Journal of Universal Computer Science, vol. 20, no. 5 (2014), 746-771
submitted: 18/7/13, accepted: 15/2/13, appeared: 1/5/14 © J.UCS

A Catalogue of Refactorings to Remove Incomplete Annotations

Flávio Medeiros
(Federal University of Campina Grande, Campina Grande, Brazil
flaviomedeiros@copin.ufcg.edu.br)

Márcio Ribeiro
(Federal University of Alagoas, Maceió, Brazil
marcio@ic.ufal.br)

Rohit Gheyi
(Federal University of Campina Grande, Campina Grande, Brazil
rohit@dsc.ufcg.edu.br)

Baldoino Fonseca
(Federal University of Alagoas, Maceió, Brazil
baldoino@ic.ufal.br)

Abstract: Developers use the C Preprocessor (CPP) to handle portability and variability in program families of different sizes and domains. However, despite the widely use of the CPP in practice, it is often criticised due to its negative impact on code quality and maintainability, tool development, and its error-prone characteristics. In particular, developers aggravate these problems when using incomplete annotations, i.e., directives encompassing only parts of syntactical units. In a previous work, we performed an empirical study on 41 C program family releases and found that almost 90% of syntax errors occur in incomplete annotations. There are some refactorings to remove incomplete annotations proposed in the literature. However, they clone code and increase Lines of Code (LOC). To avoid incomplete annotations and their intrinsic problems, in this article we propose a catalogue of refactorings that converts incomplete annotations into complete ones without cloning code. We implement an *Eclipse plug-in* to help developers applying our refactorings automatically. To evaluate our catalogue, we performed a study to analyse questions related to code cloning, LOC, and number of directives. To answer our research questions, we analyse releases of 12 C program families of different domains ranging from 4.9 thousand to 1.5 million LOC. The results show that our catalogue can remove all incomplete annotations without cloning code, and increasing only in 0.04% the LOC and in 2.10% the number of directives.
Key Words: Refactoring, C Language, Preprocessors, Program Families
Category: D.2.3, D.2.7, D.3.4

1 Introduction

The CPP is widely used in practice to handle portability and variability in C program families [Spencer, 1992]. A program family is a set of programs whose commonality is so extensive that it is advantageous to study their common properties before analysing individual members [Parnas, 1976]. In this context, the

C preprocessor is used to implement these individual members in several domains and open-source C program families of different sizes, such as the *Linux* operating system, *Apache* web server, and *Dia* drawing software.

Despite the widespread use of the CPP, existing studies criticise it due to its negative impact on code quality and maintainability, tool development, and its error-prone characteristics [Kästner et al., 2011, Garrido and Johnson, 2005, Ernst et al., 2002, Liebig et al., 2010, Liebig et al., 2011]. In particular, we aggravate these problems when we use incomplete annotations, i.e., preprocessor directives encompassing only parts of syntactical units [Garrido and Johnson, 2005], leading to problems like annotating an opening bracket without the closing one [Ernst et al., 2002, Baxter and Mehlich, 2001, Liebig et al., 2011]. To better understand incomplete annotations, in a previous study we performed an empirical study on 41 C program families and found that almost 90% of syntax errors occur in incomplete annotations [Medeiros et al., 2013].

In this sense, we consider incomplete annotations as a kind of bad smell in C code [Fowler, 1999]. Previous studies measure that incomplete annotations represent more than 15% of all CPP directives of open-source program families [Liebig et al., 2011]. This way, we can apply refactorings to improve the quality of the code. By removing incomplete annotations, directives appear only in specific places of the code, i.e., only between complete C syntactical units. Further, there are only a few C parsers, e.g., *TypeChef* [Kästner et al., 2011] and *SuperC* [Gazzillo and Grimm, 2012], that deal with incomplete annotations in the literature. Thus, without incomplete annotations, developers can use more tools to analyze the code and find bugs.

There are studies proposing refactorings to remove incomplete annotations in the literature, but these refactorings clone code [Garrido and Johnson, 2005]. Moreover, these studies provide hypotheses that when turning incomplete annotations into complete ones (i.e., preprocessor directives encompassing complete syntactical units), we might increase code cloning, LOC and the number of directives (see Section 2) [Liebig et al., 2011, Garrido and Johnson, 2005, Schulze et al., 2011].

To avoid incomplete annotations, in this article we propose a catalogue of refactorings to remove them without cloning code, i.e., we focus on improving code quality (maintainability) [Arthur, 1988]. Our catalogue defines five categories of refactorings: wrappers, conditions, commands, arrays and enums, and function definitions. Further, we develop an *Eclipse plug-in*—named *Colligens*[1]—that performs our refactorings automatically. We evaluate our catalogue by answering the following research questions:

– Does our catalogue of refactorings increase code cloning?

[1] `https://sites.google.com/a/ic.ufal.br/colligens/`

- Does our catalogue of refactorings increase LOC?
- Does our catalogue of refactorings increase the number of directives?

To answer our questions, we analyse releases of 12 program families of different domains ranging from 4.9 thousand to 1.5 million LOC. We select not only well-known program families that are widely used (e.g., *apache* and *ghostscript*), but also families that are not popular and narrowly used in practice (e.g., *fvwm* and *mptris*). The results reveal that our catalogue can remove all incomplete annotations increasing only in 0.04% the LOC and in 2.10% the number of directives. Further, despite existing studies correlate complete annotations and cloning [Schulze et al., 2011, Liebig et al., 2011], the use of our catalogue does not clone code. In summary, the main contributions of this work are:

- We present a catalogue of refactorings to remove incomplete `#ifdef`[2] annotations without cloning code (Section 3);
- We develop a tool (*Colligens*) that applies our refactorings automatically (Section 4);
- We evaluate our catalogue of refactorings in 12 program families regarding code cloning, LOC and number of directives (Sections 5 and 6).

We organise the remainder of this article as follows. Section 2 shows an incomplete annotation that motivates our study. Then, Section 3 describes our catalogue to remove incomplete annotations, and Section 4 explains our tool support. Afterwards, we present the study settings in Section 5, and discuss the results in Section 6. Last, we present the related work in Section 7 and the concluding remarks in Section 8.

2 Motivating Example

Developers often use preprocessors to handle portability and variability in C program families. For instance, *libpng*[3] is a family implementing the official PNG reference library. Figure 1 (a) presents part of the *libpng* family related to progressive display style, which is useful to read images from the network. Figure 1 (a) contains a preprocessor macro that implements a progressive display style, i.e., `PNG_READ_INTERLACING_SUPPORTED`. The macro uses the interlacing method, which is responsible for encoding a bitmap image. During the download process, we can already see a copy of the whole image despite the incompleteness. It is useful for transmitting images over slow communication links.

[2] We use `#ifdef` as a placeholder for all conditional directives, i.e., `#ifdef`, `#elif`, `#else`, and `#endif`.

[3] `http://www.libpng.org`

```c
1. static void progressive_row(png_structp ppIn){
2.    // Code Here..
3.    if (new_row != NULL){
4.       // Code Here..
5.       if (y >= dp->h)
6.          png_error(pp, "invalid progressive");
7.       row = store_image_row(dp->ps, pp, 0, y);
8.       // Code Here..
9.    }
10.#ifdef PNG_READ_INTERLACING_SUPPORTED
11.   else if (dp->interlace_type)
12.      png_error(pp, "missing row");
13.#endif
14.}
```

(a)

```c
1. static void progressive_row(png_structp ppIn){
2.    // Code Here..
3. #ifdef PNG_READ_INTERLACING_SUPPORTED
4.    if (new_row != NULL){
5.       // Code Here..
6.       if (y >= dp->h)
7.          png_error(pp, "invalid progressive");
8.       row = store_image_row(dp->ps, pp, 0, y);
9.       // Code Here..
10.   } else if (dp->interlace_type)
11.      png_error(pp, "missing row");
12.#else
13.   if (new_row != NULL){
14.      // Code Here..
15.      if (y >= dp->h)
16.         png_error(pp, "invalid progressive");
17.      row = store_image_row(dp->ps, pp, 0, y);
18.      // Code Here..
19.   }
20.#endif
21.}
```

(b)

Figure 1: A refactoring to convert an incomplete annotation into complete ones.

The code snippet in Figure 1 (a) contains an incomplete annotation, i.e., the `#ifdef` directive that starts at line 10 surrounds only part of the `if` statement beginning at line 3. Previous studies [Ernst et al., 2002, Liebig et al., 2010, Liebig et al., 2011, Kästner et al., 2011] criticise the use of incomplete annotations due to its negative impact on code quality, making the tasks of reading and understanding the code difficult.

Garrido and Johnson [Garrido and Johnson, 2005] propose an alternative to remove incomplete annotations as can be seen in Figure 1 (b), i.e., it clones the entire `if` statement code block to transform the incomplete annotation into complete ones. This way, this refactoring introduces clone, increase LOC and add directives. Notice that it may be unfeasible depending on the number of statements in the `if` block. In addition, existing work correlates complete annotations to code cloning [Schulze et al., 2011]. This way, we may face problems by either keeping incomplete annotations or removing them by cloning code.

In this article, we present a catalogue of refactorings to remove incomplete annotations in C program families without cloning code (Section 3). Then, we evaluate our catalogue answering questions related to code cloning, LOC and preprocessor directives (Sections 5 and 6).

3 Catalogue of Refactorings

In this section, we present our catalogue of refactorings to remove incomplete annotations. To define this catalogue, we analyse annotations in different C program families, and identify recurrent incomplete annotations that follow specific syntax structures and occur frequently in real families.

We propose refactorings as transformation templates to remove incomplete `#ifdefs`. Each transformation is an unidirectional refactoring, and consists of two templates of C code snippets, on the left-hand (LHS) and right-hand (RHS) sides. Thus, the LHS template defines a template of C code, which contains preprocessor directives encompassing only part of C syntactical units, i.e., incomplete annotations. Conversely, the RHS template defines a template of the refactored code, i.e., it removes all incomplete annotations. This way, we have only complete C syntactical units inside directives [Garrido and Johnson, 2005, Liebig et al., 2011]. We can apply a refactoring whenever the LHS template is matched by a C code snippet, but satisfying the preconditions ($\rightarrow$). A matching is an assignment of all meta-variables occurring in LHS/RHS models to concrete values. Any element not mentioned in both C code snippets remains unchanged, so the refactoring templates only show the differences between the source codes.

The following refactoring templates relate program families with the same configurations. Moreover, both program families generate products with the same observable behaviour. This notion is a simplified version of the notion proposed by Borba et al. [Borba et al., 2012]. The two program families must have the same feature models, configuration knowledges and asset mappings. Additionally, for each configuration of the source program family, the products of both program families must have the same observable behaviour.

Next, we explain each refactoring of our catalogue. We classify our refactorings into five categories: *wrappers*, *conditions*, *commands*, *arrays and enums*, and *function definitions*.

3.1 Wrappers

In LHS of Refactoring 1 we show a directive encompassing an `else if` statement. The LHS template contains an incomplete annotation and two valid configurations, i.e., with and without macro `expression_1`. The same configurations are valid and syntactically correct in the RHS template. To remove this incomplete annotation, we replace the `else if` by an `if` statement that is complete and does not depend on any other part of the code. Here, we use an extra variable to keep the conditional expression. To easy comprehension, we define this variable (`test`) with the `bool` type, but it is simply an integer. In this sense, we define a precondition that the source code is not using the specific identifier (`test`). We cannot define variables with the same identifier in the same scope. Otherwise, we introduce a compilation error.

Refactoring 1 defines a fine-grained transformation and both templates are equivalent, i.e., they behave in the same way for each configuration. Notice that using variable `test` we avoid the second evaluation of `condition_1` at line 6 (RHS template), eliminating the chances of introducing behavioural changes in conditions with side effects. Moreover, this refactoring does not increase LOC, introduce code cloning, or add extra preprocessor directives.

Refactoring 1 ⟨else if wrappers⟩

```
if (condition_1){
    // Lines of code 1..
}
#ifdef expression_1
    else if (condition_2) {
      // Lines of code 2..
    }
#endif
```

↦

```
bool test = condition_1;
if (test) {
     // Lines of code 1..
}
#ifdef expression_1
    if (!(test) && condition_2) {
        // Lines of code 2..
    }
#endif
```

(→) `test` is not used in the code.

The example presented in Figure 1 (a) matches the LHS template, i.e., the meta-variables assume values *condition_1* = `(new_row != null)`, *condition_2* = `(dp->interlace_type)` and *expression_1* = `PNG_READ_INTERLACING_SUPPORTED`. Thus, we can use this refactoring to remove the incomplete annotation of *libpng*.

We also detect preprocessor directives surrounding only part of `if`, `else`, `while`, and `case` statements. We remove `else` wrappers using a refactoring similar to the one presented in Refactoring 1. In Refactoring 2, we present a refactoring to remove `if` wrappers. Here, we use an extra variable to keep the conditional expression and define a precondition as well. Using this refactoring we maintain the code behaviour and remove the incomplete annotation without duplicating code, or introducing directives. However, we introduce one extra line of code, independently of the size of the statement block. We apply a similar refactoring to remove `while` wrappers.

Refactoring 2 ⟨if wrappers⟩

```
#ifdef expression_1
    if (condition_1)
#endif
{
     // Lines of code..
}
```

↦

```
bool test = 1;
#ifdef expression_1
     test = condition_1;
#endif
if (test) {
     // Lines of code..
}
```

(→) `test` is not used in the code.

In Refactoring 3, we show a refactoring to remove `case` wrappers, which are part of `switch` statements. Here, we use an additional macro to define the `case` statement. Despite we can define the same macro several times, we set a precondition that the code does not define macro `CASE1`. If we change a macro definition that the original source code is already using, we may introduce behavioural changes. Using this strategy, we modify the source code locally without global impact, i.e., without impacting other parts of the code.

Refactoring 3 ⟨case wrappers⟩

```
switch (var) {
#ifdef expression_1
   case VALUE: // Commands
#endif
   // Other cases..
}
```

```
#ifdef expression_1
   #define CASE1 case VALUE: // Commands
#else
   #define CASE1 ""
#endif
switch (var) {
   CASE1
   // Other cases..
}
```

(→) `CASE1` is not used in the code.

Notice that we introduce three additional lines of code using this refactoring, but it does not depend on the size of the case blocks. However, if we have more optional cases we need more lines to remove the incomplete annotations, i.e., if n is the number of optional cases, we need $3 * n$ LOC. Next, we present two particular types of wrappers.

3.1.1 Alternative Statements

Developers also use incomplete annotations to select alternative statements. For example, Refactoring 4 shows an incomplete annotation encompassing alternative `if` statements. In this refactoring, we also use an additional variable to keep the statement condition. This way, we define a precondition (code is not using variable `test`) to avoid compilation errors. The LHS and RHS templates behave in the same way independently if we define `expression_1` or not. In this refactoring, we introduce an extra line of code, no preprocessor directives and we do not clone code to remove the incomplete annotation. We have similar refactorings for alternative control-flow statements (`while` and `switch`).

3.1.2 If Statements ending with an Else Statement

To remove `if` statements ending with `else`, we propose Refactoring 5. We replace the `else` statement by another `if` statement. Here, we use variable `test` to control when we execute the commands of the `else` block. To avoid compilation errors, we define a precondition. When using this refactoring we introduce four lines of code, add an extra directive, but we do not clone code.

Refactoring 4 ⟨alternative if statements⟩

```
#ifdef expression_1
    if (condition_1){
#else
    if (condition_2){
#endif
      // Lines of code..
}
```

```
#ifdef expression_1
    bool test = condition_1;
#else
    bool test = condition_2;
#endif
if (test) {
      // Lines of code..
}
```

(→) `test` is not used in the code.

Refactoring 5 ⟨if statements ending with an else statement⟩

```
#ifdef expression_1
    if (condition_1){
        // Lines of code 1..
    } else
#endif
    {
        // Lines of code 2..
    }
```

```
bool test = condition_1;
#ifdef expression_1
    if (test) {
        // Lines of code 1..
    }
    test = !(test);
#else
    test = 1;
#endif
if (test) {
      // Lines of code 2..
}
```

(→) `test` is not used in the code.

3.2 Incomplete Conditions

To remove incomplete boolean expressions, which are used in conditional statements (`if`) and control-flow statements (`while`), we propose Refactoring 6. In this refactoring, we also use an additional variable to keep the statement conditions and define a precondition to avoid compilation errors. Here, we do not introduce clone, directives or extra lines of code. Both codes behave equivalently since we evaluate the statement conditions only once. We refactor `while` conditions with a similar refactoring.

3.3 Incomplete Commands

Here, we consider command as a simple statement with no compound statement blocks. This way, Refactoring 7 presents our refactoring to complete commands such as returns, attributions, and function calls. We use a `return` statement as an example, but we refactor similar commands in the same way. This refactoring

Refactoring 6 ⟨incomplete if conditions⟩

```
if ( condition_1
#ifdef expression_1
      && condition_2
#endif
){
     // Lines of code..
}
```

→

```
bool test = condition_1;
#ifdef expression_1
     test = test && condition_2;
#endif
if (test) {
     // Lines of code..
}
```

(→) `test` is not used in the code.

does not introduce code cloning, does not add extra preprocessor directives, and needs less LOC than the original code.

Refactoring 7 ⟨returns⟩

```
return id_1
#ifdef expression_1
    && id_2
#else
    && id_3
#endif
;
```

→

```
#ifdef expression_1
  return id_1 && id_2;
#else
  return id_1 && id_3;
#endif
```

3.4 Incomplete Array and Enum Definitions

To remove incomplete annotations surrounding array and enum elements, we use Refactoring 8. Again, we use an additional macro (`ELEMS`) to maintain the array or enum elements. This way, we define a precondition to remove the incomplete annotation with local impact.

Notice that we introduce three additional lines of code when using this refactoring. However, if we have more optional array elements we need more lines, i.e., being n the number of optional elements, we need $3 * n$ LOC. We also add an extra preprocessor directive, but we do not clone the source code.

3.5 Incomplete Function Definitions

We present our refactoring to complete function definitions in Refactoring 9. Here, we use an additional macro (`PARAM`) to keep the optional function parameters. To avoid behavioural changes, we also use a precondition that the original code does not define macro `PARAM`. We add LOC and preprocessor directives in this refactoring, but we do not clone code.

Refactoring 8 ⟨incomplete array definitions⟩

```
type id[] = {
    element_1,
    element_2
#ifdef expression_1
     , element_3
#endif
};
```

↦

```
#ifdef expression_1
    #define ELEMS , element_3
#else
    #define ELEMS ""
#endif
type id[] = {
    element_1,
    element_2
    ELEMS
};
```

(→) `ELEMS` is not used in the code.

Refactoring 9 ⟨incomplete function definitions⟩

```
type function_name (
#ifdef expression_1
    type param_id
#endif
){
    // Lines of code..
}
```

↦

```
#ifdef expression_1
    #define PARAM type param_id
#else
    #define PARAM ""
#endif
type function_name (PARAM){
    // Lines of code..
}
```

(→) `PARAMS` is not used in the code.

4 Tool Support

To perform these refactorings automatically, we implement *Colligens*. Our tool is a *plug-in* for the *Eclipse* platform written in Java, it removes incomplete annotations using the refactorings we present in the previous sections. *Colligens* uses existing tools, such as a variability-aware parser (*TypeChef*)[4] [Kästner et al., 2011] and the *Uncrustify*[5] pretty printer as we explain next.

4.1 TypeChef

TypeChef is a variability-aware parser that can check the presence of syntax and types errors in all possible configurations of the source code. We present an example of C code in Figure 2. As can be seen, this code contains a preprocessor macro (`A`) and two possible configurations, i.e., the first without macro `A` (Program Variant 1), and the second with it (Program Variant 2). Running

[4] `http://ckaestne.github.io/TypeChef/`
[5] `http://uncrustify.sourceforge.net/`

TypeChef to analyze this code, it detects a syntax error when we define macro `A` in Program Variant 2.

After running *TypeChef* in a code without syntax errors, it generates an Abstract Syntax Tree (AST) with variability information (see Figure 3). In this example, we correct the syntax error presented before and present the AST that *TypeChef* generates. The expression node contains two alternative branches depending on the macro `A`. *Colligens* modifies the AST that *TypeChef* creates to refactor the source code, i.e., we change the nodes of the AST to refactor the code. Then, we develop a component that prints the *TypeChef* AST back to source code.

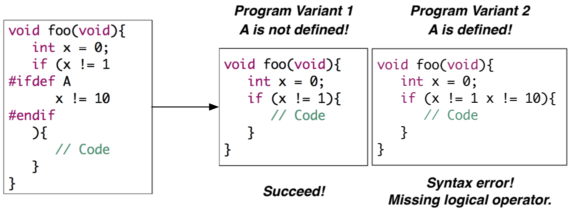

Figure 2: Using *TypeChef* to check the presence of syntax errors.

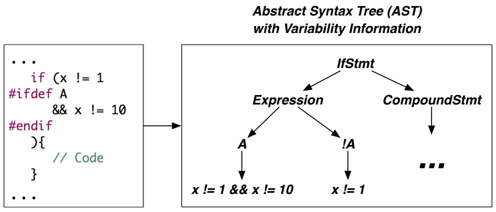

Figure 3: *TypeChef* abstract syntax tree.

4.2 Uncrustify

Uncrustify is a source code beautifier for several languages, including C. We use this tool to align code, parentheses, assignments, variable definitions, structure initializers, preprocessor directives, and so on. We present an example in Figure 4. After translating the *TypeChef* AST back to code, we use *Uncrustify* to organise the source code. Section 4.3 presents our strategy to remove incomplete annotations that uses *TypeChef* and *Uncrustify*.

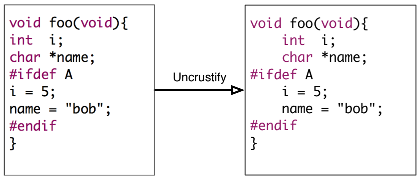

Figure 4: Using *Uncrustify* to pretty print the source code.

4.3 Removing Incomplete Annotations

To remove incomplete annotations, we use the strategy presented in Figure 5 as we explain next:

1. We use *TypeChef* to check the presence of syntax errors and to generate an AST of the original source code;

2. Our refactoring engine uses the AST to refactor the code and remove the incomplete annotations;

3. Then, we use our pretty printer component to pretty print the source code back using the AST. In this step, we use *Uncrustify* to pretty print the source code;

4. Last, we use *TypeChef* again to ensure that our refactorings do not introduce syntax errors in any configuration.

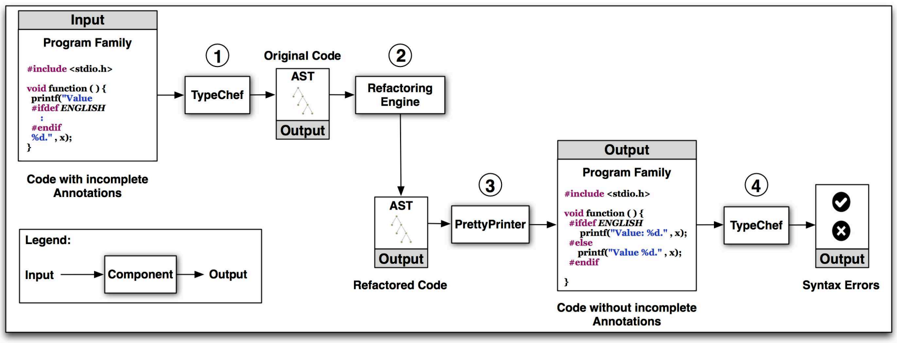

Figure 5: Strategy that *Colligens* uses to remove incomplete annotations.

We present a refactoring example using *Colligens* in Figure 6. In this example, we select a file with an incomplete statement and the tool proposes a refactoring to remove the incomplete annotation. Developers can check the refactored code proposed and accept the changes or not.

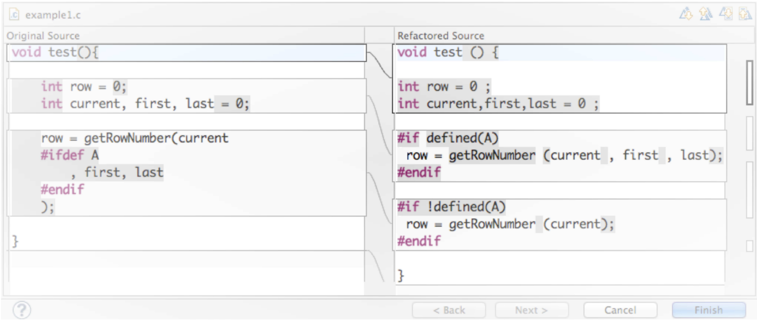

Figure 6: *Colligens* view to refactor incomplete annotations.

5 Study Settings

This section presents the settings of the study we perform to evaluate our refactorings. We use the Goal, Question, Metrics (GQM) approach [Basili et al., 1994].

5.1 Planning

The goal of this study is to evaluate our catalogue of refactorings for the purpose of evaluation with respect to verifying the introduction of code cloning, additional LOC, and preprocessor directives after removing incomplete annotations in the context of C program families. In particular, this study addresses the following research questions:

- **Question 1.** Does our catalogue of refactorings increase code cloning?
- **Question 2.** Does our catalogue of refactorings increase LOC?
- **Question 3.** Does our catalogue of refactorings increase the number of preprocessor directives?

To answer Question 1, we use a similarity analyser to detect clones in the parts modified by the refactorings and compare the original and the refactored source files. In Question 2, we count the LOC of the original source files and the LOC of the source files after applying our refactorings. Regarding Question 3, we count the number of preprocessor directives before and after performing our

Table 1: Subject characterisation

Project Name	**Project Version**	**Application Domain**	**Number of LOC**	**Number of Files**	**Number of `#ifdefs`**
apache	2.4.3	web server	144,768	362	2,173
bc	1.0.3	calculator	5,177	27	91
dia	0.97.2	drawing software	19,106	551	320
expat	2.1.0	XML library	17,103	54	362
flex	2.5.37	lexical analyzer	17,954	45	216
fvwm	2.4.15	windows manager	102,301	270	1,375
ghostscript	9.05	postscript interpreter	1,536,979	3,230	3,168
gnuchess	5.06	chess player	9,293	37	67
gzip	1.2.4	file compressor	5,809	36	298
lighttpd	1.4.30	web server	38,847	132	933
lua	5.2.1	programming language	14,503	59	193
mptris	1.9	game	4,988	29	361
Total			1,916,828	4,832	9,557

refactorings as well. To answer our questions we consider only the source files changed by our refactorings. Next, we describe the subjects and the instrumentation of our study.

5.1.1 Subjects Selection

We analyse 12 program families written in C ranging from 4,988 to 1,536,979 LOC. These program families are from different domains, such as web servers, diagramming programs, and lexical analysers. Table 1 presents the details of each program family, i.e., program family name, version, application domain, total LOC, total number of files, and the total number of annotations.

We select program families analysed by previous studies [Ernst et al., 2002, Garrido and Johnson, 2005, Liebig et al., 2011]. These studies analysed the use of the C Preprocessor in program families of different domains and detected that the majority of the families use the preprocessor in unstructured ways, i.e., they use incomplete annotations. This way, we select program families with incomplete annotations randomly based on these studies.

5.1.2 Instrumentation

We use *TypeChef* version 0.3.3 to parse all possible configurations of the program families to detect syntax errors. Further, we use *Eclipse*[6] 4.2.2 to implement and run *Colligens* to refactor C program families. We use the *Simian*[7] similarity analyser version 2.3.34 to detect clones. We develop a Java tool that counts the number of `#ifdef`, `#elif` and `#else` directives. We use *Uncrustify* version 0.60 to pretty print the source code. Finally, we count the number of lines of code and the number of files of each program family using the Count Lines of Code (CLOC)[8] tool version 1.56, which ignores blank lines and comments.

5.2 Operation

As a first step of our analysis, we run *TypeChef* to find syntax errors. Next, we use our tool to identify recurrent types of incomplete annotations that occur frequently. We define refactorings to remove these types of incomplete annotations. Then, we use *Colligens* to remove the incomplete annotations of the 12 C program families we consider in this study. Last, we run *TypeChef* again in all families to ensure that our refactorings do not introduce syntax errors.

In the second step of our analysis, we use the original and refactored source files to measure cloning, LOC, and the number of directives. To detect small blocks of code cloning, we configure the *Simian* similarity analyser with *threshold = 2*, i.e., detecting any block of cloning with at least two LOC. In addition, we configure the analyser to ignore character cases, curly brackets, and modifiers.

Table 2 presents the number of incomplete annotations, and the total number of clones, lines of code, and preprocessor directives that we introduce using our catalogue of refactorings. As can be seen, the use of our catalogue does not introduce code cloning. In *flex*, we need only 16 extra LOC and no directives to remove all incomplete annotations. On the other hand, we add 39 LOC to remove all incomplete annotations in *mptris*, and 13 directives in *dia*, which represents an increase of 0.78% of LOC, and 4.06% of directives.

We measure the LOC and the number of preprocessor directives that we introduce to remove each incomplete annotation. Table 3 presents the number of incomplete annotations we remove for each category, and the median of extra LOC and directives. For example, to remove an array incomplete annotation we introduce in median 3.06 LOC. On the other hand, the incomplete commands category of incomplete annotation needs only 0.42 lines of code. Regarding extra preprocessor directives, an array incomplete annotation needs in median 0.83 extra preprocessor directives, while the extern category of incomplete annotations

[6] `http://www.eclipse.org/`
[7] `http://www.harukizaemon.com/simian/`
[8] `http://cloc.sourceforge.net/`

Table 2: Results of our refactorings to remove incomplete annotations

Project Name	Number of Incomplete #ifdefs	Code Cloning	Extra LOC	Extra Directives
apache	178	0	257 (0.18%)	48 (2.21%)
bc	6	0	6 (0.12%)	0
dia	31	0	59 (0.31%)	13 (4.06%)
expat	31	0	76 (0.44%)	14 (3.87%)
flex	16	0	16 (0.09%)	0
fvwm	61	0	115 (0.11%)	46 (3.35%)
ghostscript	87	0	143 (0.01%)	30 (0.95%)
gnuchess	2	0	2 (0.02%)	0
gzip	19	0	37 (0.64%)	12 (4.03%)
lighttpd	23	0	33 (0.08%)	11 (1.18%)
lua	6	0	18 (0.12%)	6 (3.11%)
mptris	17	0	39 (0.78%)	11 (3.05%)
Total	477	0	801 (0.04%)	191 (2.10%)

Table 3: LOC and number of directives after applying our refactorings

Category	Incomplete `#ifdefs`	LOC Median	Directives Median
Array	52	3.06	0.83
Commands	53	0.42	0.38
Conditions	39	2.08	0.77
Extern	146	1	0
Function Definition	19	1	0.05
Wrappers	168	2.19	0.64
Total	477	-	-

needs no extra directives. Next, we interpret and discuss the results of this study to evaluate our catalogue.

6 Discussion

In this section, we answer the research questions in Section 6.1, present the benefits of complete annotations in Section 6.2, discuss behavioural changes in Section 6.3 and generalisation of our refactorings in Section 6.4, and present the threats to validity in Section 6.5. All experimental data are available online.[9]

[9] `http://www.dsc.ufcg.edu.br/~spg/jucs2014/`

6.1 Research Questions

Next we answer and discuss the research questions.

6.1.1 Does our catalogue of refactorings increase code cloning?

Our catalogue can remove all incomplete annotations without cloning code (see Table 2). The number of code pairs we measure is zero, i.e., we do not find any block of code cloning with at least two lines of code introduced by our refactorings (we use the *Simian* similarity analyser). Notice that we only analysed the parts of the source code that we modify with our refactorings.

We detect incomplete annotations in statements with thousand lines of code, which would cause a considerable negative impact on the code quality if we remove them using the strategy of Figure 1 [Garrido and Johnson, 2005]. This way, we can apply our refactorings to remove these incomplete annotations without introducing code cloning.

6.1.2 Does our catalogue of refactorings increase LOC?

Our catalogue introduces a few LOC (see Tables 2 and 3). The extra LOC represents 0.04% of the total LOC of all families. The array category of incomplete annotations is the one that requires the highest number of extra LOC. In particular, we need three extra LOC for each optional element of the array.

Regarding the median results, the *array* and *wrappers* categories of incomplete annotations are the ones that requires the highest number of extra LOC. As we can see, the medians of extra lines of code for these categories are 3.06 and 2.19 lines respectively. We detect 168 *wrappers*, out of which 38 (22.62%) require no extra lines of code or less LOC than the original code. On the other hand, we also identify a category, i.e., *incomplete commands*, which our catalogue removes adding only 0.42 extra LOC in median.

6.1.3 Does our catalogue of refactorings increase the number of preprocessor directives?

Our catalogue introduces a few extra preprocessor directives. The extra number of directives represents 2.10% of the total number of directives of all program families. The majority of the incomplete annotations 284 (59.54%) require no extra preprocessor directives, and we need more than one directive to remove 3 (0.63%) of the incomplete annotations.

We remove 477 incomplete annotations, out of which 193 (40.46%) require additional preprocessor directives. The category of incomplete annotations that requires more extra preprocessor directives is the *array* category, we need extra directives to remove 36 (69.23%) of them. On the other hand, we can remove all *extern* incomplete annotation with no extra directive.

6.2 Benefits of Complete Annotations

Using our catalogue we transform all incomplete annotations into complete ones. This way, we find preprocessor directives only in specific places of the source code, i.e., they appear only between complete C syntactical units. In this context, we make tool development easier [Liebig et al., 2011], since tools do not need to consider that preprocessor directives can appear anywhere on the source code.

Besides, in a previous work we identify that almost 90% of the syntax errors occur in incomplete annotations [Medeiros et al., 2013]. Thus, when using only complete annotations we may minimise the chances of introducing subtle syntax errors. Further, incomplete annotations can also cause semantic errors and memory leaks. For example, Figure 7 presents a real code snippet of the *fvwm* project where the use of incomplete annotations can cause a memory leak. In this example, if we define macro `RE_ENABLE_I18N`, we allocate memory to variable `mbcset` at line 4. However, depending on the condition at line 7, we can finalize the function execution at line 13 keeping the memory allocated to variable `mbcset`. Since others functions of the source code use function `parse_bracket_exp`, we can waste a considerable amount of memory.

```
1. static bin_tree_t * parse_bracket_exp (){
2.   // code here..
3. #ifdef RE_ENABLE_I18N
4.   mbcset = (re_charset_t *) calloc (sizeof (re_charset_t), 1);
5. #endif
6. #ifdef RE_ENABLE_I18N
7.   if (sbcset == NULL || mbcset == NULL)
8. #else
9.   if (sbcset == NULL)
10.#endif
11.    {
12.      *err = REG_ESPACE;
13.      return NULL;
14.    }
15.  // code here..
16.#ifdef RE_ENABLE_I18N
17.  re_free (mbcset);
18.#endif
19.  return NULL;
20.}
```

Figure 7: Memory leak example of the *fvwm* program family.

In addition, previous studies criticise the problems of using incomplete annotations, such as its negative impact on code understanding and maintainability [Ernst et al., 2002, Liebig et al., 2011]. In this context, our catalogue may be helpful to remove incomplete annotations that are difficult to deal when reading, modifying, and understanding the source code. However, some refactorings of our catalogue may introduce additional complexities, such as the use of macros. This way, we are still getting the benefits of complete annotations, i.e., development of tool support and syntax errors, but we may not improve code understanding.

6.3 Behavioural Changes

A refactoring is a code transformation that changes the internal structure of the software without modifying its external behaviour [Fowler, 1999]. Thus, to minimize the chances of introducing behavioural changes with our catalogue of refactorings, we check the preprocessed code before and after applying our catalogue. If the original and refactored codes generate the same program variants for all possible configurations of the source code, we improve confidence that there is no behavioural changes (see Figure 8). First, we preprocess the original source code for all possible configurations (*step 1*). Then, we do the same for all possible configurations of the refactored source code (*step 2*). Last, we check if the original and refactored source codes generate the same program variants syntactically (*step 3*). Notice that we can check it locally, i.e., only the part of the code we modify with our refactorings.

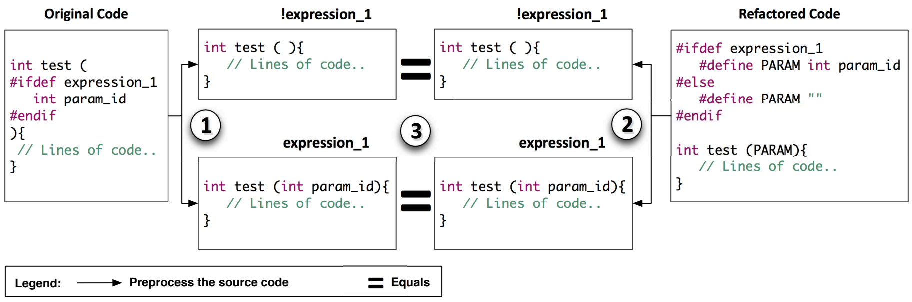

Figure 8: Preprocessing the original and refactored source codes.

However, some refactorings introduce local variables. This way, we cannot check the preprocessed code using the original and refactored codes. To minimize the chances of introducing behavioural changes in these refactorings, we avoid evaluating the statement conditions in a different number of times. For example, on the RHS of Refactoring 1, we use the variable `test` to keep the if statement condition (`condition_1`) and avoid another evaluation at line 6. This way, we avoid behavioural changes in conditions with side effects such as (`++i < 0`). Further, we do not change the order of condition evaluations in our refactorings to avoid behavioural changes as well.

The increase of the number of variables is minimum and may not impact performance. In the families we analyze in our study, it ranges from 0% to 2.75%. For example, *mptris* is the project that we add the highest percentage of variables. We add 3 new variables, but the source code contains only 109 variable declarations, which represents an increasing of 2.75%. On the other hand, we do not add any variables in some projects, such as *bc*, *flex*, *gnuchess* and *lua*.

6.4 Generalisation and Code Cloning

Our catalogue does not clone code to remove incomplete annotations. This way, we define specific refactorings for each type of directive. The refactorings in the literature are more generic, but they clone source code when completing the annotations [Schulze et al., 2013, Garrido and Johnson, 2005, Liebig et al., 2011]. For example, Figure 9 presents a generic refactoring for incomplete conditions. As we can see, this refactoring clones the complete `if` statement to remove the incomplete annotation. Depending on the LOC of the statement block, this refactoring clones several LOC. Using this strategy, we can generalise this refactoring for all kinds of statements, function definitions, and so on. However, to avoid clone, we define specific refactorings as presented in Section 3.

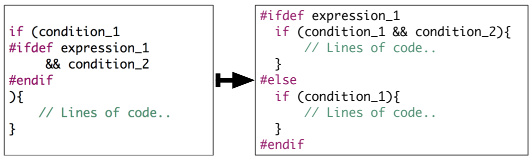

Figure 9: A generic refactoring for `if` statements.

6.5 Threats to Validity

In this section we discuss some threats to validity that are important for the evaluation of our catalogue of refactorings. The next sections present the threats related to construct validity (Section 6.5.1), internal validity (Section 6.5.2) and external validity (Section 6.5.3).

6.5.1 Construct Validity

Construct validity refers to whether our refactorings really remove incomplete annotations without introducing cloning, syntax errors and behavioural changes. We minimise this threat by using *Simian* to measure that our catalogue does not clone code, and a variability-aware parser, *TypeChef*, to check syntax errors before and after performing our refactorings. In addition, our refactorings consist of fine-grained and simple code transformations to avoid behavioural changes.

6.5.2 Internal Validity

The catalogue of refactorings deals with incomplete annotations. Moreover, *TypeChef* changes some incomplete annotations by cloning code to represent them in the AST that it creates, i.e., it uses the strategy of Garrido and Johnson presented in Figure 1 [Garrido and Johnson, 2005]. For example, consider the example presented in Figure 10 (a), it is an incomplete function definition. *TypeChef* transforms the source code into Figure 10 (b). To minimise this threat, we identify the transformations that *TypeChef* clones code and remove the cloning that it introduces as can be seen in Figure 10 (c).

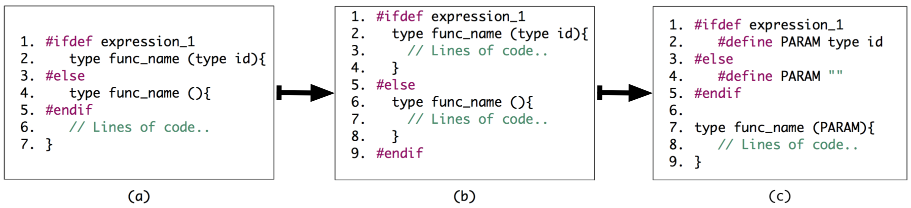

Figure 10: Code transformation that *TypeChef* performs [Kästner et al., 2011], followed by our refactoring to remove code cloning.

6.5.3 External Validity

We refactor 12 C program family releases of different domains and sizes, they range from 4.9 thousand to 1.5 million LOC. We select web servers, diagramming software, lexical analysers and file compressors. Moreover, we select well-known C program families, but also families that are not widely used in practice. In this way, we alleviate this threat but we cannot generalise the results to C program families with more than 5 million lines of code such as *Linux* and *FreeBSD*.

7 Related Work

In this section we present the related work. In Section 7.1 we discuss studies that refactor C program families. Then, we relate our work to studies that propose variability-aware parsers in Section 7.2, and strategies to avoid behavioural changes and check well-formedness in Section 7.3.

7.1 Refactoring C Program Families

Opdyke [Opdyke, 1992] defines refactoring as a behaviour-preserving program transformation. To evaluate behaviour-preserving, he uses successive compilation and tests. However, his work focuses only on refactorings of a single program.

The refactoring of C code is different from refactoring in other languages due to the presence of the C preprocessor. In this context, we have a number of program variants and not a single program. This way, refactoring tools have to consider all possible program variants. Early work on software family refactorings focuses on the software architecture to identify problems and refactor high-level components and connectors [Critchlow et al., 2003]. Alves et al. [Alves et al., 2006] extend the notion of refactorings to software families with refactorings based on feature models.

There are some approaches to refactor C code with preprocessor directives. Garrido and Johnson [Garrido and Johnson, 2003] developed the *CRefactory*, a refactoring tool for C program families that considers all possible configurations. Garrido and Johnson also propose a strategy to remove incomplete annotations [Garrido and Johnson, 2005], but it introduces code cloning (see Section 2). Moreover, *CRefactory* focuses on C refactorings such as renaming functions and extracting macros [Garrido and Johnson, 2013]. Our work has a different focus. We propose C refactorings to the directives themselves to remove incomplete annotations without cloning code. Thus, we minimise the problems related to incomplete annotations, such as syntax errors and code understanding.

Baxter and Mehlich propose *DMS*, a source-code transformation tool for C/C++ [Baxter, 1992]. In a more recent work, they used *DMS* and emphasised the problems of using unstructured annotations, similar to incomplete annotations [Baxter and Mehlich, 2001]. The *DMS* tool focuses on reverse engineering to gather design information and easy maintenance tasks, but not in refactorings.

Vittek presents the *Xrefactory*, a refactoring browser for the C language and discusses certain complications introduced by the C preprocessor [Vittek, 2003]. She uses a strategy that preprocesses the code keeping information about the conditional directives and refactoring the code directly. In our work, we perform our refactorings on the AST. In addition, the AST we use contains variability, i.e., we do not preprocess the code. Thus, we keep all the variability information on the AST.

Tokuda and Batory also propose a refactoring tool to class diagrams of C++ programs [Tokuda and Batory, 2001]. Their work focuses on refactorings of object-oriented systems. Moreover, their work does not deal with preprocessor directives. This way, a simple refactoring such as a function renaming may introduce behavioural changes, since the tool does not change the function name in all possible configurations. Basically, this work refactors only a single C++ program. Our work focuses on refactorings to remove incomplete annotations.

In addition, we refactor a C program family, and not a single program.

Other studies investigate the refactorings of conditional directives into aspects. Adams et al. [Adams et al., 2009] propose an abstract model and analyse the feasibility of refactoring `#ifdef` to aspects, but it does not implement any tool to perform the refactorings automatically. According to their work, it is possible to refactor 99% of the conditional compilation into aspects. Lohmann et al. [Lohmann et al., 2006] refactor the *eCos* operating system kernel using AspectC++, an Aspect-Oriented Programming (AOP) extension to the C++ language, and analyse the runtime and memory costs of aspects. Our work also focuses on refactorings of preprocessor directives, but we refactor the directives without introducing another variability implementation mechanism like aspects.

7.2 Variability-Aware Parsers

In refactorings of C program families, we have to take into consideration all program variants. So, it is important to check if we do not introduce syntax errors and behavioural changes in any possible configuration (program variant). There are some strategies to parse C code with directives. Some approaches [Padioleau, 2009, Somé and Lethbridge, 1998] applied the strategy of preprocessing or modifying the source code before parsing it. However, their strategy is not interesting to refactor incomplete annotations since we lose information about variability.

Kästner et al. [Kästner et al., 2011] propose a variability-aware parser, i.e., a parser that analyse all possible configurations of a C code. In addition, it performs type checking analysis [Kästner et al., 2012]. In our work, we use *TypeChef* to identify syntax errors before and after applying our refactorings.

Gazzillo and Grimm [Gazzillo and Grimm, 2012] propose a variability-aware parser (*SuperC*). It is faster than *TypeChef*, but it does not perform type checking analysis. *SuperC* does not recognise some C constructions of different standards. This way, it does not parse some families we use in this study completely.

7.3 Verifying Behavioural Changes and Well-Formedness

Borba et al. [Borba et al., 2010] define a theory to refactor software families. In their work, they use specific artefacts, such as feature models and configuration knowledges, and propose a theory to detect when a product line refactors another. Further, it defines a theory using a formal specification language and proves some compositionality properties of this theory.

In another study, Ferreira et at. [Ferreira et al., 2012] present an implementation of this software family theory. It proposes tools to evaluate if an SPL transformation preserves behaviour. These tools use test cases to minimise the chances of introducing behavioural changes with refactorings. They are based on *SafeRefactor*, which creates test cases automatically to increase confidence that a transformation preserves behaviour [Soares et al., 2010]. In this study, they define four strategies to identify behavioural changes. In our work we perform only simple and fine-grained refactorings to avoid behavioural changes.

Other studies propose strategies to verify if all program variants are well-formed. They use strategies to verify type errors and missing dependencies, e.g., feature models, configuration knowledges and SAT solvers [Apel et al., 2010, Batory and Thaker, 2007, Delaware et al., 2009], i.e., the safe composition problem. However, existing C program families, such as *Apache*, *Dia* and *Gzip*, do not have some artefacts that these studies uses, e.g., feature models and configuration knowledge. This way, we use fine-grained code transformation to avoid the introduction of syntax errors and behavioural changes.

8 Concluding Remarks

In this article, we presented a catalogue of refactorings to remove incomplete annotations in C program families. Using our catalogue, we complete all the preprocessor directives of 12 families without cloning code differently from previous work [Garrido and Johnson, 2005]. Moreover, we refactor the incomplete annotations introducing 2.10% of new directives and 0.04% of extra LOC.

Our refactorings consist of small, simple, localised, and fine-grained code transformations. This way, it is simpler to reason about their soundness to avoid problems related to bugs in refactoring implementations [Soares et al., 2013]. However, we can compose them to derive coarse-grained transformations, as we did to remove incomplete annotations in 12 program families. This way, we find some evidences that our catalogue is representative. Therefore, we need to consider more program families to get more evidence.

As a future work, we intend to adapt *SafeRefactor* [Soares et al., 2010] to evaluate C program family refactorings by using test suite generators, such as *Pex* and *Randoop* [Pacheco et al., 2008]. Then, we intend to evaluate our refactoring using a similar approach of Ferreira et al. [Ferreira et al., 2012]. In addition, we also intend to remove incomplete annotations in more C program families.

Acknowledgments

This work was partially supported by the National Institute of Science and Technology for Software Engineering (INES), funded by CNPq grants 573964/2008-4, 304470/2010-4, 480160/2011-2, 306610/2013-2, and 477943/2013-6.

References

[Adams et al., 2009] Adams, B., De Meuter, W., Tromp, H., and Hassan, A. E. (2009). Can we refactor conditional compilation into aspects? In *ACM International Con-*

ference on Aspect-Oriented Software Development (AOSD), pages 243–254. ACM.

[Alves et al., 2006] Alves, V., Gheyi, R., Massoni, T., Kulesza, U., Borba, P., and Lucena, C. (2006). Refactoring product lines. In *Proceedings of the 5th International Conference on Generative Programming and Component Engineering*, GPCE '06, pages 201–210, New York, NY, USA. ACM.

[Apel et al., 2010] Apel, S., Kästner, C., Gröblinger, A., and Lengauer, C. (2010). Type safety for feature-oriented product lines. *Automated Soft. Eng.*, 17(3):251–300.

[Arthur, 1988] Arthur, L. J. (1988). *Software Evolution: The Software Maintenance Challenge*. Wiley-Interscience, New York, NY, USA.

[Basili et al., 1994] Basili, V., Caldiera, G., and Rombach, D. H. (1994). The goal question metric approach. In *Encyclopedia of Software Engineering*. Wiley.

[Batory and Thaker, 2007] Batory, D. and Thaker, S. (2007). Safe composition of product lines. In *Proceedings of the 6th International Conference on Generative Programming and Component Engineering*, GPCE '07, pages 1–3.

[Baxter, 1992] Baxter, I. (1992). Design maintenance systems. *Communication of the ACM*, 35(4):73–89.

[Baxter and Mehlich, 2001] Baxter, I. and Mehlich, M. (2001). Preprocessor conditional removal by simple partial evaluation. In *Proceedings of the 8th Working Conference on Reverse Engineering*, WCRE '01, pages 281–290, Los Alamitos, USA. IEEE Computer Society.

[Borba et al., 2010] Borba, P., Teixeira, L., and Gheyi, R. (2010). A theory of software product line refinement. In *Proceedings of the 7th International Colloquium Conference on Theoretical Aspects of Computing*, ICTAC '10, pages 15–43, Berlin, Heidelberg. Springer-Verlag.

[Borba et al., 2012] Borba, P., Teixeira, L., and Gheyi, R. (2012). A theory of software product line refinement. *Theor. Comput. Sci.*, 455:2–30.

[Critchlow et al., 2003] Critchlow, M., Dodd, K., Chou, J., and van der Hoek, A. (2003). Refactoring product line architectures. In *Proceedings of the 1st International Workshop on Refactoring: Achievements, Challenges, and Effects*, REFACE '03, pages 23–26.

[Delaware et al., 2009] Delaware, B., Cook, W. R., and Batory, D. S. (2009). Fitting the pieces together: a machine-checked model of safe composition. In van Vliet, H. and Issarny, V., editors, *Proceedings of the 20th Symposium on the Foundations of Software Engineering*, FSE '09, pages 243–252. ACM.

[Ernst et al., 2002] Ernst, M., Badros, G., and Notkin, D. (2002). An empirical analysis of C preprocessor use. *IEEE Transactions on Software Engineering*, 28:1146–1170.

[Ferreira et al., 2012] Ferreira, F., Borba, P., Soares, G., and Gheyi, R. (2012). Making software product line evolution safer. In *Proceedings of the 6th Brazilian Symposium on Software Components, Architectures and Reuse*, SBCARS '12, pages 21–30, Los Alamitos, CA, USA. IEEE Computer Society.

[Fowler, 1999] Fowler, M. (1999). *Refactoring: Improving the Design of Existing Code*. Addison-Wesley.

[Garrido and Johnson, 2003] Garrido, A. and Johnson, R. (2003). Refactoring C with conditional compilation. In *Proceedings of the 18th Automated Software Engineering*, ASE '03, pages 323–326, Los Alamitos, USA. IEEE Computer Society.

[Garrido and Johnson, 2005] Garrido, A. and Johnson, R. (2005). Analyzing multiple configurations of a C program. In *Proceedings of the 21st IEEE International Conference on Software Maintenance*, ICSM '05, pages 379–388. IEEE.

[Garrido and Johnson, 2013] Garrido, A. and Johnson, R. (2013). Embracing the C preprocessor during refactoring. *Journal of Software: Evolution and Process*.

[Gazzillo and Grimm, 2012] Gazzillo, P. and Grimm, R. (2012). SuperC: parsing all of C by taming the preprocessor. In *Proceedings of the 33rd Programming Language Design and Implementation*, PLDI '12, pages 323–334, New York, USA. ACM.

[Kästner et al., 2012] Kästner, C., Apel, S., Thüm, T., and Saake, G. (2012). Type checking annotation-based product lines. *ACM Transactions on Software Engineering and Methodology*, 21(3):14:1–14:39.

[Kästner et al., 2011] Kästner, C., Giarrusso, P., Rendel, T., Erdweg, S., Ostermann, K., and Berger, T. (2011). Variability-aware parsing in the presence of lexical macros and conditional compilation. In *Proceedings of the 26th Object-Oriented Programming Systems Languages and Applications*, OOPSLA '11, New York, USA. ACM.

[Liebig et al., 2010] Liebig, J., Apel, S., Lengauer, C., Kästner, C., and Schulze, M. (2010). An analysis of the variability in forty preprocessor-based software product lines. In *Proceedings of the 32nd International Conference on Software Engineering*, ICSE '10, pages 105–114, New York, USA. ACM.

[Liebig et al., 2011] Liebig, J., Kästner, C., and Apel, S. (2011). Analyzing the discipline of preprocessor annotations in 30 million lines of C code. In *Proceedings of the 10th Aspect-Oriented Software Development*, AOSD '11, pages 191–202. ACM.

[Lohmann et al., 2006] Lohmann, D., Scheler, F., Tartler, R., Spinczyk, O., and Schröder-Preikschat, W. (2006). A quantitative analysis of aspects in the eCos kernel. In *ACM Euro Conference on Computer Systems (EuroSys)*, pages 191–204. ACM.

[Medeiros et al., 2013] Medeiros, F., Ribeiro, M., and Gheyi, R. (2013). Investigating preprocessor-based syntax errors. In *Proceedings of the 12th International Conference on Generative Programming: Concepts and Experiences*, GPCE '13, pages 75–84, New York, NY, USA. ACM.

[Opdyke, 1992] Opdyke, W. (1992). *Refactoring Object-Oriented Frameworks*. PhD thesis, University of Illinois at Urbana-Champaign (UIUC).

[Pacheco et al., 2008] Pacheco, C., Lahiri, S. K., and Ball, T. (2008). Finding errors in .NET with feedback-directed random testing. In *Proceedings of the 6th Symposium on Software Testing and Analysis*, ISSTA '08, pages 87–96, NY, USA. ACM.

[Padioleau, 2009] Padioleau, Y. (2009). Parsing C/C++ code without pre-processing. In *Compiler Construction*, volume 5501 of *Lecture Notes in Computer Science*, pages 109–125. Springer Berlin Heidelberg.

[Parnas, 1976] Parnas, D. (1976). On the design and development of program families. *IEEE Transactions on Software Engineering*, 2:1–9.

[Schulze et al., 2011] Schulze, S., Jurgens, E., and Feigenspan, J. (2011). Analyzing the effect of preprocessor annotations on code clones. In *IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM)*.

[Schulze et al., 2013] Schulze, S., Liebig, J., Siegmund, J., and Apel, S. (2013). Does the discipline of preprocessor annotations matter?: a controlled experiment. In *Proceedings of the 12th International Conference on Generative Programming: Concepts and Experiences*, GPCE '13, pages 65–74.

[Soares et al., 2013] Soares, G., Gheyi, R., and Massoni, T. (2013). Automated behavioral testing of refactoring engines. *IEEE Transactions on Soft. Eng.*, 39(2):147–162.

[Soares et al., 2010] Soares, G., Gheyi, R., Serey, D., and Massoni, T. (2010). Making program refactoring safer. *IEEE Software*, 27:52–57.

[Somé and Lethbridge, 1998] Somé, S. and Lethbridge, T. (1998). Parsing minimization when extracting information from code in the presence of conditional. In *Proceedings of the Workshop on Program Comprehension*, IWPC '98, pages 118–125.

[Spencer, 1992] Spencer, H. (1992). Ifdef considered harmful, or portability experience with C news. In *USENIX Annual Technical Conference*, pages 185–197.

[Tokuda and Batory, 2001] Tokuda, L. and Batory, D. (2001). Evolving object-oriented designs with refactorings. In *Proceedings of the 14th International Conference on Automated Software Engineering*, ASE '01.

[Vittek, 2003] Vittek, M. (2003). Refactoring browser with preprocessor. In *Proceedings of the 7th European Conference on Software Maintenance and Reengineering*, CSMR '03. IEEE Computer Society.