Analyzing the Effect of Preprocessor Annotations on Code Clones

Sandro Schulze
University of Magdeburg
sanschul@ovgu.de

Elmar Jürgens
Technische Universität München
juergens@in.tum.de

Janet Feigenspan
University of Magdeburg
feigensp@ovgu.de

Abstract

The C preprocessor cpp *is a powerful and language-independent tool, widely used in different programming languages (C, C++, Java). One core feature of the cpp is that it allows to express variability in programs using conditional compilation. To this end, the code can be annotated on different levels of granularity such as functions or statements. Depending on this granularity, we differentiate between* disciplined *and* undisciplined *annotations. In this paper, we investigate whether there is a relation between code clones and preprocessor annotations. Specifically, we address the question whether the discipline of annotation has an effect on code clones. To this end, we perform a case study on fifteen different C programs and analyze them regarding code clones and* `#ifdef` *occurrences. We found only minor effects of annotations on code clones but a relationship between disciplined annotations (and code clones). With this work, we provide new insights on how/why code clones are created. Furthermore, the results can support the decision whether it is beneficial to remove clones or not.*

1. Introduction

The *cpp* preprocessor is a powerful text processing tool tightly coupled with the C programming language [1]. Due to its token-based nature, the cpp is language-independent and by now, even used with many other languages including C++ and Java. The major advantage of the cpp is that it provides expressive capabilities to introduce variability into programs using conditional compilation. In fact, preprocessor directives (or annotations)[1], such as `#ifdef`, `#ifndef` etc., can be used on any level of granularity. Conversely, this flexibility makes it a root for poor code quality, caused by the missing structure of the cpp tool. Amongst others, the cpp is considered to be error prone and to impair readability and maintainability of the code [2], [3], [4], [5]. A pivotal role for the effect of annotations on source code quality is whether these annotations are *disciplined* or *undisciplined*. It is commonly accepted that undisciplined annotations contribute to unstructured, tangled source code with the mentioned negative effects [6], [2], [7], [8].

In Figure 1, we show two code fragments containing an undisciplined and a disciplined annotation, respectively. Undisciplined annotations (Figure 1 (a)) are made on arbitrary syntactical units, such as parameters or branch conditions, and do not align with the overall code structure. By contrast, disciplined annotations (Figure 1 (b)) are mapped to corresponding syntactical units such as functions or statements and thus align with the code structure. As a result, it is commonly accepted that disciplined annotations alleviate the drawbacks of annotations on source code quality [2], [9], [10]. However, we and others observed that disciplined annotations may lead to replicated code fragments, commonly known as *code clones* [6].

Code clones are considered to be common in software development [11], [12], [13]. Furthermore, several studies reveal that code clones have a negative effect on software structure, leading to

1. Although there are different possibilities of preprocessor directives with cpp, we focus only on *conditional inclusion* within this paper.

```
class Stack {
  void push(Object o
#ifdef SYNC
  , Transaction txn
#endif
  ){
      if (o==null
#ifdef SYNC
      || txn==null
#endif
      )
      return;
#ifdef SYNC
      Lock l=txn.lock(o);
#endif
      elementData[size++] = o;
#ifdef SYNC
      l.unlock();
#endif
      fireStackChanged();
  }
}
```

(a) undisciplined

```
class Stack {
#ifdef SYNC
  void push(Object o,
      Transaction txn) {
      if (o==null || txn==null)
          return;
      Lock l = txn.lock(o);
      elementData[size++] = o;
      l.unlock();
      fireStackChanged();
  }
#else
  void push(Object o) {
      if (o==null)
          return;
      elementData[size++] = o;
      fireStackChanged();
  }
#endif
}
```

(b) disciplined

Figure 1. Example for effect of annotations on code cloning

increased maintenance effort, inconsistent changes, and introduction of errors [14], [15], [16], [17]. Beyond that, several studies exist that aim at identifying causes for code clones as well as evaluating the effect and occurrence of clones on the software system [18], [19], [20]. Unfortunately, these studies do not investigate the effect of preprocessor annotations.

Research problem. Disciplined annotations align with the source code and thus limit expressiveness. On the other hand, they alleviate the drawbacks of annotations on source code quality. Furthermore, recent observations indicate that such annotations cause code clones. However, up to now it is unclear whether disciplined annotations lead to an increased amount of code clones as a matter of fact. If this is the case, this poses the question whether replacing undisciplined annotations by disciplined ones (at the expense of code clones) really improve the source code quality.

Contribution. In this work, we extend existing studies by a large case study that investigates the relation between code clones and preprocessor annotations. As a result, we provide new insights on *why* and *where* code clones occur. Furthermore, the information of our analysis can be useful to support the decision whether to remove clones or not.

2. Background

A variety of research has been done for both, code clones and preprocessor directives. In this section, we give an overview of terms and definitions relevant in this paper.

2.1. Code Cloning

Code clones are source code fragments that are similar to each other. Different types of clones exist depending on the degree of similarity between two code fragments [21]. Code fragments that are identical are called *type-I* or *exact clones*. Furthermore, code clones that differ only slightly are called *type-II* clones. For instance, differences due to renaming of variables or constants typically lead to type-II clones. Finally, *type-III* or *gapped clones* are similar code fragments that differ due to adding, removing, or changing code units of at least one of the code fragments. Finally, two code fragments that are identified as code clones are called a *clone pair*. Additionally, a set that consists of two or more code clones is a *clone group*.

Different approaches exist to detect the different types of code clones. First, *text-based* approaches use simple string or character comparison and thus detect only *type-I* clones [22]. Second, *token-based* approaches perform a tokenization on the source code and compare these tokens to detect *type-I* and *type-II* clones [23]. Third, *AST-based* approaches create a grammar-based abstraction of the source code, the *Abstract Syntax Tree (AST)*. A sophisticated variant is the *PDG-based* approach hat additionally takes data and program flow into account. Both approaches detect *type-I* and *type-II* clones, based on their internal source code representation [12], [24]. Finally, some approaches can even detect *type-III* clones such as ConQAT that defines a threshold for the maximum edit distance between clones [25].

2.2. Annotations in cpp

Annotations, or more specifically conditional inclusion using `#ifdef`[2], can be used to generate different variants (with different functionality) of a program [7]. Each annotation contains a boolean expression that is evaluated by the cpp tool to determine whether the corresponding code is included in a certain program or not. Usually, such an expression represents a *feature*, an increment in user-visible functionality [26]. As a result, an annotated program is rather a set of programs and thus, can be considered as a *Software Product Line (SPL)* [27].

Generally, annotations can be classified into two categories, based on the syntactical units they annotate: *disciplined* and *undisciplined* annotations [6]. This, in turn, raises the question where the borderline between these two categories is. Obviously, defining such a borderline depends on several criterias. Liebig et al. propose a definition for disciplined annotations and accordingly, annotations of one or a sequence of functions, type definitions, statements, and elements inside type definitions are disciplined [6]. We rely on this definition within this paper since it is reasonable and suitable for our purposes. For clarification, we depict some examples for both kinds of annotations in Figure 2. For a more detailed overview of undisciplined annotations we refer to the work of Liebig et al. [6].

3. Code Clone Analysis Process

In this section, we describe the case study we performed. First of all, we introduce research questions, we address by means of our case study. Afterwards, we present the general design of the study and the C programs that were subjects in our study.

3.1. Research Questions

We perform the case study to gain insights on the relation between code clones and annotations

2. Within this paper, `#ifdef` is a placeholder for all possibilities of conditional inclusion: `#ifndef`, `#if`, `#elif`, `#else` and `#endif`

```
  need_redraw =
     check_timestamps(
#ifdef FEAT_GUI
     gui.in_use
#else
     FALSE
#endif
  );
```

```
  int n = NUM2INT(num);
#ifndef FEAT_WINDOWS
  w = curwin;
#else
  for (w = firstwin; w != null;
         w = w−>w_next,
            −−n)
#endif
    if (n == 0)
      return window_new(w);
```

(a) examples for undisciplined annotations

```
void tcl_end() {
#ifdef DYNAMIC_TCL
  if (hTclLib) {
    FreeLibrary(hTclLib);
    hTclLib = NULL;
  }
#endif
}
```

```
typedef struct {
  typebuf_T save_typebuf;
  int typebuf_valid;
  struct buffheader
     save_stuffbuff;
#if USE_INPUT_BUF
  char_u *save_inputbuf;
#endif
} tasave_T;
```

(b) examples for disciplined annotations

Figure 2. Examples for undisciplined and disciplined annotation

in preprocessor-based programs. To achieve our goals, we formulate research questions that we address by means of our study.

RQ 1 *To what extent do code clones occur in annotated* `#ifdef` *blocks ?*

Several studies reveal the existence of code clones in C programs. However, none of these studies analyzes how much of the detected code clones occur in preprocessor blocks. We aim at answering this question with the help of our code clone analysis. As a result, we can evaluate whether preprocessors are prone to code clones or not.

RQ 2 *Are there differences between disciplined and undisciplined annotations regarding code clone occurrence ?*

This question is motivated by the observation that disciplined annotations may come at the expense of introducing code clones [6]. Consequently, we evaluate whether this observation is accidental or may depend on the discipline of annotations. Answering this question may also affect the evaluation of code clones regarding their harmfulness. For instance, if a code clone is introduced in order to overcome undisciplined annotations, should the clone considered as harmful ? As a direct consequence, this information may also support refactoring decisions in terms of code clone removal.

3.2. Study Design

This section gives an overview of our study design, especially of the measures we compute to address the research questions.

For our study we perform a code clone analysis supplemented by clone detection and source code analysis. The detailed process is described in Section 3.3. As a result, we gain information on the amount of code clones, *#ifdef code* (i.e., code that is contained between `#ifdefs`), and *#ifdef clones* (i.e., code clones that are enclosed by `#ifdefs`). Subsequently, we compute different measures based on these analysis results .

First, we compute the code clone and #ifdef coverage.The term coverage denotes the part of the source code that is covered by code clones or `#ifdef` blocks either. These measures provide us with general information on the systems and whether it is worth to further investigate these systems or not.

Second, we compute the #ifdef clone coverage to investigate how much of the overall code contains #ifdef clones. Additionally, we compute the ratio of #ifdef clones compared to a) all detected code clones (*#ifdef-clone/clone ratio*) and b) the total amount of annotated code (*#ifdef-clone/#ifdef ratio*). With these measures we can determine whether there other correlations that are likely to cause #ifdef clones.

Finally, we compute all of our measures for disciplined and undisciplined systems separately and thus, can compare both categories.

3.3. Analysis of `#ifdef` Clones

In this section, we give an overview of the design of our code clone analysis process. For answering our research questions, we set up a three-staged process, which we depict in Figure 3. In the following we explain the three phases *clone detection, source code analysis* and *code clone analysis* in detail.

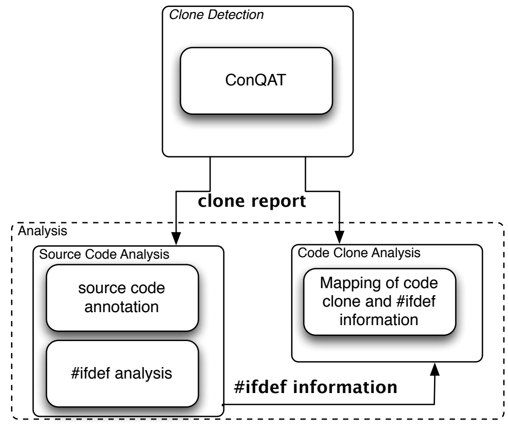

Figure 3. Outline of the clone analysis process

Clone Detection. For clone detection, we use ConQAT[3], a token-based clone detection tool that can detect *gapped clones*. This is an important fact, since such clones may occur within disciplined annotations, as indicated by our example in Figure 1. Initially, the source code is transformed into a token sequence while comments and whitespaces are removed. Afterwards, ConQAT performs normalization on the token sequence, which can be divided into two parts. First, statements are created from the token sequence since it leads to better clone detection results (e.g., ignoring clones that start/end within statements). Second, tokens are normalized by user-defined rules, which eliminates differences between the specified syntactical units such as identifiers or constants. For instance, we set up the normalization in such a way that differences between literals such as boolean, string, or numbers are ignored for the actual clone detection. By contrast, we do not normalize differences between identifiers.

Finally, the clone detection is performed on the normalized token sequence. In a nutshell, a suffix tree is built on the token sequence and then, the algorithm searches for all identical or similar substrings in the tree. The user can influence the clone detection result by specifying different pa-

3. www.conqat.org

rameters such as the minimum clone length. For our purposes, we selected a minimum clone length of eight statements. Furthermore, we performed a gapped clone detection so that gapped clones are detected as well. Therefore, we have to specify the *gap ratio*, a measure that determines the maximum number of gaps between two code clones. We selected a gap ratio of 0.25, which means for a clone pair of eight statements that two statements of at least one clone may have been deleted, added, or changed. The result of the clone detection is subject to post-processing such as filtering out overlapping clone groups. At the end, a clone report is generated, containing information on all source files as well as on all clone groups and its corresponding code clones, that can be used for further usage. For a more detailed description of the clone detection algorithm, we refer to [14].

Source Code Analysis. For obtaining information on occurrences of annotations, we have to analyze the source code that was subject to clone detection in the previous step. To this end, we firstly annotate corresponding source files using *src2srcml*[4], a source code markup language that annotates the source code in an xml-like fashion without breaking its overall structure [28]. Afterwards, we detect `#ifdefs` in the annotated code using *XPath*, an xml path language that can be used to navigate through the nodes of an xml document. As a result, we obtain all occurrences of `#ifdef` annotations, identified by their absolute position (i.e., line number) in the source file. Note, that we get this information for *complete* `#ifdef` blocks, that is, code fragments that are enclosed by annotations such as `#ifdef` and `#endif`. Finally, the results of this analysis can be used for code clone analysis.

Code Clone Analysis. For our analysis, we map the detected code clones to the detected `#ifdef` annotations, based on their absolute position in the source file. We illustrate the respective mapping algorithm in Figure 4.

Our algorithm has two inputs: A list of all clone groups from the clone report, and a list of preprocessor annotations together with the information where they can be found (i.e., the file and the line number). For the mapping, we consider the code clones of each clone group separately. Then, we compare the position of each clone with the positions of all preprocessor annotations that have been found in the file containing the clone. We have a match, if at least one *complete* `#ifdef` block is within the clone. Hence, `#ifdef` blocks that are not entirely located within a code clone (e.g., they start or end outside the clone) are ignored. We imposed this restriction because partial #ifdef clones may lead to less accurate results. Furthermore, the `#ifdef` block must have a length of at least three source lines of code, excluding the lines containing the annotations, so that we can prevent #ifdef clones that occur accidental. In case of a match, we create a new *#ifdef clone*. We do this for all clones of a clone group. Finally, all corresponding ifdef clones are merged to an *#ifdef clone group*.

4. http://www.sdml.info/projects/srcml/

proc $mapClones(cg, pa)$
Input: cg = list of clone groups, pa = list of annotations
for each $cg_i \in cg$ **do**
  get all code clones from cg_i
  for each $clone$ **do**
    $fa \subset pa$ = all annotations for the file containing the clone
    if $(a \in fa)$ is within $clone$ **then**
      $clone'$ = new clone with the position of a
      create new ifdef clone group with $clone'$ (if this is the first $clone$ of cg_i)
      otherwise add $clone'$ to existing ifdef clone group (for cg_i)
    end if
  end for
end for
Result: list of `#ifdef` clone groups

Figure 4. Algorithm for mapping preprocessor annotations to code clones

3.4. Study Objects

For our case study, we use fifteen software systems written in the C programming language. We selected programs of different size and domains to have a representative sample. Furthermore, to evaluate our second research question, we split the sample in two comparable groups: Seven systems are disciplined (i.e., contain mostly disciplined annotations), eight systems undisciplined (i.e., contain $12\,\%$ undisciplined annotations in average). The classification is based on a recent study of Liebig et al., who analyzed the discipline of annotations in C programs [6]. Beyond that, both groups are comparable regarding size and domains. In Table 1 we give an overview of the analyzed programs.

	program	# SLOC	description
undisciplined	cherokee	47 983	Web server
	gnuplot	67 854	plotting tool
	lynx	111 994	Web browser
	php	471 604	program interpreter
	privoxy	24 784	proxy server
	sendmail	85 094	mail transfer agent
	tcl	122 460	program interpreter
	vim	233 426	text editor
disciplined	berkeleyDB	160 283	database system
	dia	121 117	diagramming software
	ghostscript	491 703	postscript interpreter
	lighttpd	37 380	Web server
	minix	54 627	operating system
	parrot	84 222	virtual machine
	python	331 014	program interpreter

Table 1. Overview of analyzed C Programs

4. Results

In this section, we present the results of our case study.

First of all, we observed that in all systems, annotated code as well as code clones exist. We show the results in Figure 5. In undisciplined systems (Figure 5 a) the coverage of annotated code is $32,7\,\%$, whereas it is $16,7\,\%$ on average for the disciplined systems (Figure 5 b). By contrast, the disciplined systems exhibit a higher clone coverage ($10\,\%$ onn average) compared to the undisciplined systems ($4,3\,\%$ on average). Furthermore, six of eight undisciplined systems have a clone coverage less than $3\,\%$. Nevertheless, all systems contain code clones as well as `#ifdef` annotated code in a reasonable amount and thus, are further investigated.

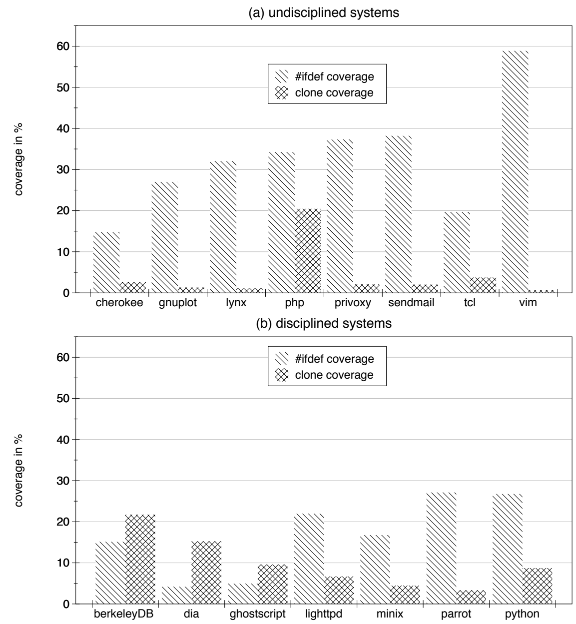

Figure 5. Clone and `#ifdef` coverage of the analyzed systems (note the different scales)

To evaluate RQ 1, we measured the #ifdef clone coverage supplemented by the *clone/#ifdef-clone* and the *#ifdef/#ifdef-clone* ratio. We depict our results in Figure 6. Note that black spots indicate undisciplined systems and white spots indicate disciplined systems. The scatter plot in Figure 6 a indicates that the #ifdef clone coverage is rather small. Indeed, only four systems exhibit a coverage of more than $0,5\,\%$, and only one system (*BerkeleyDB*) has an #ifdef clone coverage higher than $2\,\%$. In the same way, Figure 6 b indicates that only a minor fraction of all detected code clones occur within #ifdef blocks, independent of the actual amount of clones. Similarly, only a small

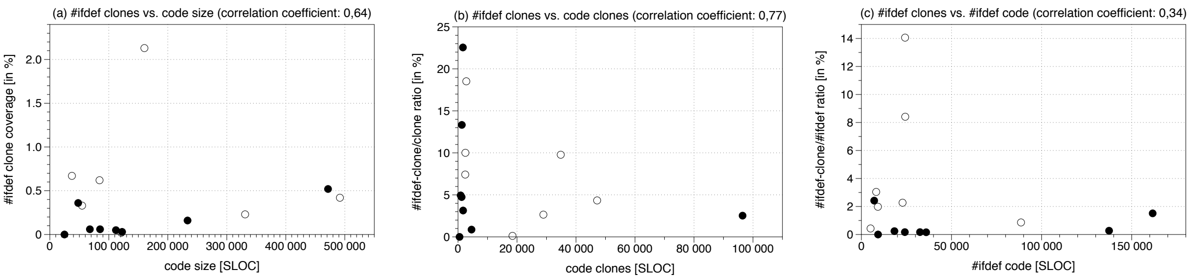

Figure 6. Analysis results for #ifdef clones: a) w.r.t the overall code size, b) w.r.t all code clones c) w.r.t annotated code

fraction of the overall #ifdef code contains code clones (Figure 6 c). We observed that those systems with the highest amount of #ifdef code contain only few #ifdef clones. By contrast, amongst the systems with a rather small amount of #ifdef code ($< 25K$ SLOC), in particular systems #ifdef code contains up to $14\,\%$ #ifdef clones.

With RQ 2, we aim at investigating whether there are differences between disciplined and undisciplined systems regarding the amount of #ifdef clones. Therefore, we compare the amount of #ifdef clones in disciplined and undisciplined systems. Although #ifdef clone coverage is rather low, Figure 6 a reveals that the amount of #ifdef clones is considerable higher in disciplined systems. We observed that five of the eight undisciplined systems have a coverage close to zero and the average #ifdef clone coverage is $0,15\,\%$. By contrast, four of the seven disciplined systems have an #ifdef clone coverage of approximately $0,5\,\%$ or higher ($0,63\,\%$ in average). Furthermore, we observed that annotated code in disciplined systems is more likely to contain clones than in undisciplined systems (cf. Figure 6 c). Although the latter systems contain more annotated code, in six systems this code contains nearly no code clones. Contrary, out of the six system that contain $2\,\%$ or more code clones in #ifdef blocks, five are disciplined systems. All of these observations are confirmed by a medium or even high correlation coefficient we computed, using the method of Pearson.

5. Discussion

The results we presented in the previous section imply that the effect of `#ifdefs` on code clones is rather small, but that there are differences between systems with disciplined/undisciplined annotations. In the following we interpret these results and discuss threats to validity of our case study.

5.1. Interpretation of Results

The results of RQ 1 reveal that the amount of code clones in preprocessor annotation is rather small (with minor exceptions). This especially holds for the systems with a high amount of undisciplined annotations, where all measured data indicate that #ifdef clones are negligible. However, due to our very strict definition of #ifdef clones (only complete annotated blocks are considered), this result can be interpreted as a lower bound. Consequently, a more fine-grained investigation of #ifdef clones, where partial #ifdef clones are allowed up to a certain threshold, may even increase their overall amount. Beside this, a larger case study with more systems can also support the investigation of #ifdef clones in cpp-based programs.

For RQ 2, the results indicate that #ifdef clone coverage is significantly higher for disciplined systems compared to undisciplined systems. This, in turn, confirms the assumption that disciplined annotations lead to code clones. Furthermore, our data reveal that other factors such as code size and amount of code clones do not influence this observation and thus, can be neglected.

To evaluate whether the observed differences between disciplined and undisciplined systems are significant or rather occurred randomly, we conducted a significance test. To this end, we applied an adapted version of the Mann-Whitney-U test: Instead of providing the significance level of the test, we check whether the calculated U values are significant according to a table specifically designed for small sample sizes [29], [30]. The results of the Mann-Whitney-U test reveal that the differences for the #ifdef/#ifdef-clone ratio as well as the #ifdef clone coverage are significant. First, for the #ifdef/#ifdef-clone ratio, we obtained the following results: $U = 6$, $p < 0.01$. Second, for the #ifdef clone coverage our test produced the following results: $U = 12$, $p < 0.05$. Since both significance levels are smaller than 0.05, we can assume that the differences we observed are significant and not caused randomly.

Revisiting the introduced research problem, our study results indicate that disciplined annotations increase the amount of code clones compared to undisciplined one. However, due to the small #ifdef clone coverage, the effects of these clones may be not as negative as for undisciplined annotations. Consequently, the benefits of disciplined annotations outweigh the drawbacks of code cloning and thus code clone removal may be not necessary.

5.2. Threats to Validity

Single Programming Language. Although the cpp tool is language-independent and thus used with several programming languages, we only considered C programs within our study for two reasons. First, for the selected systems we had prior knowledge about the discipline of annotations, based on the study of Liebig et al. [6]. Second, with the decision for one specific language, we can prevent that certain mechanisms of different languages influence the amount of (#ifdef) code clones. Finally, the cpp tool has been used with C programs for a long time and thus the systems are mature and different case studies exist. Overall, we assume that this decision may limit generalizability but does not affect our results.

Selected Software Systems. With case studies on software systems, there is a risk that the selected systems bias the study results. To mitigate this effect, we selected systems of different size and of different domains as far as this was possible.

Clone Detection. Both, initial clone detection as well as #ifdef clone detection, have been performed automatically, based on certain input parameters. Due to the large code amount, it is impossible to check each clone regarding the precision of the clone detection. However, we randomly selected samples (for code clones *and* #ifdef clones) from each subject system for a manual review process. All of these samples w ere true code clones that is, we detected no false positives. Furthermore, for the clone detection we selected parameters that are commonly accepted and thus prevented that meaningless or even false code clones are detected.

Study evaluation. During interpreting the results of our case study, we made several observations regarding our research questions. To strengthen our results, we conducted statistical computations, namely Pearson's correlation coefficient and a significance test. The main problem of the statistical evaluation is the quite small sample set represented by the fifteen subject systems. Although the statistic results indicate the correctness of the relations we observed, we have to be careful with the conclusions we draw. Hence, a larger case study with more systems is necessary. Nevertheless, our study results provide first insights on the relation between preprocessor annotations and code clones.

6. Related Work

For Both, the preprocessor cpp as well as code cloning, we give an overview of prior work that is related to ours.

For cpp, Ernst et al. conducted a large case study that aim at investigating the usage of preprocessor annotations and its implications [2]. In their work, they highlight advantages of disciplined preprocessor use and why this fails regularly. However, they mainly focus on usage of macro definition such as `#define` and thus conditional inclusion is just mentioned partly. Furthermore, preprocessors are not considered in the context of clones (and vice versa). Recently, Liebig et al. presented comprehensive results regarding preprocessor usage on which we partially base our paper. First, they analyzed how the cpp tool is used to implement variability in programs. Therefore, they conducted a large case study and defined several metrics for measuring system properties such as granularity or types of extensions [31]. Although they provide some metrics that may indicate the occurrence of code clones such as homogeneous extensions, they do not focus on code clones explicitly. Second, they analyzed the discipline of annotations in cpp-based programs [6]. In this context, they give a definition for disciplined annotations and conducted a case study on how disciplined and undisciplined annotations are used. Interestingly, they state that there is a trade-off between expressiveness due to undisciplined annotations and code replication due to disciplined annotations. However, different to our work they did not analyze to what extent code clones occur in systems with different discipline of annotations.

Amongst the several case studies on code clones, some of them specifically focus on clones in C programs and thus, are related to our work. First, Mayrand et al. presented an experiment on function clones in C programs [18]. Within their work they propose a taxonomy for function clones as well as different notions of similarity, based on metrics. However, their code clone analysis focus only on function blocks and thus, they do not consider preprocessors as we do. Second, Roy et al. conducted a large case study on code clones in open source systems (C & Java) [19]. They propose different metrics such as clone density, clone location and clone size for comprehensive insights on cloned code and verified their results manually. Nevertheless, they also focus rather on clones on function level. By contrast, we considered code clones in the context of preprocessor annotations.

7. Conclusions and Future Work

In this paper, we conducted an empirical study on fifteen C systems to analyze the effect of preprocessor annotations on code clones. Based on our results, we detected only minor relations between annotations and code clones. Hence, we state that preprocessor annotations have no obvious effect on code cloning. Beyond that, we observed significant differences between systems with disciplined and undisciplined annotations. Our results indicate that systems with disciplined annotations are more prone to code clones than systems with undisciplined annotations. However, due to the small amount of #ifdef clones we conclude that it is probably more beneficial to manage these clones instead of remove them (and probably introduce undisciplined annotations).

Regarding our results, there are still questions that remain unanswered: For what types of annotations the code clones occur in particular? Is there an overall relation between code clones and variability in cpp-based systems? How does our result change for different types of clones (e.g., only type-II clones or partial #ifdef clones) and more subject systems? In future work, we will focus on these questions to gain more insights on the relation between code clones and preprocessor annotations. Beyond that, we aim at providing a bigger picture of clones in the context of variable systems such as software product lines.

References

[1] B. W. Kernighan, *The C Programming Language*, 2nd ed., D. M. Ritchie, Ed. Prentice Hall, 1988.

[2] M. Ernst, G. Badros, and D. Notkin, "An empirical analysis of c preprocessor use," *IEEE Trans. Soft. Eng.*, vol. 28, no. 12, pp. 1146 – 1170, Dec. 2002.

[3] B. Stroustrup, *The Design and Evolution of C++*. New York, NY, USA: Addison-Wesley, 1995.

[4] J. Favre, "Understanding-in-the-large," in *Proc.of the Int. Workshop on Program Comprehension*. IEEE Computer Society, 1997, pp. 29–38.

[5] H. Spencer and G. Collyer, "#ifdef considered harmful, or portability experience with c news," in *Proc. USENIX Tech. Conf.* USENIX Association Berkeley, 1992, pp. 185–197.

[6] J. Liebig, C. Kästner, and S. Apel, "Analyzing the discipline of preprocessor annotations in 30 millions lines of c code," in *Proc. Int. Conf. on Aspect-Oriented Software Development*. ACM, 2011, to appear.

[7] C. Kästner, S. Apel, and M. Kuhlemann, "Granularity in software product lines," in *Proc. Int. Conf. on Software Engineering*. ACM Press, 2008, pp. 311–320.

[8] C. Kästner, S. Apel, S. Trujillo, M. Kuhlemann, and D. Batory, "Guaranteeing syntactic correctness for all product line variants: A language-independent approach," in *Proc. Int. Conf. on Objects, Models, Components and Patterns*. Springer-Verlag, 2009, pp. 175–194.

[9] C. Kästner and S. Apel, "Virtual separation of concerns – a second chance for preprocessors," *J. Obj. Tech. (JOT)*, vol. 8, no. 6, pp. 59–78, Sep. 2009, refereed Column.

[10] I. Baxter and M. Mehlich, "Preprocessor conditional removal by simple partial evaluation," in *Proc. of the Work. Conf. on Reverse Engineering*. IEEE Computer Society, 2001, pp. 281–291.

[11] B. S. Baker, "On Finding Duplication and Near-Duplication in Large Software Systems," in *Proc. of the Work. Conf. on Reverse Engineering*. IEEE Computer Society, 1995, pp. 86–95.

[12] I. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier, "Clone Detection Using Abstract Syntax Trees," in *Proc. of the Int. Conf. on Software Maintenance*. IEEE Computer Society, 1998, pp. 368–377.

[13] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, "DECKARD: Scalable and Accurate Tree-based Detection of Code Clones," in *Proc. Int. Conf. on Software Engineering*. IEEE Computer Society, 2007, pp. 96–105.

[14] E. Jürgens, F. Deissenboeck, B. Hummel, and S. Wagner, "Do code clones matter?" in *Proc. Int. Conf. on Software Engineering*. IEEE Computer Society, 2009, pp. 485–495.

[15] B. Laguë, D. Proulx, J. Mayrand, E. Merlo, and J. Hudepohl, "Assessing the benefits of incorporating function clone detection in a development process," in *Proc. of the Int. Conf. on Software Maintenance*. IEEE Computer Society, 1997, pp. 314–321.

[16] L. Aversano, L. Cerulo, and M. Di Penta, "How clones are maintained: An empirical study," in *Proc. of the Eur. Conf. on Software Maintenance and Reengineering*. IEEE Computer Society, 2007, pp. 81–90.

[17] M. Kim, V. Sazawal, D. Notkin, and G. Murphy, "An empirical study of code clone genealogies," in *Proc. Eur. Soft. Eng. Conf./Int. Symp. on Foundations of Soft. Eng. (ESEC/FSE)*. ACM, 2005, pp. 187–196.

[18] J. Mayrand, C. Leblanc, and E. Merlo, "Experiment on the automatic detection of function clones in a software system using metrics," in *Proc. of the Int. Conf. on Software Maintenance*. IEEE Computer Society, 1996, pp. 244–253.

[19] C. Roy and J. Cordy, "Near-miss function clones in open source software: An empirical study," *Journal of Software Maintenance*, vol. 22, no. 3, pp. 165–189, 2010.

[20] H. Basit and S. Jarzabek, "A data mining approach for detecting higher-level clones in software," *IEEE Trans. Soft. Eng.*, vol. 35, pp. 497–514, 2009.

[21] C. Roy and J. Cordy, "A Survey on Software Clone Detection Research," School of Computing, Queen's University at Kingston, Tech. Rep. 2007-451, 2007.

[22] S. Ducasse, M. Rieger, and S. Demeyer, "A language independent approach for detecting duplicated code," in *Proc. of the Int. Conf. on Software Maintenance*. IEEE Computer Society, 1999, pp. 109–118.

[23] T. Kamiya, S. Kusumoto, and K. Inoue, "CCFinder: A Multilinguistic Token-Based Code Clone Detection System for Large Scale Source Code," *IEEE Trans. Soft. Eng.*, vol. 28, no. 7, pp. 654–670, 2002.

[24] J. Krinke, "Identifying Similar Code with Program Dependence Graphs," in *Proc. of the Work. Conf. on Reverse Engineering*. IEEE Computer Society, 2001, pp. 301–309.

[25] E. Jürgens, F. Deissenboeck, and B. Hummel, "Clonedetective - a workbench for clone detection research," in *Proc. Int. Conf. on Software Engineering*. IEEE Computer Society, 2009, pp. 603–606.

[26] C. Prehofer, "Feature-Oriented Programming: A Fresh Look at Objects," in *Proc. of the Eur. Conf. on Object-Oriented Programming*. Springer, 1997, pp. 419–443.

[27] P. Clements and L. Northrop, *Software Product Lines: Practices and Patterns*. Addison Wesley, 2006.

[28] J. Maletic, M. Collard, and A. Marcus, "Source code files as structured documents," in *Proc.of the Int. Workshop on Program Comprehension*. IEEE Computer Society, 2002, pp. 289–292.

[29] T. Anderson and J. Finn, *The New Statistical Analysis of Data*. Springer, 1996.

[30] L. Giventer, *Statistical Analysis for Public Administration*, 2nd ed. Jones and Bartlett Publishing,

2008.
[31] J. Liebig, S. Apel, C. Lengauer, C. Kästner, and M. Schulze, “An analysis of the variability in forty preprocessor-based software product lines,” in *Proc. Int. Conf. on Software Engineering*. ACM, 2010, pp. 105–114.