Tool Demonstration: Finding Duplicated Code Using Program Dependences

Raghavan Komondoor and Susan Horwitz
{raghavan, horwitz}@cs.wisc.edu
Computer Sciences Department, University of Wisconsin-Madison

1 Introduction

The results of several studies [1, 7, 8] indicate that 7–23% of the source code for large programs is duplicated code. Duplication makes programs harder to maintain because when enhancements or bug fixes are made in one instance of the duplicated code, it is necessary to search for the other instances in order to perform the corresponding modification.

A tool that finds clones (instances of duplicated code) can help to alleviate this problem. When code is modified, the tool can be used to find the other copies that also need modification. Alternatively, the clones identified by the tool can be extracted into a new procedure, and the clones themselves replaced by calls to that procedure. In that case, there is only one copy to maintain (the new procedure), and the fact that the procedure can be reused may cut down on future duplication.

We have designed and implemented a tool for C programs that finds clones and displays them to the programmer. To find clones in a program, we represent each procedure using its program dependence graph (PDG) [6]. In the PDG, nodes represent program statements and predicates, and edges represent data and control dependences. To find a pair of clones we use a variation on *backward slicing* [11, 10]. We start with two matching nodes (nodes that represent statements or predicates with matching syntactic structure, ignoring variable names and literal values), and we then slice backwards from those nodes in lock step, including a predecessor (and the connecting edge) in one slice iff there is a corresponding, matching predecessor in the other slice. Thus, when the process finishes, we will have identified two isomorphic subgraphs (two partial backward slices), that represent two clones. Pairs of clones are then combined into groups using a kind of transitive closure. For example, clone pairs $(S1, S2)$, $(S1, S3)$, and $(S2, S3)$ would be combined into the clone group $(S1, S2, S3)$.

The key benefits of a slicing-based approach, compared with previous approaches to clone detection such as [1, 2, 7, 4, 9, 3, 5], is that our tool can find non-contiguous clones (i.e., clones whose statements do not occur as contiguous text in the program), clones in which matching statements have been reordered, and clones that are intertwined with each other. (Our tool can also find clones in which variables have been renamed, and different literal values have been used; however, this is also true of some previous clone-detection algorithms.) Furthermore, the clones found using slicing are likely to be meaningful computations, and thus good candidates for procedural extraction. These benefits arise mainly because slicing is based on the PDG, which provides an abstraction that ignores arbitrary sequencing choices made by the programmer, and instead captures the important dependences among program components. In contrast, most previous approaches to clone detection use the program text, its control-flow graph, or its abstract-syntax tree, all of which are more closely tied to the (sometimes irrelevant) lexical structure.

2 Example Clones Found by Our Tool

Non-contiguous clones: Shown below are three fragments of code from the Unix utility *bison* that contain a group of three clones identified by our tool. The clones (which were found by slicing back from the statement "`*p++ = c;`") are indicated by "++" signs. The function of the clones is to grow the buffer pointed to by `p` if needed, append the current character `c` to the buffer and then read the next character. The clone in Fragment 3 is contiguous, but the corresponding clones in Fragments 1 and 2 are non-contiguous.

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);
   }
```

Fragment 3:

```
   while (c != '>') {
      if (c == EOF) fatal();
      if (c == '\n') {
          warn("unterminated type name");
          ungetc(c, finput);
          break;
      }
++    if (p == token_buffer + maxtoken)
++        p = grow_token_buffer(p);
++    *p++ = c;
++    c = getc(finput);
   }
```

Although two of the three clones are non-contiguous, they can still be extracted into a procedure: in both cases the statement in the middle of the clone can be moved out of the way without affecting semantics. We have observed that non-contiguous clones that are good candidates for extraction (as in the example above) occur frequently in real programs. Therefore, the fact that our approach can find such clones is a significant advantage over most previous approaches to clone detection.

Reordered clones: Non-contiguous clones are a kind of *near* duplication. Another kind of near duplication occurs when the ordering of matching nodes is different in the different clones, as illustrated below by two code fragments from *bison*. Both clones modify a portion of a bit array (`lookaheadset` / `base`) by performing a bit-wise *or* with the contents of another array (`LA` / `F`). The clones are identified by slicing back from the statement inside the `while` loop. Fragment 2 differs from Fragment 1 in two ways: the variables have been renamed (including renaming `fp1` to `fp2` and vice versa), and the order of the first and second lines has been reversed. Since this renaming and reordering does not affect the data or control dependences, the first and second statements of Fragment 1 are matched with the second and first statements of Fragment 2 by our approach (and the two `while` loops are matched with each other).

Fragment 1:

```
++ fp1 = LA + i * tokensetsize;
++ fp2 = lookaheadset;
++ while (fp2 < fp3)
++     *fp2++ |= *fp1++;
```

Fragment 2:

```
++ fp1 = base;
++ fp2 = F + j * tokensetsize;
++ while (fp1 < fp3)
++     *fp1++ |= *fp2++;
```

Intertwined clones: The use of backward slicing is also effective in finding intertwined clones. An example from the Unix utility *sort* is given below. In this example, one clone is indicated by "++" signs while the other clone is indicated by "xx" signs. The clones take a character pointer (`a`/`b`) and advance the pointer past all blank characters, also setting a temporary variable (`tmp1/tmpb`) to point to the first non-blank character. The final component of each clone is an `if` predicate that uses the temporary. Those predicates were the roots of the slices used to find the two clones (the second one – the last line shown below – occurs 43 lines further down in the code).

```
++ tmpa = UCHAR(*a),
xx tmpb = UCHAR(*b);
++ while (blanks[tmpa])
++   tmpa = UCHAR(*++a);
xx while (blanks[tmpb])
xx   tmpb = UCHAR(*++b);
++ if (tmpa == '-')
   ...
xx else if (tmpb == '-') ...
```

3 Experimental Results

We have performed several studies in which we compared the output of our tool to the clone groups identified by a programmer. We found that the tool is quite effective at identifying interesting clones; in particular, it found at least one clone group for every human-identified group of clones. Furthermore, many of the clones were non-contiguous and involved variable renaming; some also involved reordering and intertwining; this is an indication that our approach is capable of finding interesting clones that would be missed by other approaches.

A limitation of the tool is that the clones that it finds are often not "ideal", i.e., are not exactly those that a human would have identified. Instead, they contain some extra statements, or fail to contain some statements that should be part of the clones. Because of this, the tool often identifies several overlapping groups of clones that are variants of each other rather than identifying a single group of ideal clones (i.e., the clones in each group found by the tool have many statements in common with the clones in the other groups, but are not exactly the same). For example, in one study the programmer identified 4 ideal clone groups in one file, while the tool identified 17 clone groups that were variations on the 4 ideal groups. Future work includes finding heuristics that will help the tool to identify clones that are as close as possible to ideal.

Acknowledgements

This work was supported in part by the National Science Foundation under grants CCR-9970707 and CCR-9987435, and by IBM.

References

1. B. Baker. On finding duplication and near-duplication in large software systems. In *Proc. IEEE Working Conf. on Reverse Eng.*, pages 86–95, July 1995.
2. B. Baker. Parameterized duplication in strings: Algorithms and an application to software maint. *SIAM Jrnl. of Computing*, 26(5):1343–1362, Oct. 1997.
3. I. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. Clone detection using abstract syntax trees. In *Int. Conf. on Software Maint.*, pages 368–378, 1998.
4. N. Davey, P. Barson, S. Field, R. Frank, and D. Tansley. The development of a software clone detector. *Int. Jrnl. of Applied Software Tech.*, 1(3-4):219–36, 1995.
5. S. Debray, W. Evans, R. Muth, and B. D. Sutter. Compiler techniques for code compaction. *ACM Trans. on Prog. Lang. and Sys.*, 22(2):378–415, Mar. 2000.
6. J. Ferrante, K. Ottenstein, and J. Warren. The program dependence graph and its use in optimization. *ACM Trans. on Prog. Lang. and Sys.*, 9(3):319–349, July 1987.
7. K. Kontogiannis, R. Demori, E. Merlo, M. Galler, and M. Bernstein. Pattern matching for clone and concept detection. *Automated Software Eng.*, 3(1–2):77–108, 1996.
8. B. Lague, D. Proulx, J. Mayrand, E. Merlo, and J. Hudepohl. Assessing the benefits of incorporating function clone detection in a development process. In *Int. Conf. on Software Maint.*, pages 314–321, 1997.
9. 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 Maint.*, pages 244–254, 1996.
10. K. Ottenstein and L. Ottenstein. The program dependence graph in a software development environment. In *Proc. ACM SIGSOFT/SIGPLAN Software Eng. Symp. on Practical Software Development Environments*, pages 177–184, 1984.
11. M. Weiser. Program slicing. *IEEE Trans. on Software Eng.*, SE-10(4):352–357, July 1984.