG_B
3 #endif
```

In this example, in total three variants can be composed: If both items are unset, both blocks are skipped. The second block is selected only if `CONFIG_A` is unset and `CONFIG_B` is set. If both flags are set, the first block gets precedence.

Considering the configuration flags as input boolean variables and the selection of the configuration blocks as output boolean variables, we can denote the following truth table.

CONFIG_A	CONFIG_B	Block 1	Block 2
0	0	0	0
0	1	0	1
1	0	1	0
1	1	1	0

This table makes it obvious that the number of permutations of configuration flag settings is commonly higher than the real number of variants that can be composed by the `CPP` tool. In order to quantify the effects of conditional compilation, this distinction must be carefully taken into account.

3 Related Work

The problem of **parsing C code in presence of conditional compilation** has been studied by several authors. Among of them, there are techniques to detect parsing problems due to the misuse of `CPP` directives. Lattendresse [16] proposes symbolic evaluation in order to find, for every line of code, the boolean expression under which it can be reached. He presents three *rewrite systems* to transform *conditional values* into boolean expressions, which are applicable at different phases of macro expansion and evaluation. Finding the boolean expression that describes the conditions under which each line of code can be compiled is very similar to our work. However, this work has different goals. Latendresse's work focuses on possible problems caused by macro expansion, whereas we concentrate on *conditional compilation*. Badros et. al. [1] propose a framework that integrates the C preprocessor and a parser. The framework permits analysis to be expressed in terms of both preprocessing and parsing actions. The framework permits the analysis of preprocessor constructs. Unlike our work, this analysis is used to support parsing.

Another series of work tackles the problem of **visualizing the variability described by `CPP` directives**. Krone et. al. [15] developed a tool that accepts source code with *configuration-specific statements*. Using this information an algorithm computes a concept lattice which allows visualization of the structure and properties of possible configurations. Analogous to our work is the idea of an abstract representation of all possible configurations of files. However, this abstract representation is used to evaluate the overall quality of the code according to software engineering principles. Kästner et. al. present the tool CIDE [13] that is able to annotate the *abstract syntax tree* of source files with information extracted from `CPP` directives. The tool provides a source code editor with an enhanced visualization by using background colors. Moreover, the tool is able to detect syntactic problems. Although the tool relies on the information obtained from the `CPP` directives, it tackles problems of *visualization* and *syntactic error detection*, whereas our approach focuses on the *quantification and analysis of variability*.

The **evaluation of variability from models** is also related to our work. Czarnecki et. al. [6] 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. The same kind of reasoning could be applied to the boolean formula generated by our approach in order to improve the analysis of the variability and to detect problems.

To the best of our knowledge, we are not aware of any work that uses the information from `CPP` directives in order to quantify variability. Moreover, in order to reduce the complexity of our approach we decided not address *syntactic error detection*, neither from the `CPP` language nor from the programming language using conditional compilation. Our approach simply requires that the files to be analyzed can be successfully parsed by the C preprocessor.

4 Calculating Variability

In this section we present an algorithm that calculates the variability of `CPP`-based compilation units. In order to make the understanding of the algorithm easier we provide some definitions in advance. The terms *configuration flag* and *conditional block* are used consistent to their descriptions in Section 2.

4.1 Basic definitions

Definition 1 (Configuration)**.** *Given n boolean flags, a **configuration** is the boolean vector $\vec{f} = f_1, \ldots, f_n$, where f_i is the value of the i-th configuration flag.*

When compiling a file with `CPP` directives, the user assigns configuration flags so that the blocks of interest are enabled for compilation. Each member of this vector represents the configuration flag assignment as given to `CPP`. As already mentioned in Section 2, in this report we consider boolean configuration flags only. The variables are set to 1 (*true*) if they are set, and 0 (*false)* otherwise.

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

Each bit in this vector represents a conditional block in the compilation unit u. When applying the tool CPP to a source file, the block selection represents which blocks CPP has selected (the bit is set) and which are skipped. Note that in practice, the same *configuration* is applied to different compilation units. With these two definitions, the process of applying a set of configuration flags $\vec{f}$ to a compilation unit u that contains m conditional blocks $\vec{b}$ can be expressed with the following function $\mathcal{P}$:

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

This function represents a *mapping* from a given *configuration* to a specific *block selection* when preprocessing is performed by the CPP tool. This function is well defined by the semantics of the CPP language.

4.2 Variability estimations

The function $\mathcal{P}$ is not *injective*, as in general there can be two different *configurations* that map to the same *block selection*. It is also not *surjective*, as in general there may be *block selections* that cannot be composed by any *configuration* because of the semantics of the CPP language.

The total number of different *configurations* a source file can be compiled with is (obviously) 2^n. This number is an upper bound of *block selections* for a given compilation unit. Another upper bound for a compilation unit u with m conditional blocks is 2^m. However, due to the semantics imposed by the CPP language, it is very common that different *configurations* map to the same *block selection*. Consider the conditional block in Listing 1. We note three different configuration flags and a single conditional block. The upper bound for the number of possible configuration is in this example $2^{3+1} = 16$. However, the real number of configurations is obviously 2: Either all three items are defined – then the block is selected – or the block is skipped. This means that 7 out of $2^3 = 8$ flag combinations lead to exactly the same post-processed file.

```
1 #ifdef (CONFIG_A && CONFIG_B && CONFIG_C)
2 #endif
```

Listing 1: Conditional Block with four items

As we have already seen in Section 2, simply using the upper bound 2^n as number of variants is not appropriate for a serious quantification of variability. Instead, an algorithm for quantifying variability must properly consider all details of the CPP language that are relevant for conditional compilation:

- **Compound expressions** like in Listing 1.
- **Nested blocks**: Only if the outer expression evaluates to *true*, the inner expression are further evaluated.
- **`#if`–`#elif`–`#else` conditions**: only if the first expression in the `#if` group evaluates to *false*, the following `#elif` expression are evaluated. If the leading `#if` expression evaluates to *true*, the other blocks are not selected.

4.3 The checker function

The algorithm described in this section builds a helper function that *checks* if a configuration of conditional blocks $\vec{b}$ is selected with a given input configuration $\vec{f}$.

Definition 3 (Checker Function)**.** *Given a function $\mathcal{P}$ 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*** *$\mathcal{C}$ is defined as:*

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

Algorithm 1 Constructing the **checker function** $\mathcal{C}$

1: $C := true$
2: **for** $i := 1$ to m **do**
3: $\quad C := C \wedge \overbrace{(b_i \rightarrow parent(b_i))}^{\blacktriangleright 1}$
4: $\quad C := C \wedge \overbrace{(b_i \rightarrow exp(b_i))}^{\blacktriangleright 2}$
5: $\quad C := C \wedge \overbrace{(b_i \rightarrow \neg succ(b_i))}^{\blacktriangleright 3}$
6: $\quad C := C \wedge \overbrace{((parent(b_i) \wedge exp(b_i) \wedge \neg succ(b_i)) \rightarrow b_i)}^{\blacktriangleright 4}$
7: **end for**

Algorithm 1 builds the *checker function* $\mathcal{C}$. In Step 1 the boolean variable C is initialized to *true*. The algorithm iterates over all *block boolean variables* B. In each iteration, different implications are conjugated to C. These implications result directly from the semantics of the `CPP` language and are described in the following.

A block always implies its parent when it is not a top-level block, this clause in shown on Step 3 under the symbol $\blacktriangleright 1$. This rule ensures that a block can only be selected if its parent is also selected as well. If the block is nested, the function $parent(b)$ returns *true* if and only if the block in which the current block is nested is selected as well. If no such block exists or the block is not selected, the function returns *false*.

The rule in Step 4 reflects that the selection of a block also always implies its expression (clause shown on Step 4 under the symbol $\blacktriangleright 2$). This means that if a block is selected, its expression is implied. The function $exp(b)$ returns the *expression boolean variable* for this block, which may be a complex expression or a simple flag. In the case of `#else` blocks, the negation of the expression of its predecessor block (which may be a `#if`, `#if` or `#elif` block) is returned.

`#ifdef` clouds require special treatment. With this term we mean groups of directives containing at least one `#elif` or `#else` directives, multiple occurrences of `#elif` directives are allowed and at most one `#else` is permitted. In such groups at most one of them will be selected at a time. When an `#else` directive is present, there is always exactly one block of the cloud that is present, even if the expression of multiple directives of the same cloud evaluate to *true*. In this case, the block that is selected is determined by the order in which they are declared, that is, the first block with an expression that evaluates to *true*. Subsequent blocks will always be skipped, independent of their expressions. In the algorithm, Step 5 generates the necessary constraints. That clause uses the function $succ(b)$ (under the brace $\blacktriangleright 3$ in Figure 1), which ensures that none of its predecessors in the same `#ifdef` cloud is selected.

The last clause generated for each block is shown in Step 6 and marked with the symbol $\blacktriangleright 4$. It ensures that when a block has all its conditions fulfilled (by re-using the functions $\blacktriangleright 1, \blacktriangleright 2$ and $\blacktriangleright 3$) the block itself is selected.

4.4 Calculating possible block selections

With the *checker function* we define the term *variability* as follows:

Definition 4 (Variability)**.** *Given a compilation unit u and the checker function $\mathcal{C}$, the **variability** $\mathcal{V}$ of a compilation unit u is the set of all block selections $\vec{b}$ for that there exists a selection $\vec{f}$, so that $\mathcal{C}$ is satisfied:*

$$\mathcal{V} = \{\vec{b} \mid \exists \vec{f} : \mathcal{C}(\vec{f}, \vec{b_u})\} \tag{3}$$

Algorithm 2 Algorithm for calculating block selections

1: **for all** conditional blocks **do**
2: introduce first-order logic variable b_i
3: **end for**
4: **for all** `#ifdef` expressions e **do**
5: introduce first-order logic variable e_i
6: **for all** configuration items inside e_i **do**
7: introduce first-order logic variable f_i
8: **end for**
9: **end for**
10: construct **checker function** $\mathcal{C}$
11: calculate $\vec{f}$ and $\vec{b_u}$ that satisfy $\mathcal{C}$
12: **return** all $\vec{b_u}$ of the calculated set

The *variability* $\mathcal{V}$ is therefore the set of vectors $\vec{b_u}$ that can be mapped by the function $\mathcal{P}(\vec{f}, u)$ with at least one *input configuration* $\vec{f}$. Algorithm 2 shows how to calculate $\mathcal{V}$. First, for each block a boolean variable is declared (Step 1). These *block variables* b_i represent the vectors $\vec{b_u}$ of equation (2) and will be used for constructing the *checker function* $\mathcal{C}$ in Step 10. Then, for each configuration item inside an `#ifdef` expression a *configuration flag variable* $\vec{f_i}$ is declared (Step 7) that represents the setting for that a configuration flag. Finally, for each expression a boolean variable e_i is declared for the corresponding `#ifdef` directive, which is also used for constructing the *checker function*. These variables reference the *configuration flag variables* and essentially represent the evaluation of the respective logical expression.

After all these helper variables are declared, the *checker function* $\mathcal{P}$ (Step 10) is created and inputs that result *true* are calculated, which can be done with standard SAT solvers or BDD packages. Sets with the same vector $\vec{b}$ are considered equivalent, as they represent the same *block selection*. $\mathcal{V}$ is then the set of unique vectors $\vec{b}$.

5 Examples

In order to explain how the algorithm from Section 4 works in practice, we show the results of the *checker function* for selected trivial examples. In order to make the understanding easier, we use the following annotations:

- The symbol ↻ and the end of each line specifies in which iteration of the algorithm the clause was generated.
- Each clause has been marked with an overbrace ($\overbrace{\ \ }^{\blacktriangleright n}$) in order to indicate which step of the algorithm generated the clause. This clarifies which of the four conditions of the respective block is enforced.

5.1 Simple Conditional

```
1 #ifdef CONFIG_A
2 #endif
```

Listing 2: Trivial Example

With the file shown on Listing 2 is given as input to our algorithm the following *checker function* is generated with our algorithm:

$$\mathcal{C} = \Big(\overbrace{(b_1 \rightarrow A)}^{\blacktriangleright 2} \wedge \overbrace{(A \rightarrow b_1)}^{\blacktriangleright 4}\Big) \qquad (\circlearrowleft\ 1)$$

As the file has only one conditional block, the algorithm builds the formula in a single iteration. The first clause ($\blacktriangleright 2$) represents the dependency of the block and the flag (A) used in the `#ifdef` directive. Due to the fact that the sole block is a top-level block (no nesting) and it does not form a *if-cloud*, clauses generated by the conditions $\blacktriangleright 1$ and $\blacktriangleright 3$ do not appear in the resulting formula. As a result, the last clause $\blacktriangleright 4$ is simply the implication of the flag A and the block b_1. The formula enforces that the block will necessarily be selected if and only if the flag it depends on is also selected. This can be seen when we rewrite the formula in an equivalent and simpler form, resulting in $X = b_1 \leftrightarrow A$. For this example the following inputs[1] satisfy the *checker function*:

$$\Big\{(\emptyset, \emptyset), (\{A\}, \{b_1\})\Big\}$$

From these tuples the algorithm calculates as variability the following vectors $\vec{b_u}$: $(0), (1)$. For the following examples, we use the following abbreviated form:

$$\mathcal{V} = \Big\{\emptyset, \{b_1\}\Big\}$$

The vector $\vec{b_u}$ is noted as sets of selected blocks. The set has only two members, the empty set, which means that when no configuration flags are set then the block is skipped. The other configuration show that by enabling the flag A, the Block b_1 will be selected as well.

5.2 Expressions

```
1 #if defined(CONFIG_A) && defined(CONFIG_B)
2 #endif
```

Listing 3: Example of a `CPP` directive with a logic expression

[1] NB: We note the inputs as tuples with the vectors $\vec{f}$ and $\vec{b}$. In order to improve readability we describe the vector in form of a set that omits blocks and flags that are not set in the vector; the sets contain only the name of blocks and flags that are selected or set. Moreover, we denote the null vector $\vec{0}$ as the empty set $\emptyset$. This notation is be used in the remainder of this document.

The example shown in Listing 3 is very similar to the previous one (Listing 2) with the difference that the block does not depend on a single flag, but on a combination of flags that form a logic expression. With this listing as input, the algorithm produces the following *checker function*:

$$\mathcal{C} = \Big(\overbrace{(b_1 \rightarrow (A \wedge B))}^{\blacktriangleright 2} \wedge \overbrace{((A \wedge B) \rightarrow b_1)}^{\blacktriangleright 4}\Big) \qquad (\circlearrowleft\ 1)$$

As the input file has only one block, this function is again built in one iteration. Clauses $\blacktriangleright 1$ and $\blacktriangleright 3$ are not applied because the block is top level and there are no *if-clouds*. The clause $\blacktriangleright 2$ is built using the expression $(A \wedge B)$ present on the `#if` directive. The same for the clause $\blacktriangleright 4$, which forms the bi-implication. Therefore, the block b_1 will be selected only if the expression evaluates to true, and in this case (no nesting, and no clouds), the block will always be selected if the expression evaluates to true. The solution for this *checker function* is:

$$\Big\{(\emptyset, \emptyset), (\{A\}, \emptyset), (\{B\}, \emptyset), (\{A, B\}, \{b_1\})\Big\}$$

The first tuple represents the trivial solution. The next two sets do not contain the block b_1 because the expression does evaluate to true if and only if both flags are selected. This solution is represented by the last set and therefore the algorithm produces as variability $\mathcal{V}$:

$$\mathcal{V} = \Big\{\emptyset, \{b_1\}\Big\} \qquad (4)$$

5.3 Parent Relationship

```
1 #ifdef CONFIG_A
2 #ifdef CONFIG_B
3 #ifdef CONFIG_C
4 #endif
5 #endif
6 #endif
```

Listing 4: Example with nested `CPP` directives

Listing 4 consists of three blocks – one is top level and the other two are nested. With this listing as input to the variability algorithm, the boolean function in Figure 1 is generated.

The algorithm iterates over three blocks. In the first iteration the conditions for the top level block (b_1) are generated, which are, the expression clause ($\blacktriangleright 2$) and the clause $\blacktriangleright 4$ to form the implication of the block's conditions. The second iteration begins by generating the parent condition of the block b_2, which is nested on block b_1, and therefore, b_2 can only be selected if b_1 is also selected. This is ensured by the clause $\blacktriangleright 1$. Clause $\blacktriangleright 2$ ensures that the second block also depends on its expression. Clause $\blacktriangleright 4$ is the implication of the block's conditions (expression and parent) and the block itself. This guarantees that when all of its conditions are met, the block will necessarily be selected.

The third block (b_3) has the same conditions to be selected as the second, namely, expression ($\blacktriangleright 2$) and parent ($\blacktriangleright 1$). However, it depends on a different flag (C), and it has a different parent

$$\mathcal{C} = \Big(\overbrace{(b_1 \to A)}^{\blacktriangleright 2} \wedge \overbrace{(A \to b_1)}^{\blacktriangleright 4}\Big) \quad (\circlearrowleft 1)$$

$$\wedge \quad \Big(\overbrace{(b_2 \to b_1)}^{\blacktriangleright 1} \wedge \overbrace{(b_2 \to B)}^{\blacktriangleright 2} \quad (\circlearrowleft 2)$$

$$\wedge \quad \overbrace{((A \wedge b_1) \to b_2)}^{\blacktriangleright 4}\Big) \quad (\circlearrowleft 2)$$

$$\wedge \quad \Big(\overbrace{(b_3 \to C)}^{\blacktriangleright 2} \wedge \overbrace{(b_3 \to b_1)}^{\blacktriangleright 1} \quad (\circlearrowleft 3)$$

$$\wedge \quad \overbrace{((B \wedge b_2) \to b_3)}^{\blacktriangleright 4}\Big) \quad (\circlearrowleft 3)$$

Figure 1: Checker function for Listing 4

(b_2). Note that its parent is a nested block as well, that is, the third block can only be selected if the first block is selected.

As the property *nested in* is transitive, it is not necessary to apply the implication $\blacktriangleright 1$ to all parents of a given block; the direct parent is sufficient. This means that in this example the condition $b_3 \to b_1$, (i.e., Block 3 cannot be selected without Block 1) is implicitly assured. Therefore, the nesting depth of each block does not matter for the algorithm.

The inputs that satisfy the *checker function* for this listing is:

$$\Big\{(\emptyset, \emptyset), (\{B\}, \emptyset), (\{C\}, \emptyset), (\{B, C\}, \emptyset), (\{A\}, \{b_1\}), (\{A, C\}, \{b_1\}),$$
$$(\{A, B\}, \{b_1, b_2\}), (\{A, B, C\}, \{b_1, b_2, b_3\})\Big\}$$

Again, the trivial solution is found first. The next three sets are the configuration of variable flags that do not select any blocks. In the fifth set shows that if the flag `CONFIG_A` is set alone, then the block b_1 is selected. Adding the flag `CONFIG_C` (sixth set) does not cause additional blocks to be selected. Selecting flag `CONFIG_A` and `CONFIG_B` (seventh set) selects blocks b_1 and b_2. The last set shows the solution when all three flags are selected: all three blocks are selected as well. From these solutions, we identify as *variability*:

$$\mathcal{V} = \Big\{\emptyset, \{b_1\}, \{b_1, b_2\}, \{b_1, b_2, b_3\}\Big\}$$

5.4 IF Groups

```
1 #ifdef CONFIG_A
2 #elif CONFIG_B
3 #else
4 #endif
```

Listing 5: Example with `CPP` directives that belong to an `#if` Group

$$\mathcal{C} = \Big(\overbrace{(b_1 \rightarrow A)}^{\blacktriangleright 2} \wedge \overbrace{(b_1 \rightarrow \neg(b_2 \vee b_3))}^{\blacktriangleright 3} \qquad (\circlearrowleft 1)$$

$$\wedge \quad \overbrace{((A \wedge \neg(b_2 \vee b_3)) \rightarrow b_1)}^{\blacktriangleright 4}\Big) \qquad (\circlearrowleft 1)$$

$$\wedge \quad \Big(\overbrace{(b_2 \rightarrow B)}^{\blacktriangleright 2} \wedge \overbrace{(b_2 \rightarrow \neg(b_3))}^{\blacktriangleright 3} \qquad (\circlearrowleft 2)$$

$$\wedge \quad \overbrace{((B \wedge \neg(b_3) \rightarrow b_2)}^{\blacktriangleright 4}\Big) \qquad (\circlearrowleft 2)$$

$$\wedge \quad \Big(\overbrace{(b_3 \rightarrow \neg B)}^{\blacktriangleright 3} \wedge \overbrace{(\neg B \rightarrow b_3)}^{\blacktriangleright 4}\Big) \qquad (\circlearrowleft 3)$$

Figure 2: Checker function for Listing 5

Listing 5 shows a file containing three top-level blocks all belonging to a single *if-cloud*, when it is given as input to the variability algorithm, the *checker function* shown in Figure 2 is generated.

As the listing contains the *conditional blocks*, the *checker functions* is constructed in three iterations. In the first iteration, the expression condition ($\blacktriangleright 2$) for the block b_1 is generated, and also the *if-cloud* condition ($\blacktriangleright 3$). The latter assures that this block implies the negation of the disjunction of all of its *successors* (b_2, b_3 and b_4) in the *if-cloud*. This means that when this block is selected, none of its successors is selected. Again, condition $\blacktriangleright 4$ enforces that b_1 will necessarily be true when all of its conditions are met.

The second iteration is very similar to the first one, except by the fact that the block b_2 has a different expression (shown on clause $\blacktriangleright 2$) and it has one successor less then b_1 (shown on clause $\blacktriangleright 3$). Naturally, these differences are reflected on clause $\blacktriangleright 4$.

The last iteration generates the conditions for the block b_3 representing the `#else` directive. For the generation of the expression condition ($\blacktriangleright 2$), the function $exp(b_3)$ returns the negation of the *cloud leader* (the first block of the *if-cloud*), this function could also return the negation of the disjunction of all precedent cloud members' expressions. Because clause $\blacktriangleright 3$ is generated for all of its predecessors in previous iterations, the negation of the disjunctions is not necessary; the negation of the *cloud leader's* expression is sufficient. The clause $\blacktriangleright 4$ for b_3 is the implication of the expression returned by $exp(b_3)$ and the block itself, as it is neither nested nor it has any successors.

The following solutions satisfy the calculated *checker function*:

$$\Big\{(\emptyset, \{b_3\}), (\{A\}, \{b_1\}), (\{B\}, \{b_2\}), (\{A, B\}, \{b_1\})\Big\}$$

The first set is again the solution with no `CPP` flags set. In this case however, it is not empty but contains the block b_3. When looking at Listing 5, we see that when neither the flags `CONFIG_A` nor `CONFIG_B` is set, the `#else` block in line 3 is selected.

The next two sets represent the solutions that the flag `CONFIG_A` causes block b_1 (Line 1) and that `CONFIG_B` block b_2 to be selected. The last shows the situation when both flags have been selected. In this case, block b_1 wins. This is again in accordance with the `CPP` language specification.

From these solutions we identify as *variability*:

$$\mathcal{V} = \Big\{\{b_3\}, \{b_1\}, \{b_2\}\Big\}$$

5.5 Combined Example

```
1  #if defined CONFIG_A || defined  CONFIG_B || defined (CONFIG_C)
2  # if defined ( CONFIG_A && CONFIG_B )
3  # elif defined ( CONFIG_A && CONFIG_C )
4  # elif defined ( CONFIG_B && CONFIG_C )
5  #   ifdef CONFIG_C
6  #   else
7  #   endif
8  # else
9  # endif
10 #endif
```

Listing 6: Combined example

The example shown on Listing 6 shows a combination of all semantic elements presented in the previous examples. The complexity of the example is typical for real world code.

From the source code in Listing 6, the *checker function* shown in Figure 3 is constructed. The listing consists of the top-level block b_1 (line 1) which ranges over all of the listing. Nested inside that block, there is an *#if-cloud* with four members. The first block (`#if`) is the *block-leader*, followed by two `#elif`'s and one `#else` directives.

The fist iteration generates the clauses required by the top level block (b_1). Clauses $\blacktriangleright 2$ and $\blacktriangleright 4$ assure that it depends only on its expression.

For blocks b_2 (line 2), b_3 (line 3) and b_4 (line 4) the clauses $\blacktriangleright 1$, $\blacktriangleright 2$, $\blacktriangleright 3$ and $\blacktriangleright 4$ are generated with the corresponding expressions and successors and with the same parent. This can be seen on iterations 2, 3 and 4 in the above formula.

Blocks b_5 (line 5) and b_6 (line 6) form a cloud nested in block b_4. Therefore, iteration 5 and 6 generate the clauses $\blacktriangleright 1$, $\blacktriangleright 2$, $\blacktriangleright 4$ for both blocks, naturally, with the corresponding expressions and using the same parent (b_4). The clause $\blacktriangleright 3$ is generated only for block b_5 as the block b_6 represents the `#else` directive that has no successors.

Block b_7 (line 6) is treated in the last iteration. Only the clauses $\blacktriangleright 1$, $\blacktriangleright 2$ and $\blacktriangleright 4$ are generated.

The algorithm identifies the following solutions that satisfy this *checker function*:

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

From these solutions, we identify as *variability*:

$$\mathcal{V} = \Big\{\{b_1,b_7\},\{b_1,b_4,b_5\},\{b_1,b_7\},\{b_1,b_3,\}\{b_1,b_2\}\Big\} \tag{5}$$

6 Discussion

The algorithm presented in this report can be used for many applications. An important use is the **analysis and evaluation** of the variability regarding the *consistency* of the code base. Examples for consistency violations are *dead code*, that is, conditional blocks that cannot be selected, and *zombie code*, that means blocks that cannot be disabled by any *configuration* [22]. An example of *dead*

$$\mathcal{C} = \Big(\overbrace{(b_1 \to (A \vee B \vee C))}^{\blacktriangleright 2} \wedge \overbrace{((A \vee B \vee C) \to b_1)}^{\blacktriangleright 4} \Big) \quad (\circlearrowleft 1)$$

$$\wedge \quad \Big(\overbrace{(b_2 \to b_1)}^{\blacktriangleright 1} \wedge \overbrace{(b_2 \to (A \wedge B))}^{\blacktriangleright 2} \quad (\circlearrowleft 2)$$

$$\wedge \quad \overbrace{(b_2 \to \neg(b_3 \vee b_4 \vee b_7))}^{\blacktriangleright 3} \quad (\circlearrowleft 2)$$

$$\wedge \quad \overbrace{(b_1 \wedge (A \wedge B) \wedge \neg(b_3 \vee b_4 \vee b_7)) \to b_2)}^{\blacktriangleright 4} \Big) \quad (\circlearrowleft 2)$$

$$\wedge \quad \Big(\overbrace{(b_3 \to b_1)}^{\blacktriangleright 1} \wedge \overbrace{(b_3 \to (A \wedge C))}^{\blacktriangleright 2} \quad (\circlearrowleft 3)$$

$$\wedge \quad \overbrace{(b_3 \to \neg(b_4 \vee b_7))}^{\blacktriangleright 3} \quad (\circlearrowleft 3)$$

$$\wedge \quad \overbrace{(b_1 \wedge (A \wedge C) \wedge \neg(b_4 \vee b_7)) \to b_3)}^{\blacktriangleright 4} \Big) \quad (\circlearrowleft 3)$$

$$\wedge \quad \Big(\overbrace{(b_4 \to b_1)}^{\blacktriangleright 1} \wedge \overbrace{(b_4 \to (B \wedge C))}^{\blacktriangleright 2} \quad (\circlearrowleft 4)$$

$$\wedge \quad \overbrace{(b_4 \to \neg(b_7))}^{\blacktriangleright 3} \quad (\circlearrowleft 4)$$

$$\wedge \quad \overbrace{(b_1 \wedge (B \wedge C) \wedge \neg(b_7)) \to b_4)}^{\blacktriangleright 4} \Big) \quad (\circlearrowleft 4)$$

$$\wedge \quad \Big(\overbrace{(b_5 \to b_4)}^{\blacktriangleright 1} \wedge \overbrace{(b_5 \to C)}^{\blacktriangleright 2} \quad (\circlearrowleft 5)$$

$$\wedge \quad \overbrace{(b_5 \to \neg(b_6))}^{\blacktriangleright 3} \quad (\circlearrowleft 5)$$

$$\wedge \quad \overbrace{(b_4 \wedge C \wedge \neg(b_6)) \to b_5)}^{\blacktriangleright 4} \Big) \quad (\circlearrowleft 5)$$

$$\wedge \quad \Big(\overbrace{(b_6 \to b_4)}^{\blacktriangleright 1} \wedge \overbrace{(b_6 \to \neg(C))}^{\blacktriangleright 2} \quad (\circlearrowleft 6)$$

$$\wedge \quad \overbrace{(b_4 \wedge \neg(C)) \to b_6)}^{\blacktriangleright 4} \Big) \quad (\circlearrowleft 6)$$

$$\wedge \quad \Big(\overbrace{(b_7 \to b_1)}^{\blacktriangleright 1} \wedge \overbrace{(b_7 \to \neg(A \vee B \vee C))}^{\blacktriangleright 2} \quad (\circlearrowleft 7)$$

$$\wedge \quad \overbrace{(b_1 \wedge \neg(A \vee B \vee C)) \to b_7)}^{\blacktriangleright 4} \Big) \quad (\circlearrowleft 7)$$

Figure 3: Checker function for Listing 6

code is shown in Listing 6 on page 14, where the block starting in Line 6 is dead. If we look at the variability for the example shown on Equation 5, the Block b_6 does not appear in any member of the variability set. This means that this block will never be selected by the C preprocessor, and therefore, can be considered as *dead code*. Similarly, *zombie code* is represented by code blocks that appear on all members of a variability set, and, therefore, will never be skipped by the C preprocessor.

For large code bases, our algorithm can be used for **exploration of variability**. Especially for new developers in a software project, the implementation and effects of variability often is difficult to assess and requires a lot of time to be fully understood. But also existing developers and reviewers could be assisted with a tool that calculates the variability as described in this report in order to verify refactorings of variability implementations. Last, we imagine that on the basis of the calculated variability, test cases can be written to ensure that the expected variability is achieved.

Another area for the application of our approach is checking the **consistency between the code base and abstract variability models**. Normally, software projects with a large amount of variability often maintain an *abstract variability model* in order to ease the configuration process. Such models are responsible for describing the variability in the project and are basically a compact form representing all possible variants of the project. In the Software Product Line community *feature models*[5] are used. For the Linux kernel[17], *kconfig* [20, 19] is used as the abstract model that allows the user to tailor a specific kernel configuration according to his needs. We assume that in many cases the variability of the abstract model in such projects does not follow the variability of the respective *implementation assets* and during development, diverges over time. Here, our approach can help to detect inconsistencies by calculating the variability of both the implementation and the models independently, and comparing the results afterwards. Furthermore, several kinds of comparisons between the actual implementation and abstract models can be performed, like comparing the number of allowed configurations in the code base with the abstract model variability. This allows, for example, insight into the *coverage* of the code base with respect to the set of variants that the user can configure.

There are several approaches to analyze the variability described by such abstract variability models. However, to the best of our knowledge, we are not aware of any work that checks the consistency between the code base variability and the abstract models. Several researchers [6, 12, 2, 3] have studied the transformation of feature models into boolean formulas. We aim at combining such formulas with our algorithm in order to check consistency. By means of the conjunction between the boolean formulas describing the code base and the abstract model, we can find inconsistencies such as dead and zombie code that arise from the divergence of the family model and the variability implemented in the source artifacts.

The algorithm described in this work has been developed as part of the VAMOS project. More information about the developed tools inclunding screenshots and demos can be found on the project website: `http://www4.informatik.uni-erlangen.de/Research/VAMOS/demo.shtml`.

Threats of Validity and Future Work

The algorithm we have presented in this report 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. In practice, for evaluating `CPP` expressions with nonboolean configuration flags like strings and integers, they are always used with a comparator function (e.g., ==, <, etc.). In order to support these cases as well, the algorithm can be extended to create a new boolean variable for each term that needs to be evaluated at compilation time. Further (static) knowledge about the assignment of these flags can be used to add additional conjunctions for the *checker function* in order to improve the variability calculations.

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 again adding additional conjunctions the *checker function*, more specifically the *exp(b_i)* function. Here, for each expression the function would need to take undefinitions and redefinitions of the undefined or redfined `CPP` flags into account.

While both cases are rather straight forward to implement, we believe that even the current implementation of the algorithm is well suited for most of the use cases we outlined above.

7 Summary

This report discusses the effects of conditional compilation regarding the resulting variability in implementation assets that use the C Preprocessor language `CPP`. In essence, the tool `CPP` selects available conditional blocks in an implementation unit according to a given input *configuration*. In order to quantify implementation variants that can arise from different configurations, we present an algorithm that calculates and properly quantifies the *implementation variability* of any source text that uses the `CPP` language. This algorithm can be seen as building block for various future tools. First, bugs like dead code caused by inconsistencies can be found systematically. Second, tools that assist developers and reviewers with exploring mechanisms that implement variability could make great use of the calculated variability especially in code bases with a large amount of variability. Last, in model driven software projects and Software Product Line engineering, crosschecking the variability model with the implementation variability of the source code allows to verify the intended amount of variability.

References

[1] Greg J. Badros and David Notkin. A framework for preprocessor-aware c source code analyses. *Softw. Pract. Exper.*, 30(8):907–924, 2000.
(Cited on page 5)

[2] Don S. Batory. Feature models, grammars, and propositional formulas. In *Proceedings of the 9th Software Product Line Conference (SPLC '05)*, pages 7–20, 2005.
(Cited on page 16)

[3] D. Benavides, A. Ruiz-Cortés, and P. Trinidad. Automated reasoning on feature models. *LNCS, Advanced Information Systems Engineering: 17th International Conference, CAiSE 2005*, 3520:491–503, 2005.
(Cited on page 16)

[4] D. Benavides, S. Segura, P. Trinidad, and A. Ruiz-Corts. FAMA: Tooling a framework for the automated analysis of feature models. In *Proceeding of the First International Workshop on Variability Modelling of Software-intensive Systems (VAMOS)*, 2007.
(Cited on page 6)

[5] Krysztof Czarnecki and Ulrich W. Eisenecker. *Generative Programming. Methods, Tools and Applications*. Addison-Wesley, May 2000.
(Cited on page 16)

[6] Krzysztof Czarnecki and Andrzej Wasowski. Feature diagrams and logics: There and back again. In *Proceedings of the 11th Software Product Line Conference (SPLC '07)*, pages 23–34, 2007.
(Cited on pages 6 and 16)

[7] Adam Dunkels, Björn Grönvall, and Thiemo Voigt. Contiki — a lightweight and flexible operating system for tiny networked sensors. In *Proceedings of the First IEEE Workshop on Embedded Networked Sensors (Emnets-I)*, Tampa, Florida, USA, November 2004.
(Cited on page 3)

[8] eCos homepage. http://ecos.sourceware.org/.
(Cited on page 3)

[9] Michael D. Ernst, Greg J. Badros, and David Notkin. An empirical analysis of C preprocessor use. *IEEE Transactions on Software Engineering*, 28(12):1146–1170, 2002.
(Cited on page 3)

[10] Jean-Marie Favre. A rigorous approach to support the maintenance of large portable software. In *1st Euromicro Working Conference on Software Maintenance and Reengineering (CSMR '97)*, page 44. IEEE Computer Society Press, 1997.
(Cited on page 3)

[11] FreeRTOS homepage. http://freertos.org/.
(Cited on page 3)

[12] Mikolás Janota and Joseph Kiniry. Reasoning about feature models in higher-order logic. In *Proceedings of the 11th Software Product Line Conference (SPLC '07)*, pages 13–22, 2007.
(Cited on page 16)

[13] Christian Kästner, Sven Apel, and Martin Kuhlemann. Granularity in software product lines. In *ICSE*, pages 311–320, 2008.
(Cited on page 6)

[14] Christian Kästner, Sven Apel, Syed Saif ur Rahman, Marko Rosenmüller, Don Batory, , and Gunter Saake. On the impact of the optional feature problem: Analysis and case studies. In *Proceedings of the 13th Software Product Line Conference (SPLC '09)*, Washington, DC, USA, 2009. IEEE Computer Society Press.
(Cited on page 3)

[15] Maren Krone and Gregor Snelting. On the inference of configuration structures from source code. In *ICSE '94: Proceedings of the 16th international conference on Software engineering*, pages 49–57, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.
(Cited on page 6)

[16] 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.
(Cited on page 5)

[17] Robert Love. *Linux Kernel Development*. Novell Press, 2nd edition, 2005.
(Cited on page 16)

[18] ProOSEK homepage. http://www.proosek.de/.
(Cited on page 3)

[19] 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, *Proceedings of the International Workshop on Open Source Software and Product Lines (SPLC-OSSPL 2007)*, Kyoto, Japan, 2007.
(Cited on page 16)

[20] Julio Sincero and Wolfgang Schröder-Preikschat. The linux kernel configurator as a feature modeling tool. In Steffen Thiel and Klaus Pohl, editors, *SPLC (2)*, pages 257–260. Lero Int. Science Centre, University of Limerick, Ireland, 2008.
(Cited on page 16)

[21] Henry Spencer and Gehoff Collyer. #ifdef considered harmful, or portability experience with C News. In *Proceedings of the 1992 USENIX Technical Conference*. USENIX Association, June 1992.
(Cited on page 3)

[22] Reinhard Tartler, Julio Sincero, Wolfgang Schröder-Preikschat, and Daniel Lohmann. Dead or alive: finding zombie features in the linux kernel. In *Proceedings of the 1st Workshop on Feature-Oriented Software Development (FOSD '09)*, pages 81–86, 2009.
(Cited on pages 3 and 14)