Detecting Refactorable Clones Using PDG and Program Slicing

Ammar Hamid
10463593
ammarhamid84@gmail.com

August 17, 2014, 22 pages

Supervisor: Dr. Vadim Zaytsev
Host organisation: University of Amsterdam

Universiteit van Amsterdam
Faculteit der Natuurwetenschappen, Wiskunde en Informatica
Master Software Engineering
http://www.software-engineering-amsterdam.nl

Contents

Abstract

Code duplication in a program can make understanding and maintenance difficult. This problem can be reduced by detecting duplicated code, refactoring it into a separate new procedure, and replacing all the occurrences by calls to the new procedure. This study is an evaluation and extension of the paper of Komondoor and Horwitz, titled "Using Slicing to Identify Duplication in Source Code", which describes the approach and implementation of a tool, based on Program Dependence Graph (PDG) and Program Slicing. The tool can find non-contiguous (clones whose components do not occur as contiguous text in the program), reordered, intertwined, refactorable clones, and display them. PDG and Program Slicing provide an abstraction that ignores arbitrary sequencing choices made by programmer, and instead captures the most important dependencies (data and control flow) among program components.

Chapter 1

Introduction

Several studies [1, 10, 12] show that 7-23% of the source code for large programs is duplicated code. Within a project, code duplication will increase code size, and make maintenance difficult. Fixing a bug within a project with a lot of duplicated code is becoming a challenge because it is necessary to make sure that the fix is being done for all of the duplicated instances. Lague et al [12] studied the development of a large software system over multiple releases and found that programmers often missed some copies of the duplicated code when performing modification.

A tool that can find clones can help alleviate these problems: the clones identified by the tool can then be extracted into a new procedure, and the clone themselves replaced by calls to that procedure. This way, there will be only one copy to maintain, which is the new procedure only.

This study is a replication study of the code clone detection technique by Komondoor and Horwitz [9] in 2001 that describes the design and implementation of a tool that finds clones that can be refactored into a new procedure for C programs and displays them to the programmer. The novel approach of this work is the use of program dependence graphs (PDGs) [4], and program slicing [18, 14] to find isomorphic subgraphs of the PDG that represent clones. The benefits of this approach is that the tool can find clones of type-3 [16] where clones are non-contiguous (i.e., clones whose statements do not occur as contiguous text in the program), clones that have been reordered, and clones that are intertwined with each other. Moreover, the found clones should be refactorable into new procedures.

See Table 1.1 for an example of two procedures with some duplicated code. In the left column, the duplicated code is marked with ++ and in the right column those duplicated code is replaced with a new call to the newly extracted procedure in Table 1.2, indicated with **. In this example, it clearly shows that not everything that has the same structure or the same syntax is reported as clones (e.g. `int t = 10;`). The main reason that they are not included is because they do not have any predecessors that match between the two procedures (e.g. `int t = 10;` has no predecessor) and therefore can be left out safely. Only the clones which can be refactored into a new procedure are reported.

The structure of this replication study is as follows: chapter 2 presents briefly the original study by Komondoor and Horwitz [9], chapter 3 presents our replication study, chapter 4 presents results evaluation, chapter 5 presents future work, chapter 6 presents related studies, and finally chapter 7 presents our conclusion.

Procedure 1

```
 int foo(void) {
++   int i = 1;
     bool z = true;
     int t = 10;
++   int j = i + 1;
++   int n;
++   for (n=0; n<10; n++) {
++      j = j + 5;
     }
++   int k = i + j - 1;
     return k;
 }
```

Rewritten Procedure 1

```
 int foo(void) {
     bool w = false;
     int t = 10;
**   return new_procedure_bar();
 }
```

Procedure 2

```
 int bar(void) {
++   int i = 1;
     bool w = false;
     int t = 10;
++   int s;
++   int b = a + 1;
++   for (s=0; s<10; s++) {
++      b = b + 5;
     }
++   int c = a + b - 1;
     return c;
 }
```

Rewritten Procedure 2

```
 int bar(void) {
     bool w = false;
     int t = 10;
**   return new_procedure_bar();
 }
```

Table 1.1: Two procedures with duplicated code and the result after refactoring

Newly extracted procedure:

```
 int new_procedure_bar(void) {
++   int i = 1;
++   int j = i + 1;
++   int n;
++   for (n=0; n<10; n++) {
++      j = j + 5;
     }
++   int k = i + j - 1;
     return k;
 }
```

Table 1.2: The new extracted procedure from Table 1.1

Chapter 2

The Original Study

Komondoor and Horwitz [9] proposed the use of PDG and program slicing for clone detection. They represent each procedure using PDG representation. In PDG, nodes represent program statements and predicates, and edges represent data and control dependencies. PDG provides an abstraction that ignores arbitrary sequencing choices made by programmer, and instead captures the important dependences among program components.

On top of this representation, they use program slicing to query both data and control dependencies for a specific node or statement. There are two query types in program slicing, backward slicing and forward slicing. Backward slicing from node x means to find all the nodes that influence the value of node x. Forward slicing from node y means to find all the nodes that are influenced by node y. This is an important technique to filter out any statements that are irrelevant for clone detection.

2.1 Research questions

Can we find code clones of type-3 [16] (clones that are non-contiguous, reordered, and intertwined clones), which are refactorable into new procedures?

2.2 Algorithm description

The initial step is to find a set of pairs with syntactically equivalent node pairs. Two nodes are considered equivalent if they share equivalent AST structure. They performed backward slicing from each pair and a single forward slicing for matching predicates nodes. Finally, they removed subsumed clones and combined them into larger groups.

The key operation to find the isomorphic subgraphs (clone fragments) is the use of backward slicing, but they also use forward slicing to make sure that the clone they found, does not have cross loop. A clone with cross loop occurs when it has two nested-loops, one of the loop predicate matches and the other not. In section 3.4, we are going to explain in details (including example) why forward slicing would not be necessary to detect clones.

2.3 Implementation

The authors of the original paper used CodeSurfer [5] version 1.8 to generate PDG and write Scheme program that access CodeSurfer API to work with the generated PDGs and C implementation to do the processing of those PDG to find clones.

2.4 Summary of the results

They were able to find isomorphic subgraphs of the PDG by implementing a program slicing technique that uses a combination of backward slicing and forward slicing. They applied it to some open-source software written in C and they demonstrate the capability of slicing to detect non-contagious code clones. We will show more details on the results in chapter 4 where we will compare the original results with the replication results.

Chapter 3

Replication study

3.1 Motivation

The motivation of this replication study is to be able to validate algorithm and results of the original study. Once validated, we would like to publish our code and intermediate results into a public repository so that it is easier for any future researchers to either re-validate our results or to extend our program.

3.2 Algorithm description

To detect clones in a program, first we created a PDG representation for each procedures in the program. In PDG, node represents program statement or predicate, and edge represents data or control dependencies. The algorithm performs four steps (described in the following subsections):

Step 1: Find reachable procedures from the main program execution

Step 2: Find pair of expression-typed nodes with equivalent syntactic structure

Step 3: Find clones

Step 4: Group clones

3.2.1 Find reachable procedures from the main program execution

We want to detect clones for procedures that are reachable from the main program execution. The reason for this, is that we can safely remove unreachable procedures from our program as they are dead code and therefore there is no need to detect clones for it. In CodeSurfer [5] API, we do this by getting a system initialization node and doing a forward-slice with data and control flow. This will return all PDGs (including user defined procedures and system defined procedures) that are reachable from the main program execution. To have a sensible result that focuses on user defined procedures, we further filter those PDGs to find only the user defined ones, ignoring system libraries.

3.2.2 Find pairs of expression-typed nodes with equivalent syntactic structure

We scan all PDGs from the previous step to find nodes that have type `expression` in CodeSurfer (e.g. `int a = b + 1`). In CodeSurfer, node with `expression` type represents any assignment statements and any expressions. From those expression nodes, we try to match their syntactic structure with

each other (within the same procedure and also with the rest of the procedures). This is important so that we can find intertwined clones as well as non-contiguous clones. To find two expressions with equivalent syntactic structure, we make use of Abstract Syntax Tree(AST). This way, we ignore variable names, literal values, and focus only on the structure, e.g. `int a = b + 1` has syntactic structure that is equivalent to `int k = l + 5`, where both expression have the same type, which is `int`.

3.2.3 Find clones

From a pair of equivalent structure expressions, we do a backward slice call to find their predecessors and compare them among themselves. The main reason to find their predecessors is to make sure that the found clones are meaningful and it should be possible to move the found clones into a new procedure without changing their semantic. If the AST structures of their predecessors are the same then we store it in the collection of clones found. Because of this step, we can find non-contiguous, reordered, intertwined and refactorable clones. See example Table 3.1 for code fragments and Figure 3.1 for its PDGs representation.

Fragment 1

```
while (isalpha(c) || c == '_' || c == '-') {
++  if (p == token_buffer + maxtoken)
++      p = grow_token_buffer(p);
    if (c == '-') c = '_';
++  *p++ = c;
++  c = getc(finput);
}
```

Fragment 2

```
while (isdigit(c)) {
++  if(p == token_buffer + maxtoken)
++      p = grow_token_buffer(p);
    numval = numval*20 + c - '0';
++  *p++ = c;
++  c = getc(finput);
}
```

Table 3.1: Duplicated code from bison, taken from the orginal paper by Komondoor and Horwitz [9]

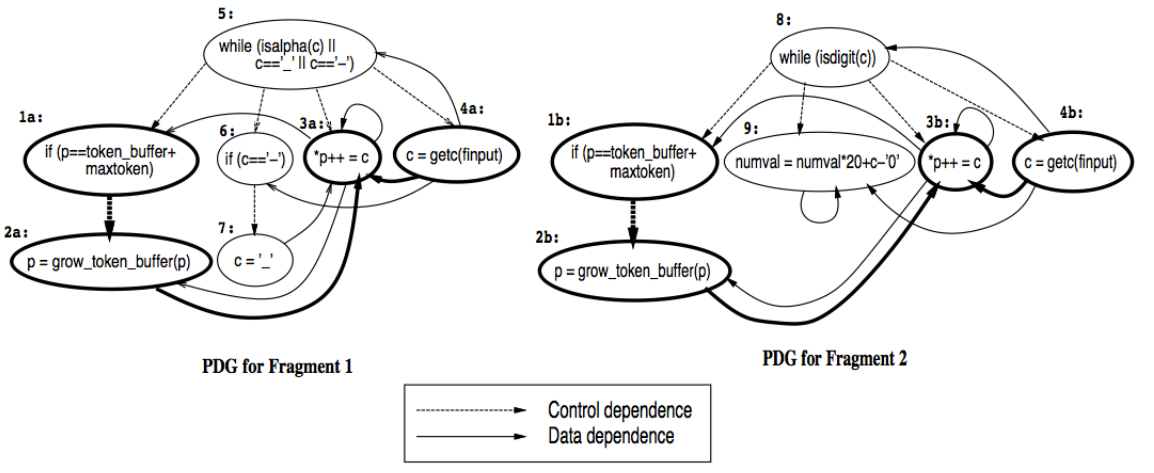

Figure 3.1: The nodes and edges in the partial slices are shown in bold. Taken from the original paper by Komondoor and Horwitz [9]

The key operation to find the isomorphic subgraphs (clone fragments) is the use of backward slicing from all matching expressions (based on their syntactic structure). For clarity purpose, we will give an example using Figure 3.1 where we will select node 3a and node 3b because they are having the same

syntactic structure. Slicing backward from nodes 3a and 3b along their incoming control-dependence edges we find node 5 and node 8 (the two while nodes). Since these nodes do not match because they have different syntactic structure (different AST structure), they are not added to the partial slices. Slicing backward from nodes 3a and 3b along their incoming data-dependence edges we find nodes 2a, 3a, 4a, and 7 in the first PDG, and nodes 2b, 3b, and 4b in the second PDG. Node 2a matches node 2b, and node 4a matches node 4b, so those nodes (and the edges just traversed to reach them) are added to the two partial slices. Slicing backward from node 2a and node 2b, we find node 1a and node 1b, which are a match, so they (and the traversed edges) are added. Slicing backward from nodes 4a and 4b, we find nodes 5 and 8, which do not match; the same two nodes are found when slicing backward from nodes 1a and 1b. The partial slices are now complete. The nodes and edges in the two partial slices are shown in Figure 3.1 using bold font. These two partial slices correspond to the clones of Fragments 1 and 2 shown in Table 3.1 using "++" signs.

3.2.4 Group clones

Grouping sub-clones into a larger group is very important to avoid confusion by showing subsumed clones, as an example, a clone pair (S1', S2') subsumes another clone pair (S1, S2) iff S1 $\subseteq$ S1' and S2 $\subseteq$ S2' [9], and therefore we only going to report (S1', S2') to the programmer.

Furthermore, in CodeSurfer, the node for a while-loop doesn't really show that it is a while loop but rather it is showing its predicate, e.g. `while(i<10)` will show as a control-point node `i<10`. Therefore, it is important that the found clones are mapped back to the actual program text representation so that programmer can understand and take action from the reported results.

3.3 Implementation

To implement the algorithm described in the previous sections, we use Scheme (536 LOC) and Ruby (161 LOC). We use Scheme to interact with the CodeSurfer API to find clones and output an intermediate raw result. After that, we use Ruby to further clean subsumed clones and group sub-clones into a larger group.

While it is possible to do everything in Scheme, we chose not to because in our experience and setup[1], Scheme is performing slow when it is faced with a huge collection of items that occupies at least 150 MB in memory. Therefore, we use Scheme only to interact with the CodeSurfer API and output an intermediate result that will be picked by Ruby for further processing.

To make sure that our code is usable and extendable, we keep a clear and clean distinction between each algorithm step in the code and the most important thing is, we code with modularity in mind. We also share our programs in GitHub[2] so that it is easier for future replication or extension to be done.

3.4 Changes to the original experiment

There are several important changes from the original paper that we need to mention, first of all, we only detect clones within the reachable procedures (excluding any unused procedures that are not reachable from main execution). This will make the result more accurate and at the same time we can clean all dead code.

Furthermore, we only use backward slicing and no forward slicing to detect clones. Before we explain why, lets have a look at the example in Table 3.2.

[1] Scheme interpreter that we use, is CodeSurfer's custom made and bundled together with CodeSurfer package
[2] https://github.com/ammarhamid/clone-detection

Fragment 1

```
   ...
** fp3 = lookaheadset + tokensetsize;
   for(i = lookaheads(state); i < k; i++) {
++     fp1 = LA + i * tokensetsize;
++     fp2 = lookaheadset;
++     while (fp2 < fp3) {
++         *fp2++ |= *fp1++;
       }
   }
```

Fragment 2

```
 ...
** fp3 = base + tokensetsize;
   ...
   if(rp) {
       while((j = *rp++) > 0) {
           ...
++         fp1 = base;
++         fp2 = F + j * tokensetsize;
++         while(fp1 < fp3) {
++             *fp1++ |= *fp2++;
           }
   }
```

Table 3.2: Two clones from bison that illustrates the necessity to have a forward slicing according to the original paper [9]

According to the original paper, only statements indicated by ++ will be reported as clones while statement marked with ** is excluded. The main argument according to the original paper is that `fp3` is used inside a loop but the loop predicate itself is not matching (for loop and the first while loop predicate doesn't match) – or a so called cross loop [9].

However, we argue that we should still report the statement marked with ** as clone and the fact that their loop predicate doesn't match, means we could still refactor this into two separate procedures, instead of a single procedure proposed by the original paper. Therefore we consider that forward slicing is only necessary to define refactoring strategy and not for detecting the clone itself.

Here is what it is look like after our refactoring (without forward slicing):

The new fragment 1

```
...
fp3 = location(lookaheadset, tokensetsize);
...
for(i = lookaheads(state); i < k; i++) {
    compute(LA, lookaheadset,
              i, tokensetsize, fp3);
}
```

The new fragment 2

```
...
fp3 = location(base, tokensetsize);
...
if(rp) {
    while((j = *rp++) > 0) {
        ...
       compute(F, base,
             j, tokensetsize, fp3);
    }
}
```

Table 3.3: The new fragments after refactoring (without forward slicing)

The extracted procedures

```
int location(int base, int size) {
    return base + size;
}

void compute(int cons, int base, int index, int size, int loc) {
    fp1 = cons + index * size;
    fp2 = base;
    while (fp2 < loc) {
        *fp2++ |= *fp1++;
    }
}
```

Table 3.4: The new refactored procedures. In this case, procedure location has only one statement which probably unnecessary to create a new procedure for it. But the point is if we use forward slicing in clone detection phase, we might hide this statement prematurely from the programmers, who at least should be aware of the situation before proceeding with refactoring.

Chapter 4

Results Evaluation

4.1 Comparison approach

To have a fair comparison on the results between the original and the replication study, we have two options: first one, we need to make sure that we have the original program and are able to run it; the second one, we need to re-implement the algorithm from scratch and find an approximate version of those C projects that are being used in 2001 by the original writer. We consider this is as an important step to avoid an unfair comparison that might affect the result and conclusion of this replication study.

Raghavan Komondoor, one of the authors of the original paper [9], was kind enough to provide us the sources of the original study. However, the sources are using a very old CodeSurfer (version 1.8), according to CodeSurfer support team. CodeSurfer support team can only provide us with the version 1.8 package and documentation, but they don't support licensing for it anymore and therefore this situation prevents us from running the original program.

As we still believe in the first option, we tried to port the original program to use the CodeSurfer version 2.3. However, this path poses many difficulties, one of the main problem with porting is that CodeSurfer version 1.8 and 2.3 use a different Scheme interpreter, which are STk and STklos. Secondly, understanding the original code is hard, because of the modularity (one file can have more than 4000 lines of code). We tried it anyway and spent about 2 weeks without any success signs.

Finally, after weighting the pros and cons of porting the original program, we decided to write a new implementation from scratch using CodeSurfer version 2.3. In this replication study, we can confirm that, writing the implementation from scratch is easier than porting the original program.

4.2 Comparison results

Having our own implementation, the next step is to find all the C open source projects version that are used in the original study, which are `tail`, `sort` (from Unix core utils), and `bison`. We are able to download the sources of those project from GNU git repositories[1]. To make sure that we have an approximately the same version of those projects, we downloaded versions that are released around 2001, which are CoreUtils version 4.5.2 and Bison version 1.29.

[1] http://git.savannah.gnu.org/cgit/

Study	Program	LOC	PDG nodes	Elapsed time		
				Scheme	C++	Ruby
Original	tail	1569	2580	40 sec.	3 sec.	NA
Replication	tail	1668	3052	5 sec.	NA	1 sec.
Original	sort	2445	5820	10 min.	7 sec.	NA
Replication	sort	2499	6891	30 sec.	NA	1 sec.
Original	bison	11540	28548	1 hour 33 min.	1 minute 5 sec.	NA
Replication	bison	10550	33820	2 hours 6 min.	NA	42 sec.

Table 4.1: Comparison on program size, number of nodes, implementation and elapsed time

Table 4.1 shows the comparison of the sizes of the three programs (in number of LOC and in number of nodes), and the running times for the algorithm between the original and replication study. Table 4.2 shows the comparison of the result in details between the original and replication study. Figure 4.1 shows a plot graph to understand better the pattern that exists in the result shown by Table 4.2.

Study	Program	Number of nodes in a clone							
		5-9	10-19	20-29	30-39	40-49	50-59	60-69	70+
Original	tail	21	4	0	0	4	1	0	2
Replication	tail	14	8	2	0	0	0	0	0
Original	sort	105	57	30	9	14	0	0	0
Replication	sort	52	27	11	1	0	1	0	0
Original	bison	513	164	34	16	9	9	6	49
Replication	bison	1545	638	201	15	14	6	0	1

Table 4.2: Detailed comparison results between the original and replication study

We detect less clones in `tail` and `sort` than the original study. We manually verified our results and they look sane. All of the reported clones are indeed refactorable into new procedures(s). We mentioned in the chapter 3, that we only detect clones from procedures that are reachable from the main execution. However, this does not explain why we detect less than the original study, since the number of unused procedures are 0 and 1 procedures for `tail` and `sort`, respectively.

In the case of `bison`, we detect more clones in the range of 5-29 nodes and much less clones in the range of 70+ nodes, we only detect one clone and the original study detects 49 group. We also manually verified our result for bison, though not all of them, but pick some randomly and we confirm that the clones found are refactorable into new procedures. At this point, we couldn't give any explanation why and what are the differences, since we don't have the original results in details.

In table 4.3, we demonstrated one of the results after running our implementation against `tail` program, between procedure `pipe_lines()` and `pipe_bytes()`. From the table, we show two clone fragments identified with `++` and `**`. This means that we will create two new procedures, one with the content of `++` and the other with `**`, and replace those statements with the two newly created procedures.

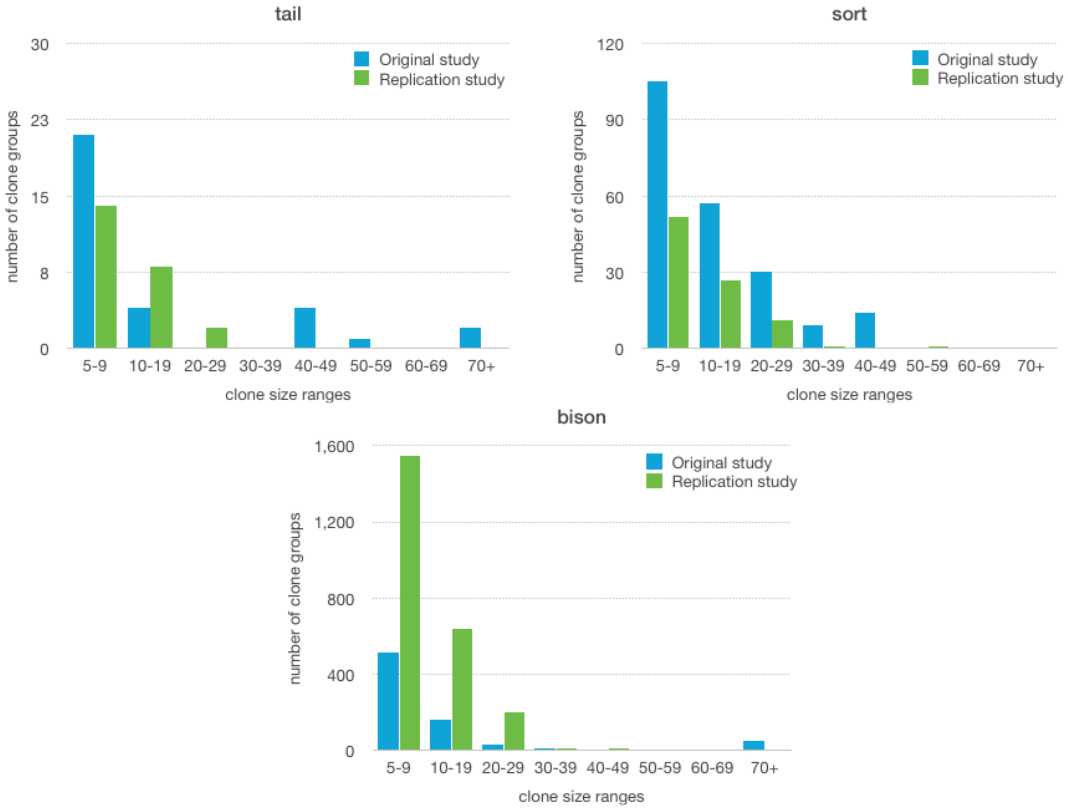

Figure 4.1: Plot representation of Table 4.2

4.3 Conclusion

Unfortunately, we could not explain why our results are different than the original study. The main reason that we cannot explain the differences, is because we don't have the detailed results of the original study. Moreover, based on our experiments and results, we are able to confirm that the given algorithm can detect clones of type-3, clones that are non-contagious, reordered, intertwined and clone that is refactorable into a new procedure.

After manual inspection of our results, we find out that the ideal number of node in a refactorable clone is 7. Less than this value, we observed that most of the clones are less valuable to be refactored to a new procedure, most but not all, it contains only initializations. This result amplifies the result of the original study.

However, we disagree on the need of forward slicing to detect refactorable clones. We found through experiments that this action is part of refactoring strategy and better performed when doing the refactoring (which is out of scope of the current study). That said, we believe that this action is prematurely applied to the original study.

What we also improved from this replication study is that we make sure that we will publish our programs, intermediate results, and this study report into a public repository, GitHub[2]. We believe that this contribution will make it is easier for any future researchers to either re-validate our results or to extend our program (e.g. as described in Chapter 5).

[2] https://github.com/ammarhamid/clone-detection

Table 4.3: Two procedures with duplicated code from tail.c version 4.5.12 - released in 2003

```
pipe_lines() {
   ...
++ while ((nbytes = tmp->nbytes =
              safe_read (fd, tmp->buffer, BUFSIZ)) > 0) {
++    tmp->next = NULL;
++    total_lines += tmp->nlines;
++    if (tmp->nbytes + last->nbytes < BUFSIZ) {
++        memcpy (&last->buffer[last->nbytes],
              tmp->buffer, tmp->nbytes);
++        last->nbytes += tmp->nbytes;
          ...
      }
++    else {
++        last = last->next = tmp;
++        if (total_lines - first->nlines > n_lines) {
++            tmp = first;
++            total_lines -= first->nlines;
++            first = first->next;
          }
++        else
++            tmp = (LBUFFER *) xmalloc (sizeof (LBUFFER));
      }
   ...
** for (tmp = first; total_lines - tmp->nlines > n_lines;
            tmp = tmp->next)
**    total_lines -= tmp->nlines;
   ...
** for (tmp = tmp->next; tmp; tmp = tmp->next)
**    xwrite (STDOUT_FILENO, tmp->buffer, tmp->nbytes);
** free_lbuffers:
**    while (first) {
**        tmp = first->next;
**        free ((char *) first);
**        first = tmp;
      }
   ...
}
```

```
pipe_bytes() {
   ...
++ while ((tmp->nbytes =
              safe_read (fd, tmp->buffer, BUFSIZ)) > 0) {
++    tmp->next = NULL;
++    total_bytes += tmp->nbytes;
++    if (tmp->nbytes + last->nbytes < BUFSIZ) {
++        memcpy (&last->buffer[last->nbytes],
             tmp->buffer, tmp->nbytes);
++        last->nbytes += tmp->nbytes;
      }

++ else {
++        last = last->next = tmp;
++        if (total_bytes - first->nbytes > n_bytes) {
++            tmp = first;
++            total_bytes -= first->nbytes;
++            first = first->next;
          }
++        else
++            tmp = (CBUFFER *) xmalloc (sizeof (CBUFFER));
      }
   ...
** for (tmp = first; total_bytes - tmp->nbytes > n_bytes;
           tmp = tmp->next)
**    total_bytes -= tmp->nbytes;
   ...
** for (tmp = tmp->next; tmp; tmp = tmp->next)
**    xwrite (STDOUT_FILENO, tmp->buffer, tmp->nbytes);
** free_cbuffers:
**    while (first) {
**        tmp = first->next;
**        free ((char *) first);
**        first = tmp;
      }
   ...
}
```

Chapter 5

Future Work

This replication study presents an algorithm to detect clones of type-3 [16] using program dependence graphs [14, 4] and program slicing [18, 3]. However, this algorithm limits its scope in finding clones in a single procedure.

Our suggestion for future work is to extend this algorithm so that it is able to detect some type-4 [16] clones where it is broadly defined as "code fragments that perform the same computation but are implemented by different syntactic variants" [16]. As an example of this type-4 clone are inline variable and extract procedure.

Figure 5.1 shows a possible scenario, where with type-3 clone detection tool such as this replication study will detect clones that are identified by red line, while type-4 clone detection is identified by blue lines. It clearly shows that the scope of type-4 clone detection tool is not a single procedure but rather interprocedural.

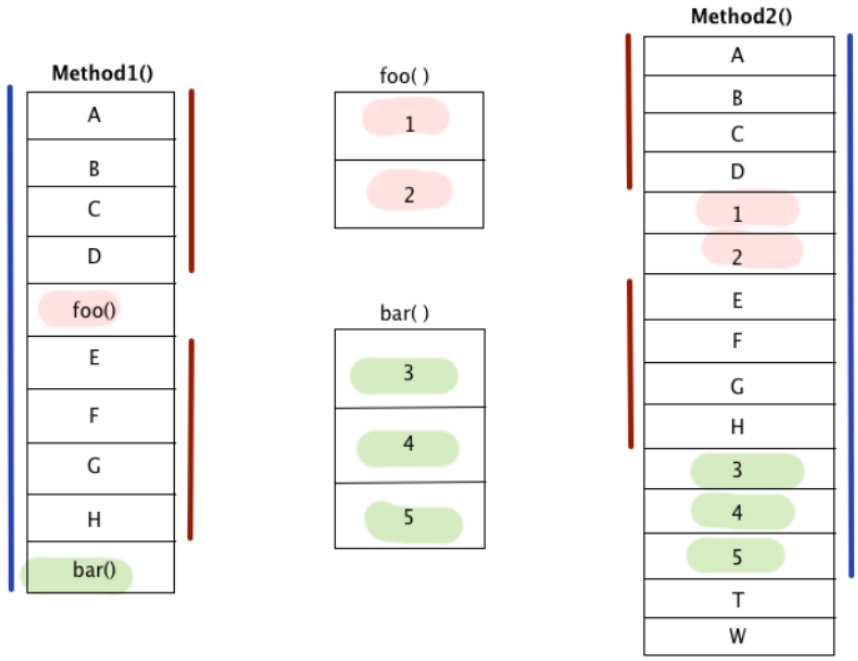

Figure 5.1: Type-4 clone detection should result in a maximal clone fragment as indicated by the blue lines

Figure 5.2 shows how the result looks like after refactoring `method1` and `method2`. In this example,

type-4 clone detection tool will introduce only a single procedure, called `newMethod`. In contrast to type-3 clone detection tool, which will introduce two new smaller procedures (which are indicated by the red line in Figure 5.1).

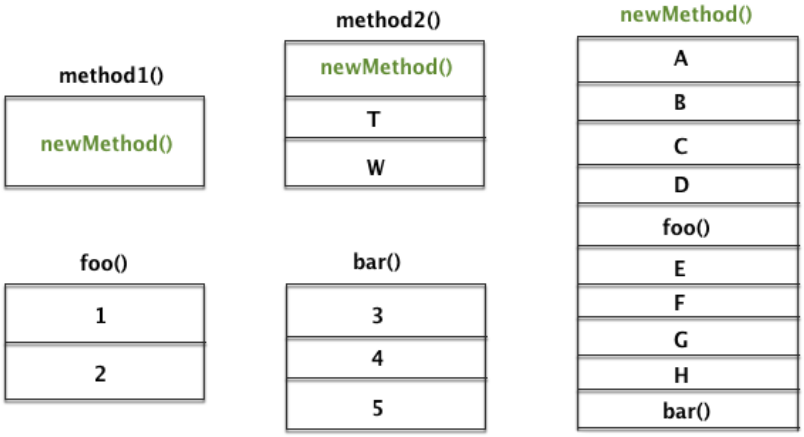

Figure 5.2: The result after refactoring method1() and method2()

One of the interesting research questions would be, are detecting the clones into one big procedure (blue line, in Figure 5.1) preferable compared to the two smaller procedures (red line, in Figure 5.1) in this example? This is highly related to the cohesiveness of the refactored clones.

Chapter 6

Related Studies

Shomrat and Feldman [17] present Cider, an algorithm that can detect code clones semantically and regardless of various refactorings that have been applied to the clones. The various refactorings include extracting and inlining variable and procedure. This is better known as type-4 [16] clone.

Their approach is to use plan calculus [15] to represent programs. The plan calculus represents each expression in the program as a node (instead of each statement as a node) and therefore the representation is insensitive to names of local variables or procedures. The good thing about this approach is its ability to find interprocedural clones but it does not guarantee that it can be refactored into a useful procedure that can be re-used in the program. This algorithm is more appropriate for plagiarism detection tool.

Baxter [2] presents clone detection tool using Abstract Syntax Tree (AST). It focuses on type-1 [16] clones that are due to copy-and-paste activity. The consequence is that the tool is only capable of finding exact clones syntactically rather than semantically. The algorithm to detect clones is by comparing ASTs, that are generated from the parsed source code, and finding maximum subtree clones. In this paper, the authors did not carry out clone removal.

Jiang [7] presents Deckard, an efficient algorithm for identifying similar subtrees and apply it to tree representations of source code. The algorithm is based on characterization of subtrees with numerical vectors in the Euclidean space to capture structural information of trees and an efficient hashing and near-neighbor querying algorithm for numerical vectors. The clone detection approach is similar to the clone detection using AST by Baxter [2]. However, this paper is using an efficient algorithm that is both scalable and accurate for large code base.

Krinke [11] presents Duplix, an algorithm for clone detection using fine-grained program dependence graph. This approach is similar to the approach using plan calculus [15, 17] where expression is used as a computational node (instead of statement as a node [9]).

Liu [13] presents GPLAG, which detects program plagiarism based on the analysis of program dependence graphs. **Kamiya** [8] presents CCFinderX, which is a code clone detection tool that based on AST-based preprocessing. This tool completely redesigned the previous tool CCFinder [8].

Chapter 7

Conclusion

This study is a replication study of the code clone detection technique by Komondoor and Horwitz [9] in 2001 that describes the design and implementation of a tool that finds clones that can be refactored into a new procedure for C programs and displays them to the programmer. The novel approach of this work is the use of program dependence graphs (PDGs) [4], and program slicing [18, 14] to find isomorphic subgraphs of the PDG that represent clones. The benefits of this approach is that the tool can find clones of type-3 [16] where clones are non-contiguous (i.e., clones whose statements do not occur as contiguous text in the program), clones that have been reordered, and clones that are intertwined with each other. Moreover, the found clones should be refactorable into new procedures. More details on the background and research questions can be found in chapter 1 and 2.

The motivation of this replication study is to be able to validate algorithm and results of the original study [9]. Once validated, we published our code and intermediate results into a public repository, GitHub[1]. We believe that this contribution will make it is easier for any future researchers to either re-validate our results or to extend our program. More details on the replication study can be found in chapter 3.

Before finishing our replication study, we presented our extended abstract [6] and early results in a software evolution seminar, SATToSE[2] 2014, in the University of L'Aquila, Italy. We would like to thank all of the participants of SATToSE 2014 who kindly gave their feedback to our extended abstract and early results, which become valuable inputs to finalize this replication study. At the time of writing this study, the coordinators of SATToSE are working on a post-proceedings, which will be published soon.

We have successfully replicated the algorithm and confirmed it shows the clones of the promised variety, but we could not explain the numerical differences in our results. The main reason that we cannot explain the differences, is because we don't have the detailed results of the original study. Moreover, based on our experiments and results, we are able to confirm that the given algorithm can detect clones of type-3, clones that are non-contagious, reordered, intertwined and clone that is refactorable into a new procedure. More details on the comparison results between the original and the replication study can be found in chapter 4.

For future work, we would like to suggest to extend this algorithm so that it is able to detect some type-4 [16] clones where it is broadly defined as "code fragments that perform the same computation but are implemented by different syntactic variants" [16]. As an example of this type-4 clone are inline variable and extract procedure. More details on our suggestion for future work can be found in chapter 5. For related studies, please consult chapter 6 for more details.

[1] https://github.com/ammarhamid/clone-detection
[2] http://sattose.org/2014

Finally, we would like to thank Raghavan Komondoor, one of the authors of the original paper [9], who was kind enough to share the original program sources for the purpose of this replication study. We also would like to thank David Vitek and CodeSurfer [5] who have granted us an academic license for CodeSurfer version 2.3. Last but not least, we would like to thank our supervisor, Vadim Zaytsev, for his endless and valuable guidance and feedback, that allows us to complete this replication study.

Bibliography

[1] B. S. Baker. On finding duplication and near-duplication in large software systems. In *WCRE*, pages 86–95, 1995.

[2] I. D. Baxter, A. Yahin, L. M. de Moura, M. Sant'Anna, and L. Bier. Clone detection using abstract syntax trees. In *ICSM*, pages 368–377, 1998.

[3] J. Beck and D. Eichmann. Program and Interface Slicing for Reverse Engineering. In *Proceedings of the 15th International Conference on Software Engineering*, ICSE '93, pages 509–518. IEEE Computer Society Press, 1993.

[4] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The Program Dependence Graph and Its Use in Optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 9(3):319–349, July 1987.

[5] GrammaTech. CodeSurfer. http://www.grammatech.com/research/technologies/codesurfer.

[6] A. Hamid. Detecting Refactorable Clones Using PDG and Program Slicing. pages 56–59. Dipartimento di Informatica Università degli Studi dell'Aquila, L'Aquila, Italy, July 2014. Extended Abstract.

[7] L. Jiang, G. Misherghi, Z. Su, and S. Glondu. Deckard: Scalable and accurate tree-based detection of code clones. In *ICSE*, pages 96–105. IEEE Computer Society, 2007.

[8] T. Kamiya, S. Kusumoto, and K. Inoue. Ccfinder: A multilinguistic token-based code clone detection system for large scale source code. *IEEE Trans. Software Eng.*, 28(7):654–670, 2002.

[9] R. Komondoor and S. Horwitz. Using Slicing to Identify Duplication in Source Code. In *Proceedings of the 8th International Symposium on Static Analysis*, SAS '01, pages 40–56. Springer-Verlag, 2001.

[10] K. Kontogiannis, R. de Mori, E. Merlo, M. Galler, and M. Bernstein. Pattern matching for clone and concept detection. *Autom. Softw. Eng.*, 3(1/2):77–108, 1996.

[11] J. Krinke. Identifying similar code with program dependence graphs. In *WCRE*, pages 301–309, 2001.

[12] B. Laguë, D. Proulx, J. Mayrand, E. Merlo, and J. P. Hudepohl. Assessing the benefits of incorporating function clone detection in a development process. In *ICSM*, pages 314–321, 1997.

[13] C. Liu, C. Chen, J. Han, and P. S. Yu. Gplag: detection of software plagiarism by program dependence graph analysis. In T. Eliassi-Rad, L. H. Ungar, M. Craven, and D. Gunopulos, editors, *KDD*, pages 872–881. ACM, 2006.

[14] K. J. Ottenstein and L. M. Ottenstein. The Program Dependence Graph in a Software Development Environment. In *Proceedings of the First ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, SDE 1, pages 177–184. ACM, 1984.

[15] C. Rich and R. C. Waters. *The programmer's apprentice.* ACM Press frontier series. ACM, 1990.

[16] C. K. Roy, J. R. Cordy, and R. Koschke. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Sci. Comput. Program.*, 74(7):470–495, 2009.

[17] M. Shomrat and Y. A. Feldman. Detecting Refactored Clones. In G. Castagna, editor, *Proceedings of the 27th European Conference on Object-Oriented Programming (ECOOP)*, volume 7920 of *Lecture Notes in Computer Science*, pages 502–526. Springer, 2013.

[18] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, 1984.