Detecting Refactorable Clones Using PDG and Program Slicing

Extended Abstract

Ammar Hamid*

Universiteit van Amsterdam

Abstract.

Code duplication in a program can make understanding and maintenance more 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 paper is an evaluation and extension of the paper of Komondoor and Horwitz [3], which describes the approach and implementation of a tool, that based on Program Dependence Graph (PDG) [4] and Program Slicing [7]. 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 dependences (data and control flow) among program components. In contrast to some approaches that used the program text, control-flow graph, and AST, all of which are more closely tied to the lexical structure which is sometimes producing irrelevant result.

1 Approach

To detect clones in a program, we represent each procedure using its PDG. In PDG, vertex represents program statement or predicate, and edge represents data or control dependences. The algorithm performs four steps (described in the following subsections):

Step 1: Find relevant procedures
Step 2: Find pair of vertices with equivalent syntactic structure
Step 3: Find clones
Step 4: Group clones

1.1 Find relevant procedures

We are only interested in finding 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 and therefore there is no need to detect clones for it. We do this by getting a system initialization vertex and do a forward-slice with data and control flow. This will return all PDGs (including user defined and system PDGs) that are reachable from the main program execution. From that result, we further filter those PDGs to find only the user defined ones, ignoring system libraries.

* email: ammarhamid84@gmail.com

1.2 Find pair of vertices with equivalent syntactic structure

We scan all PDGs from the previous step to find vertices that has type `expression` (e.g. int a = b + 1). From those expression vertices, we try to match their syntactic structure with each other. To find two expressions with equivalent syntactic structures, 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` is equivalent with `int k = l + 1`, where both expression has the same type, which is `int`).

1.3 Find clones

From a pair of equivalent structure expressions, we do a backslice call to find their predecessors and compare them with each other. 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. Refactorable clones in this case mean that the found clones are meaningful and it should be possible to move it into a new procedure without changing their semantic.

1.4 Group clones

This is the step where we make sense of the found clones before displaying them. As an example, using CodeSurfer [2], the vertex for a while-loop doesn't really show that it is a while loop but rather showing its predicate, e.g. `while(i<10)` will show as a control-point vertex `i<10`. Therefore, it is important that the found clones is mapped back to the actual program text representation and group them together before displaying them. Moreover, it is important that the programmer can understand and take action on the reported clones.

2 Evaluation

We are using CodeSurfer (version 2.3) to create PDG representation for the C-program to be analyzed. In CodeSurfer, we use the API, written in Scheme [1], to access those PDGs and perform all operations in the previous sections programmatically. The progress so far, with running this approach on a sample program, shows that it is pretty accurate and satisfying result. See example below:

Procedure 1	Procedure 2
`int foo(void) {`	`int bar(void) {`
`*  int i = 1;`	`*  int a = 1;`
`  bool z = true;`	`  bool w = false;`
`  int t = 10;`	`  int t = 10;`
`*  int j = i + 1;`	`*  int s;`
`*  int n;`	`*  int b = a + 1;`
`*  for (n=0; n<10; n++) {`	`*  for (s=0; s<10; s++) {`
`*  j = j + 5;`	`*  b = b + 5;`
`  }`	`  }`
`*  int k = i + j - 1;`	`*  int c = a + b - 1;`
`  return k;`	`  return c;`
`}`	`}`

The clones found are indicated with *. In this example, it clearly shows that not everything that has the same structure or the same syntax are reported as clones (e.g. `int t = 10;`). The reason that some vertices with equivalent syntactic structures are not included is that because they are not used anywhere and therefore can be left out safely. Only the clones which are meaningful and can be refactored into a new procedure are reported.

3 Related studies

Komondoor and Horwitz [3] proposed the use of PDG for clone detection. They were able to find isomorphic subgraphs of the PDG by implementing a program slicing technique that is using a combination of backward slicing and forward slicing. Their initial step is to find set of pair with syntactically equivalent node pairs and performed backward slicing from each pair with a single forward slicing for matching predicates nodes.

Cider [6] can detect an interprocedural clone, using Plan Calculus [5]. This algorithm can detect code clones regardless of various refactorings that may have been applied to some of the copies but not to others.

References

[1] H. Abelson, R. K. Dybvig, C. T. Haynes, G. J. Rozas, I. Adams, N. I., D. P. Friedman, E. Kohlbecker, J. Steele, G. L., D. H. Bartley, R. Halstead, D. Oxley, G. J. Sussman, G. Brooks, C. Hanson, K. M. Pitman, and M. Wand. Revised[5] Report on the Algorithmic Language Scheme. *Higher-Order and Symbolic Computation*, 11(1):7–105, 1998.

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

[3] 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.

[4] 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.

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

[6] 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.

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