Eliminating Duplication in Source Code via Procedure Extraction

Raghavan Komondoor
raghavan@cs.wisc.edu

Susan Horwitz
horwitz@cs.wisc.edu

Computer Sciences Department
University of Wisconsin-Madison
1210 W. Dayton St, Madison, WI 53706 USA

ABSTRACT

Duplication in source code is a widespread phenomenon that increases program size and complexity, and makes program maintenance more difficult. A solution to this problem is to detect clones (instances of copied code) and to eliminate them. Elimination works by extracting the cloned code into a separate new procedure, and replacing each clone by a call to this procedure. Several automatic approaches to detecting clones have been reported in the literature. In this paper we address the issue of automatically extracting a previously detected group of clones into a separate procedure. We present an algorithm that can extract "difficult" groups of clones, and a study that shows that difficult clone groups arise frequently in practice, and that our algorithm handles them well.

1. INTRODUCTION

Programs undergoing ongoing development and maintenance often have a lot of duplicated code. The results of several studies [1, 13, 16] indicate that 7–23% of the source code for large programs is duplicated code. Duplication is usually caused by copy and paste activities: a new feature that resembles an existing feature is implemented by copying and pasting code fragments, perhaps followed by some modifications. Duplication increases code size and complexity, and makes program maintenance more difficult. For example, if an enhancement or bug fix is done on one clone (an instance of duplicated code), it may be necessary to search for the other clones in order to perform the corresponding modification. In large legacy programs it is likely that programmers will miss some clones while doing modifications [16].

The maintainability of such legacy programs can be improved by detecting and eliminating clones. Elimination works by extracting the cloned code into a separate new procedure, and replacing each clone by a call to this procedure. In that case, there will be only one copy to maintain (the new procedure), and the fact that the procedure can be reused may cut down on future duplication. (Note that for a language like C with a preprocessor, macros can be used instead of procedures if there is a concern that introducing procedures will result in unacceptable performance degradation.) Several automatic approaches to detecting duplication have been reported in the literature [1, 13, 5, 8, 11]. In this paper we address the issue of automatic extraction of a previously detected group of clones into a separate procedure.

1.1 Main contributions

This paper has two main contributions:

1. An algorithm for extracting "difficult" groups of clones: groups in which statements within an individual clone are not contiguous, groups in which clones have matching statements in different orders, and groups in which clones involve *exiting jumps* (jumps from within the region containing the clone to outside that region). Such clone groups are hard to extract manually; therefore, tool support for this task is desirable. Our algorithm is a significant improvement over previously defined automatic approaches to clone-group extraction; because those approaches employ a narrow range of techniques, they often perform poorly on difficult groups of clones.

2. A study of 56 clone groups from 3 programs to determine how often difficult clone groups occur in practice, and how well our algorithm performs compared with the best previous clone-group extraction algorithm and with the results that would be produced by a human programmer. We found that nearly half of the clone groups exhibited at least one problematic aspect. We also found that our algorithm produced exactly the same output as the programmer on 66% of the difficult groups, while the previous algorithm matched that "ideal" output on fewer than 8% of the groups.

Extracting a difficult group of clones is a two-step process. The first step involves dealing with the difficult aspects by applying semantics-preserving transformations to the clones to make them easily extractable into a new procedure. The second step involves extracting the new procedure and replacing the (transformed) clones by calls to the procedure. The focus of this paper is the first step. We do not address the second step; the main issue in that step is to determine what local variables and parameters the new procedure needs; this has been addressed in previous clone-group extraction approaches [18, 8] (however, those approaches work on assembly code, and therefore need to be extended to work on source-level languages.)

1.2 Algorithm overview

We provide an overview of our algorithm using the motivating example in Figure 1, which shows a difficult group of two clones. The first column of Figure 1 shows two fragments of code, with the two clones indicated by the "++" signs. Both fragments have a loop that iterates over all employees. The cloned code reads in the number of hours worked from a file and uses that to compute the overtime pay and total pay for the current employee. The overtime pay rate and base pay for the employee are obtained from the arrays `OvRate` and `BasePay` respectively, while the computed total pay is saved in the array `Pay` (these three arrays are global variables, as is the file pointer `fh`). This clone group is difficult to extract because:

1. Intervening non-cloned statements are present in both fragments (the statement "`nOver++`" in the first fragment, the update of `totHours` in the second fragment, and also the adjustment of the variable `excess`, in the second fragment, when its value is greater than 10). In other words, both clones are non-contiguous.

2. Matching statements are in different orders in the two clones: the position of "`base = BasePay[emp]`" differs in the two cases, and so does the relative ordering of the two statements "`oRate = OvRate[emp]`" and "`excess = hours - 40`". We call such groups *out-of-order* groups.

3. Both clones contain exiting jumps (the `break` statements).

The middle column of Figure 1 shows the result of applying our algorithm; the regions that originally contained the clones have been transformed to make the group *extractable* (i.e., ready to be extracted). The final code after actual extraction (which is not done by our algorithm) is shown in the right column of Figure 1.

The algorithm runs in two phases. The first phase consists of applying the single-fragment algorithm described in [12] *individually* on each clone to make it extractable (into its own procedure). In this phase, as many intervening non-cloned statements as possible are moved out of the way, to either before or after the cloned statements, while preserving semantics. Such movement is accompanied by duplication of predicates and jumps, where necessary, to preserve semantics. If a non-cloned statement cannot be moved while preserving semantics, it is *promoted*, which means that it will be present in the extracted procedure guarded by a flag that is a parameter of the procedure.

Figure 2 shows the intermediate result after Phase 1 has been applied to the two fragments in Figure 1 (the loops enclosing the clones are omitted from the figure). Notice that the promoted code in the second fragment is indicated by "`****`" signs. In the example, "`nOver++`" in the first fragment and "`totHours += hours`" in the second fragment are moved past the last statements in the respective clones, with duplication (of two predicates and a jump in the first fragment, and one predicate and a jump in the second). The non-cloned statement "`if (excess>10) excess=10`" cannot be moved out of the way without affecting data dependences. Therefore it is promoted.

The first phase also deals with exiting jumps. Notice that in the example, since the surrounding loop is not included in the procedure, the `break` statement would not make sense if included unchanged in the extracted procedure. Instead, the extracted procedure should have a `return` statement in place of the `break`; this works because the new copy of the `break` after the cloned code will execute after control returns through this `return` statement. Therefore, the algorithm of [12] converts the two original copies of the `breaks` into `gotos` whose targets are the points just after the cloned code (as shown in Figure 2). At the time of actual extraction, these `gotos` are simply converted into `return` statements.

Phase 2 of the algorithm works on the entire group simultaneously, unlike Phase 1. The goal is to deal with the out-of-order aspect; i.e., Phase 2 permutes the statements in the clones in order to make matching statements line up in the same order in all clones in the group. As in Phase 1, dependences are preserved during this permutation in order to guarantee semantics preservation. In the example in Figure 1, as can be noted in the middle column (which is the algorithm's output), the algorithm permutes the second clone to: (a) move down the assignment to `base` to near the end of the clone, and (b) move up the assignment to `oRate` to the beginning of the "`if`" statement.

The algorithm terminates at this point, and the group has become *extractable*. Actual extraction can now proceed (the parameters and local variables need to be determined at this time); in Figure 1, the right column shows the result of extraction. The two (transformed) clones are simply replaced by calls to the newly extracted procedure `CalcPay` resulting in the rewritten fragments (notice that the code that was moved out of the way in Phase 1 is present after the two call sites). The promoted code is placed in the extracted procedure guarded by a flag parameter that is set appropriately at the two call sites.

1.3 Comparison to previous work

A few approaches to clone-group extraction in source code have been proposed in the literature [10, 2], while much work has been reported in the area of compacting assembly code by identifying and extracting clones (e.g., [18, 7, 8]). The novel aspect of our work in comparison to all previous clone-group extraction approaches is the use of semantics-preserving transformations both to move intervening non-cloned statements out of the way, and to reorder out-of-order clones. Also, ours is the first approach to handle exiting jumps in to-be-extracted code. These features allow our algorithm to perform well on difficult clone groups, where the performance of previous approaches is unsatisfactory.

In addition to disallowing exiting jumps, the approaches of [18, 7] only find sequences that are contiguous and that match exactly. The approach of [8] does allow inexactly matching sequences; however they perform no code motion, using guarding to resolve all mismatches when extracting non-contiguous clones or out-of-order clones. This means that every intervening non-matching statement, and *every copy* of an out-of-order statement will be placed in the extracted procedure in guarded form.

Original Fragment 1

```
    emp = 0;
    while(emp < nEmps) {
++    fscanf(fh,"%d",&hours);
++    if (hours < 0) {
++      error("illegal input");
++      break;
++    }
++    overPay = 0;
++    if (hours > 40) {
++      oRate = OvRate[emp];
++      excess = hours - 40;
        nOver++;
++      overPay = excess*oRate;
++    }
++    base = BasePay[emp];
++    Pay[emp] = base+overPay;
      emp++;
    }
```

Algorithm Output 1

```
    emp = 0;
    while(emp < nEmps) {
++    fscanf(fh,"%d",&hours);
++    if (hours < 0) {
++      error("illegal input");
++      goto L;
++    }
++    overPay = 0;
++    if (hours > 40) {
++      oRate = OvRate[emp];
++      excess = hours - 40;
++      overPay=excess*oRate;
++    }
++    base = BasePay[emp];
++    Pay[emp] = base+overPay;
   L: if (hours < 0)
        break;
      if(hours > 40)
        nOver++;
      emp++;
    }
```

Extracted Procedure

```
  void CalcPay(int emp,int *pHrs)
               int doLimit) {
      int overPay,oRate,
          excess,base;
++    fscanf(fh,"%d",pHrs);
++    if (*pHrs < 0) {
++      error("illegal input");
++      return;
++    }
++    overPay = 0;
++    if (*pHrs > 40) {
++      oRate = OvRate[emp];
++      excess = *pHrs - 40;
        if (doLimit)
****      if (excess > 10)
****        excess = 10;
++      overPay = excess*oRate;
++    }
++    base = BasePay[emp];
++    Pay[emp] = base+overPay;
  }
```

Original Fragment 2

```
    emp = 0;
    while(emp < nEmps) {
++    fscanf(fh,"%d",&hours);
++    if (hours < 0) {
++      error("illegal input");
++      break;
++    }
      totHours += hours;
++    base = BasePay[emp];
++    overPay = 0;
++    if (hours > 40) {
++      excess = hours - 40;
        if (excess > 10)
          excess = 10;
++      oRate = OvRate[emp];
++      overPay = excess*oRate;
++    }
++    Pay[emp] = base+overPay;
      emp++;
    }
```

Algorithm Output 2

```
    emp = 0;
    while(emp < nEmps) {
++    fscanf(fh,"%d",&hours);
++    if (hours < 0) {
++      error("illegal input");
++      goto L;
++    }
++    overPay = 0;
++    if (hours > 40) {
++      oRate = OvRate[emp];
++      excess = hours - 40;
****    if (excess > 10)
****      excess = 10;
++      overPay = excess*oRate;
++    }
++    base = BasePay[emp];
++    Pay[emp] = base+overPay;
   L: if (hours < 0)
        break;
      totHours += hours;
      emp++;
    }
```

Rewritten Fragment 1

```
    emp = 0;
    while(emp < nEmps) {
      CalcPay(emp,&hours,0);
      if (hours < 0)
        break;
      if(hours > 40)
        nOver++;
      emp++;
    }
```

Rewritten Fragment 2

```
    emp = 0;
    while(emp < nEmps) {
      CalcPay(emp,&hours,1);
      if (hours < 0)
        break;
      totHours += hours;
      emp++;
    }
```

Figure 1: Example illustrating extraction of two difficult clones

To be fair, it should be noted that while the assembly-code-compaction approaches solve a problem that is similar to ours, there is a significant difference: whereas the goal of our algorithm is to extract a *given* group of clones that represent a meaningful computation, their goal is to find and extract groups of clones that yield space savings. Because of this difference, it is reasonable for those algorithms to find and extract small, easy subsets of larger, more meaningful clones.

Previous approaches to clone-group extraction in source code [10, 2] have weaknesses similar to those exhibited by the assembly-code algorithms. The approach of [2], which works on object-oriented programs, is conceptually similar to the assembly-code approach of [8]; while they do allow extraction of inexact matches, they do no code motion. However, rather than place mismatching code in the extracted procedure with guards, they place each mismatching statement (or sequence of statements) in a new method, and they place a call to that new method at the appropriate point in the extracted procedure. The new methods that are called by the extracted procedure are then passed as parameters to the extracted procedure. The drawback of their approach is clear: since they perform no code motion, they are likely to produce extracted procedures that contain too many new method calls. However, their idea of method parameters could be incorporated into our approach to take the place of guarding, which could make the output cleaner in some situations.

The clone-group-extraction approach of [10] allows extraction of out-of-order clones. However, they do not address extracting non-contiguous clones or clones involving exiting jumps. Our studies (see Section 4) indicate that clones with these difficult aspects occur frequently, and thus handling them is essential for good performance.

The rest of this paper is organized as follows. Section 2

```
                     Clone 1

++     fscanf(fh,"%d",&hours);
++     if (hours < 0) {
++       error("illegal input");
++       goto L;
++     }
++     overPay = 0;
++     if (hours > 40) {
++       oRate = OvRate[emp];
++       excess = hours - 40;
++       overPay=excess*oRate;
++     }
++     base = BasePay[emp];
++     Pay[emp] = base+overPay;
    L: if (hours < 0)
         break;
       if(hours > 40)
         nOver++;
```

```
                     Clone 2

++     fscanf(fh,"%d",&hours);
++     if (hours < 0) {
++       error("illegal input");
++       goto L;
++     }
++     base = BasePay[emp];
++     overPay = 0;
++     if (hours > 40) {
++       excess = hours - 40;
****     if (excess > 10)
****       excess = 10;
++       oRate = OvRate[emp];
++       overPay = excess*oRate;
++     }
++     Pay[emp] = base+overPay;
    L: if (hours < 0)
         break;
       totHours += hours;
```

Figure 2: Output of Phase 1 for the two clones in Figure 1

presents basic assumptions and terminology. Section 3 defines our algorithm for extracting a group of clones. Section 4 presents the results of the study mentioned in Section 1.1, which provides quantitative data on how well our algorithm works in practice and how much of an advance it is over previous techniques.

2. ASSUMPTIONS AND TERMINOLOGY

We assume that the reader is familiar with the standard definitions of control and data dependence [9, 14, 4]. We assume that programs are represented using a set of control-flow graphs (CFGs), one for each procedure. Predicate nodes in CFGs have two or more outgoing edges (loop-predicates and "if" predicates have exactly two, whereas "switch" predicates have one for each "case" label and for the default case); the *exit* node of a CFG has no outgoing edges, whereas all other nodes (assignments and procedure calls) have a single outgoing edge. The *entry* node of a CFG is considered to be a pseudo-predicate (i.e., one that always evaluates to *true*); its outgoing *true* edge goes to the first actually executable node, and its outgoing *false* edge goes to the *exit* node.

Jumps (`gotos`, `returns`, `continues`, and `breaks`) are also considered to be pseudo-predicates, as in [3, 6]. The *true* edge out of a jump goes to the target of the jump, while the (non-executable) *false* edge goes to the node that would follow the jump if it were replaced by a no-op. Jump statements are treated as pseudo-predicates so that the statements that are semantically dependent on a jump—as defined in [15]—are also control dependent on it.

We assume that nesting relationships can be obtained from the abstract syntax tree (AST) representation of the program. "Normal" predicates (as well as the *entry* node) have nesting children, whereas non-predicates and jumps do not (whenever we say *nesting parent* or *nesting child*, we refer to only the immediate relationships, not transitive ones). Any nesting child n of a predicate p can be further qualified as a C-nesting child, where C is the label of the edge out of p under which n is nested (C can be *true*, *false*, or a "case" label value).

A *block* is a subgraph of a CFG that corresponds either to a simple statement, or to an entire compound statement such as a loop or an if-then-else statement.

The CFGs in Figure 3 illustrate blocks (the figure actually shows CFG fragments, not entire CFGs). Consider the second CFG: the node labeled a_2 in the beginning is a block by itself. Other examples of blocks are the subgraphs labeled b_2, f_2 and l (this last block is nested inside the block f_2).

Every block in a CFG is the nesting child of some normal predicate or the *entry* node; e.g., l is a *true*-nesting child of the predicate "`if (hours>40)`".

A *block sequence* B is a sequence of blocks in the CFG such that control flows out of every block to the next one in the sequence, ignoring jumps, and such that all the blocks are C-nesting children of the same predicate node p, for some value C. The block sequence B is a C-nesting child of p (and p is the nesting parent of B).

In Figure 3, $[a_2, b_2, j_2, e_2, f_2, k_2]$ is a block sequence (this sequence spans the entire second CFG fragment), as are $[b_2, j_2]$ (which is a subsequence of the previous sequence), and $[h_2, l, g_2, i_2]$ (which is nested inside the block f_2).

3. CLONE-GROUP EXTRACTION ALGORITHM

The input to the algorithm is a group of clones and the CFGs of the procedures that contain them (each individual clone is contained in a single procedure, although different clones in the group can be in different procedures). The input also includes a mapping that defines how the nodes in one clone match the nodes in the other clones. The algorithm allows inexact matches in a number of ways, thereby enabling extraction of a wide variety of difficult clones:

- Individual clones can be non-contiguous.
- Mapped nodes can be in different orders in the different clones.
- Although all mapped nodes are required to have mapped nesting parents, other differences in control structure (caused by intervening unmatched `if` blocks or jumps) are allowed.
- When the group consists of more than two clones, a node in one clone can be mapped to nodes in *some* but

not *all* other clones; i.e., different clones in the group can consist of different numbers of nodes.

The input mapping is required to satisfy the following properties: The mapping is transitive; i.e., if m, n and t are nodes in three different clones, m is mapped to n and n is mapped to t, then m is mapped to t. A node in a clone can be mapped to at most one node in any other clone (this allows for a node in a clone to be mapped to nodes in *some* but not all other clones). The clones are "non-overlapping"; i.e., the tightest block sequences containing the individual clones are disjoint (by "tightest" block sequence we mean innermost, in terms of nesting, and within that the shortest, in terms of the number of blocks). Mapped nodes have mapped nesting parents; i.e., if a node n is mapped to a node m, then one of the following must be true: neither node has a nesting parent within the tightest block sequence containing its clone, or m and n are the C-nesting children of two mapped predicates, for some value C. Mapped nodes are of the same "kind"; e.g., an assignment node is mapped only to other assignment nodes, a `while` predicate is mapped only to other `while` predicate nodes, and so on.

When the algorithm finishes, the group of clones will have been transformed so that they are easily extractable into a separate procedure.

3.1 Phase 1: Making individual clones extractable

As discussed in the Introduction, Phase 1 of our clone-group extraction algorithm applies the algorithm of [12] to each individual clone. Given the set of nodes that comprise a single clone, that algorithm transforms the tightest block sequence containing those cloned nodes in a semantics-preserving manner by moving as many intervening non-cloned nodes as possible out of the way of the cloned nodes, promoting nodes that cannot be moved, and handling exiting jumps. The output of the algorithm is three block sequences: the first one contains the non-cloned nodes that were moved out of the way before the cloned nodes, the second one contains the cloned and promoted nodes, and the last one contains the non-cloned nodes that were moved after the cloned nodes. The block sequence that contains the cloned nodes is a single-entry single-exit structure; this is a structure that is easily extractable into a procedure.

The algorithm runs in polynomial time (in the size of the block sequence that it transforms), and always succeeds. The final output produced by the algorithm for the clone in Fragment 1 of Figure 1 is shown in the top half of Figure 2.

After Phase 1, the first and third block sequences produced by that phase are ignored (because they contain only non-cloned nodes), and Phase 2 operates on the block sequences that contain the cloned nodes. The promoted nodes were originally not mapped to any nodes, and they continue to be regarded that way.

If at the end of Phase 1 the clone group is "in order" (as defined below in Section 3.2), then the algorithm terminates (the group has already become extractable). Otherwise, the block sequences in the clones (at all levels of nesting) are permuted in a semantics-preserving manner, to make the clone group be in-order. This is Phase 2 of the algorithm, and is described in Section 3.2.

3.2 Phase 2: reordering block sequences

The goal of Phase 2 is to make the given group of clones be in-order if they are not already. This involves the following two steps:

1. Identify sets of mapped blocks, and sets of mapped block sequences.
2. For every set of mapped block sequences that is not *top-level in-order*, permute the sequences in that set in a semantics-preserving manner to make it top-level in-order. All sets of mapped block sequences – those at the outer level as well as those nested inside others – are considered in this step.

Recall that part of the input to the algorithm is a mapping between the nodes in the individual clones. That mapping applies to the blocks that represent individual statements; in this phase, we extend the mapping to blocks that represent compound statements, and to block sequences.

By definition, a "compound" block b_1 in clone C_1 is mapped to a "compound" block b_2 in clone C_2 iff at least one node in b_1 is mapped to a node in b_2, and no node in b_1 is mapped to a node in C_2 that is not in b_2. (Note that either of the two blocks may contain unmapped nodes, or may contain nodes that are mapped only to nodes in other clones besides C_1 and C_2.) Similarly, by definition, a block sequence s_1 in clone C_1 is mapped to a block sequence s_2 in clone C_2 iff at least one block in s_1 is mapped to a block in s_2, and no block in s_1 is mapped to a block in C_2 that is not in s_2.

Example: Throughout this discussion, the two-clone group in Figure 3 serves as our running example. This figure shows the clones in Figure 1 in the form of CFG subgraphs, after Phase 1 is done. The dashed edges out of the `goto` nodes are "non-executable" edges (see Section 2).

In the figure, each (individual-statement and compound) block is labeled. The mappings for the individual-statement blocks are assumed to be the intuitive ones (identical statements are mapped, e.g., nodes a_1 and a_2, and nodes c_1 and c_2). Mappings for the compound blocks are determined according to the definition given above; blocks b_1 and b_2 are mapped, blocks f_1 and f_2 are mapped, and so on. Notice that f_2 (in the second clone) contains an inner block l that is not mapped to any block in the first clone.

There are a total of three sets (pairs, in this case) of mapped block sequences: (1) $[a_1, b_1, e_1, f_1, j_1, k_1]$, $[a_2, b_2, j_2, e_2, f_2, k_2]$, (2) $[c_1, d_1]$, $[c_2, d_2]$, and (3) $[g_1, h_1, i_1]$, $[h_2, l, g_2, i_2]$. Notice that the second and third pairs of block sequences are nested inside the first one, but are still considered as separate pairs of block sequences. □

A set of mapped block sequences is said to be top-level in-order iff, for every pair of block sequences in the set, considering only the top-level blocks in each sequence that are mapped to top-level blocks in the other, the corresponding top-level blocks are in the same order in both sequences.

Example: Of the three sets of mapped block sequences listed above, set 2 is top-level in-order, while sets 1 and 3 are not. (The presence of the unmapped block l has nothing to do with set 3 being not in-order.) □

Note that a set of mapped block sequences can be top-level in-order even if smaller sets of mapped block sequences nested inside are not top-level in-order (hence the use of the phrase "top-level").

It is not difficult to see that a group of clones is (completely) in-order if every set of mapped block sequences is

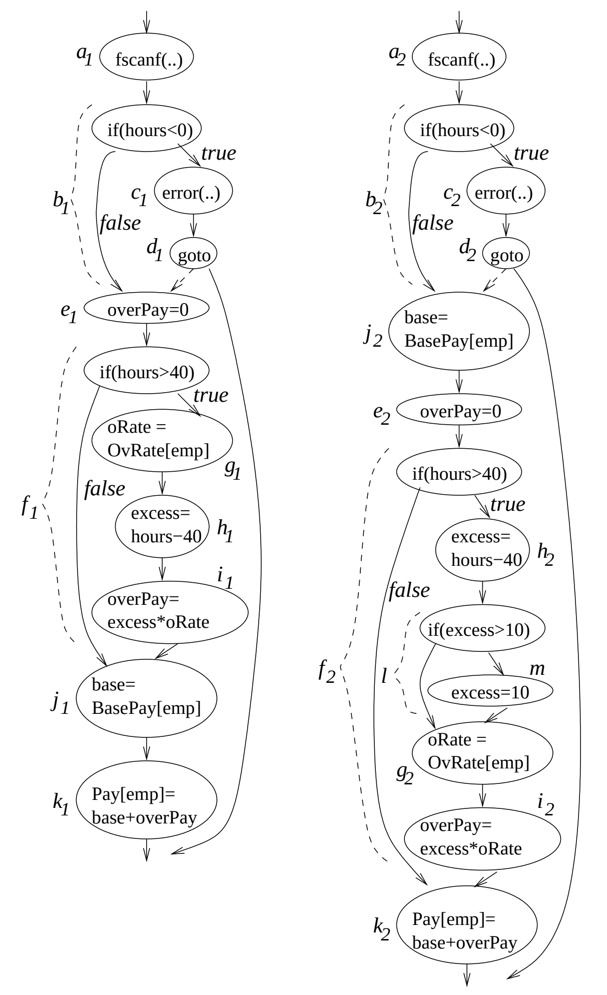

Figure 3: Clones from Figure 1, after Phase 1

top-level in-order. Therefore our approach is to visit each set of mapped block sequences that is not top-level in-order, including nested sets, and to permute the block sequences to make the set be top-level in-order.

"Visiting" a set of mapped block sequences involves generating ordering constraints among the blocks in each sequence, and then permuting the (top-level) blocks in each sequence while respecting the constraints. Constraint generation is described in the next two subsections; Section 3.2.1 describes constraints for preserving control-dependences, and Section 3.2.2 describes constraints for preserving data dependences. The actual procedure for doing the permutation is then described in Section 3.2.3.

3.2.1 Constraints for preserving control dependences

Constraints are needed to preserve control dependences while permuting a block sequence if it has any of the following properties: there are jumps outside the sequence whose target is inside, there are jumps inside the sequence whose target is outside, or there are jumps from one block in the sequence to another. If none of these conditions hold for a block sequence, then any permutation preserves all control dependences, and therefore no control-dependence-based constraints are needed.

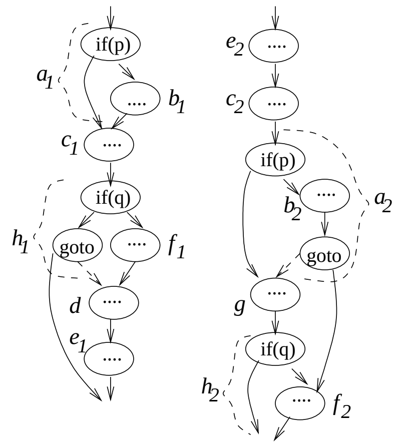

Figure 4: Example illustrating control dependence constraints

Figure 4 contains an (artificial) illustrative example. Assume every node in the example (except the predicates and jumps) is an assignment, and that block a_1 is mapped to block a_2, h_1 is mapped to h_2, and so on. The assignment nodes d and g are unmapped, as are the two `gotos`. Our algorithm allows such unmapped jumps, and these make extraction challenging (unmapped jumps are plausible in real clones; e.g., a computation could occur at two places, but one of them could have an intervening statement that checks some boundary condition and jumps out based on that).

The two outer-level mapped block sequences $[a_1, c_1, h_1, d, e_1]$, $[e_2, c_2, a_2, g, h_2]$ contain the kinds of jumps mentioned earlier, and therefore need control-dependence-based constraints. Consider the first of these two block sequences. Any permutation needs to preserve the property that d and e_1 come after h_1 (otherwise d and e_1 would execute whether or not the `goto` in h_1 executes, which is incorrect). Similarly, a_1 and c_1 need to come before h_1 (else the `goto` could incorrectly bypass a_1 and c_1). Notice that all these constraints are simple precedence constraints. Going on to the second block sequence in the figure, the constraint needed is more complex: g must remain in between blocks a_2 and h_2, whereas e_2 and c_2 can be anywhere *except* between these two blocks.

We therefore need the following rules for generating constraints that preserve control dependences:

Rule 1: If a block j in a block sequence either contains a jump whose target is outside the sequence or contains a node that is the target of a jump outside the sequence, then for every block p that precedes j in that same block sequence, generate a constraint $p < j$ (meaning p must precede j), and for every block s that follows j in the sequence generate a constraint $j < s$.

Rule 2: If e and g are blocks in a block sequence such that e precedes g and there is a jump node in either one whose target node is inside the other, then: (a) for every block f that is in between e and g in that same sequence, generate constraints $e < f$ and $f < g$; (b) for every block o in the same sequence that is *not* in between e and g generate a constraint $o \notin [e, g]$ (which says that o cannot be in between e and g after the permutation); (c) generate the constraint $e < g$.

3.2.2 Constraints for preserving data dependences

The data-dependence-based constraints ensure preservation of data dependences. The constraints are generated according to a simple rule: if a block e precedes a block f in a block sequence and if there is a flow, anti, output or def-order dependence from some node in e to some node in f then generate a constraint $e < f$.

3.2.3 Permuting the block sequences

So far we have generated a set of constraints for each block sequence in the set of mapped block sequences currently being visited. The goal now is to find a permutation of each block sequence such that the set becomes top-level in-order. This can be achieved by creating a graph G whose vertices represent top-level blocks, and whose edges represent constraints:

1. Create a graph G such that each set of mapped top-level blocks in the given set of block sequences is represented by a vertex in G. A block in one sequence that is not mapped to a block in any other sequence gets its own vertex in G.
2. For each simple precedence constraint of the form $b_1 < b_2$, add a directed edge to G from the vertex representing b_1 to the vertex representing b_2.
3. Obtain a topological ordering of the vertices in G that also respects the "$\not\in$" constraints. If no such ordering exists, fail (the given block sequences cannot be made top-level in-order while satisfying all the constraints). More details on this step are provided below.
4. For each block sequence, obtain its final permutation by simply taking the total ordering of the previous step, and projecting out the vertices that represent blocks not in the sequence.

Example: We return to the example in Figure 3. For illustration, we pick the two mapped block sequences $[g_1, h_1, i_1]$ and $[h_2, l, g_2, i_2]$. The vertices of the graph G are g, h, i, l, where g represents the mapped blocks g_1 and g_2, h represents mapped blocks h_1 and h_2, and so on. The data dependence constraints, when translated into graph edges, are: $g \rightarrow i, h \rightarrow i, l \rightarrow i, h \rightarrow l$. The edge $h \rightarrow i$ is caused both by flow and def-order dependences, whereas the others are caused by flow dependence only. The two constraints involving l represent constraints for the second block sequence, whereas the other two represent constraints for both block sequences. Neither of these two block sequences requires any control-dependence constraints.

One of the possible topological orderings that is consistent with all constraints is $[g, h, l, i]$. Using this, we obtain the following permutations for the two block sequences respectively: $[g_1, h_1, i_1]$ and $[g_2, h_2, l, i_2]$. Notice that the two block sequences are now top-level in-order. $\square$

An observation that helps in defining a solution for Step 3 above is that any "$\not\in$" constraint of the form $a \not\in [b, c]$ is logically equivalent to the constraint $(a < b) \vee (c < a)$. Therefore a straightforward way to implement this step is: for each constraint $a \not\in [b, c]$ add one of the two edges $a \rightarrow b$, $c \rightarrow a$ to the graph G, and see if a topological ordering is possible (i.e., see if the resulting graph is acyclic); use backtracking to systematically explore the choices until an ordering is found (and fail otherwise). This approach, although simple, has worst-case time complexity exponential in the number of "$\not\in$" constraints. However, this is unlikely to be a problem in practice because "$\not\in$" constraints arise only when the program contains `gotos` (excluding ones introduced by Phase 1) *and* the complex condition described in Rule 2 in Section 3.2.1 holds. In our experimental studies (Section 4), there were no instances of a "$\not\in$" constraint. Still, an interesting open question is whether there is a polynomial-time algorithm for Step 3.

4. EXPERIMENTAL RESULTS

This section describes the results of a study performed to determine how often "difficult" clone groups arise in practice, and to evaluate the performance of our algorithm compared both to an "ideal" extractor (a human), and to the best previously reported automatic approach for clone-group extraction [8]. Our dataset consisted of 56 clone groups, containing a total of 185 individual clones. The median number of clones per group was 2 while the maximum was 14; the median size (number of simple statements and predicates) of a clone was 6, while the maximum was 53. The dataset was drawn from three programs: the Unix utilities *bison* and *make*, and NARC[1] [17], a graph-drawing engine developed by IBM. These programs range in size from 11 to 30 thousand lines of code.

To expedite the process of finding clone groups to extract, we used the clone-detection tool reported in [11], which can find clone groups that involve difficult aspects. From the output of the tool we filtered away the clones that were not good candidates for extraction (very small clones, which are not beneficial to extract as well as clones that could not be cleanly extracted by an "ideal" extractor).

4.1 Comparison to ideal extraction

As the first step in the comparison we extracted each clone group in an "ideal" manner, using our own judgment. Some of the techniques used during this process (e.g., moving out intervening unmapped nodes, reordering out-of-order matches) are also used by our algorithm; other techniques not incorporated in the algorithm were also used as necessary (this is discussed later). The second step in the comparison was to (manually) apply the algorithm to each clone group. Figure 5 presents the results of the comparison.

Figure 5(a) summarizes the comparative performance of our algorithm. The set of all clone groups is divided into four disjoint categories, one per row. The first row shows that 29 clone groups (containing a total of 100 individual clones) were "not difficult"; i.e., the clones in these groups were contiguous, did not involve exiting jumps, and were in order. Such groups are extractable to begin with, and therefore our algorithm had nothing to do. The remaining 27 clone groups in the dataset were difficult (i.e., required transformations in at least one of the two phases of the algorithm).

The second row of Figure 5(a) shows that our algorithm produced exactly the ideal output on 18 of the 27 difficult clone groups (columns 2–5 characterize the difficult aspects in all the 27 groups). The third and fourth rows of Figure 5(a) pertain to clone groups on which our algorithm succeeded but produced non-ideal output, and failed, respectively (more details on these later).

Note that our algorithm makes a clone group extractable,

[1]NARC is a registered trademark of IBM.

	Groups		Individual clones		
Category	# total	# out-of ord.	# total	# non-contig.	# exiting jumps
Not difficult	29	-	100	-	-
Difficult, ideal output	18	7	58	15	12
Difficult, non ideal output	7	-	19	9	9
Difficult, failure	2	2	8	4	5

(a) Characterization of algorithm output

Technique	Ideal output	Non-ideal output	
		human	algo
Phase 1			
Moving out nodes	13	2	9
Promotion	2	6	3
Phase 2			
Reordering out-of-order clones	7	-	-

(b) Techniques used on difficult clone groups, with number of clones (Phase 1) and number of groups (Phase 2)

Figure 5: Comparison of algorithm to ideal extraction

but does not perform actual extraction; therefore, the comparison of the algorithm's output to that of ideal extraction involves verifying that the same nodes were moved out in both cases with identical usage of predicate duplication, that the same nodes were promoted, that out-of-order matches were handled in one case if handled in the other, etc. Issues pertaining to actual extraction (e.g., determining the parameters) are ignored during the comparison. (Note: when we say that the algorithm performed ideally on a group, we mean that Phase 1 performed ideally on every individual clone in the group, *and* Phase 2 subsequently performed ideally on the entire group.)

Figure 5(b) shows, for each technique incorporated in our algorithm, how often it was used by the algorithm as well as by the ideal extraction (non-difficult clone groups, as well as the two groups on which our algorithm fails, are omitted from Figure 5(b)). Each technique appears in its own row; in the rows pertaining to the techniques of Phase 1 the counts given are for individual clones, whereas clone-group counts are given in the row for Phase 2. The second column (labeled "Ideal output") pertains only to the clone groups on which the algorithm performed ideally; therefore the numbers in this column pertain both to the ideal extraction and the extraction performed by the algorithm. The third and fourth columns pertain to the groups on which the algorithm performed non-ideally; two separate sets of numbers are required here because the algorithm and the ideal extraction do not use the exact same techniques on these groups.

As shown in Figure 5(a), the algorithm performs ideally on 18 of the 27 difficult groups; however, it produces non-ideal output on 7 difficult groups, and fails on 2. One reason for the algorithm's non-ideal behavior is that on some individual clones Phase 1 over aggressively moves out unmapped nodes (with predicate duplication), while the ideal extraction uses promotion (the numbers in Figure 5(b) bear this out); this problem can likely be addressed by designing better heuristics. Another issue is that the ideal extraction is able to use more complex matches than the algorithm. For example, in one case the ideal extraction matched a single node outside an `if` block in one clone with two copies in each of the two branches of the `if` in the other clone. In another case, the ideal extraction matched a sequence of two separate `if` blocks (with mutually exclusive conditions) in one clone with a single "`if..then..else..`" block in the other clone. The algorithm does not allow such matches (they are prevented by the "mapped nesting parent" requirement, discussed in the beginning of Section 3); thus those nodes are unmapped in the input supplied to the algorithm.

The algorithm fails in one of the two cases because of an out-of-order group that cannot be reordered because our data-dependence constraints (Section 3.2.2) are conservative; human judgment in this case reveals that reordering is actually safe. The other group is not strictly a failure in this sense; here, the nesting structures of the clones are so different that all but one of the nodes matched by the ideal extraction have to be left out of the mapping provided to the algorithm (the ideal extraction, in this case, involves extensive rewriting). We therefore chose to place this group in the "failure" category.

4.2 Comparison to a previous algorithm

The second part of our study involved comparing our algorithm to the previously reported clone-group extraction algorithm of Debray et al [8]. While the goal of that work is to find and extract matching fragments of code in order to compact assembly programs, the technique can also be applied to source code. We chose that approach for our comparison because it employs more techniques than other assembly-code compaction approaches, is conceptually similar to the source-code based approach of [2], and is likely to perform better than the other source-code based approach [10] (groups involving non-contiguous clones, whose extraction is addressed by [8] but not by [10], occur more frequently in our dataset than out-of-order groups that are handled better by [10]).

The approach of [8] works as follows: For each procedure, they build a CFG in which the nodes represent basic blocks. They then find groups of isomorphic single-entry single-exit subgraphs in the CFGs such that corresponding basic blocks have similar instruction sequences, and then extract each group into a new procedure. If two corresponding basic blocks in a group do not have identical instruction sequences (modulo register renamings), they walk down the two sequences in lock-step and "promote" every mismatching instruction (i.e., it is included in the extracted procedure with a guard). Thus, although inexact matches are allowed, every mismatch is handled by their approach using the guarding mechanism: every intervening non-matching statement and every copy of an out-of-order matching statement is placed in the extracted procedure with guarding.

In addition to the fact that they use guarding to handle all mismatches, the approach of [8] has two important weaknesses compared with ours: One is the requirement that the CFG subgraphs be isomorphic; e.g., they would fail to extract the two clones in our running example (Figure 1), because the presence of the intervening non-matching block

Category	# clone-groups
They fail (ours ideal)	8
Their output non-ideal (ours ideal)	10
They fail (our output non-ideal)	4
Both fail	1
Both outputs non-ideal	1
We fail (their output non-ideal)	1
Their output ideal (ours non-ideal)	2

Figure 6: Comparison of our algorithm to that of Debray et al

labeled l makes the two subgraphs non-isomorphic. In other words, they allow non-matching code only if it is inside a basic block.

The other weakness is that, because they are restricted to extracting single-entry single-exit structures, they cannot handle exiting jumps. In the example of Figure 1, due to the presence of the `breaks`, the smallest single-entry single-exit structure enclosing each clone is the entire surrounding loop. Therefore they could extract the entire loop, but not just the desired clones. In this particular example that is not a major problem since there is very little code in the loops other than the desired clones. However, if there were a significant number of unmatched statements in the loops, then either they would do no extraction at all (if the non-matching code in the two loops caused the CFG subgraphs to be non-isomorphic), or they would place all non-matching code from both loops in the extracted procedure with guarding. Both of these alternatives are clearly less desirable than just extracting the matching code.

Figure 6 provides data comparing the performance of our algorithm and that of [8] on the 27 difficult clones in the dataset. As in the comparison to ideal extraction, we restrict our attention to the transformations applied by the two approaches; issues relating to actual extraction are ignored (because we do not address them, and they address the issues only at the assembly-code level). Therefore, for example, if two matching nodes differ in some subexpression that is parameterized away during the ideal extraction, we treat the two nodes as non-problematic for both approaches.

The 27 difficult clone groups are divided into 7 disjoint categories, with one per row in Figure 6. As shown in the first six rows, the algorithm of [8] either fails or performs non-ideally on almost all of the clone groups, 25 out of 27, while our algorithm produces the ideal output on 18 of those 27 groups (the first two rows). The main reason for the better performance of our algorithm is that, as discussed earlier, it employs a variety of transformations to tackle difficult aspects whereas [8] uses only promotion. As shown in the last two rows of the figure, their algorithm performs better than ours on 3 clone groups. One of these is the out-of-order group on which our algorithm fails; they succeed (non-ideally) on this one by using guarding. The other two groups are ones where our algorithm over aggressively moves unmapped nodes out, where guarding, which is their solution, is the ideal alternative.

4.3 Summary of studies

Our algorithm performed ideally on two thirds of the difficult clone groups, while the algorithm of [8] performed ideally on less than eight percent. Given that no automatic algorithm is likely to be able to employ the full range of transformation techniques used by a human (and thus 100% ideal performance by an automatic algorithm is probably not feasible) we consider the results of this study to be very encouraging, indicating that our algorithm is likely to be very useful in practice. The study also provides evidence that previously reported approaches for assembly-code compaction are not powerful enough in our context where meaningful clones need to be extracted at the source-code level. However, it may well be the case that our techniques for moving unmatched code and for handling exiting jumps will be beneficial in the context of assembly-code compaction.

Acknowledgements

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

5. REFERENCES

[1] B. Baker. On finding duplication and near-duplication in large software systems. In *Proc. IEEE Working Conf. on Reverse Engineering*, pages 86–95, July 1995.

[2] M. Balazinska, E. Merlo, M. Dagenais, B. Lague, and K. Kontogiannis. Partial redesign of Java software systems based on clone analysis. In *Proc. IEEE Working Conf. on Reverse Engineering*, pages 326–336, 1999.

[3] T. Ball and S. Horwitz. Slicing programs with arbitrary control flow. In *Lecture Notes in Computer Science*, volume 749, New York, NY, Nov. 1993. Springer-Verlag.

[4] S. Bates and S. Horwitz. Incremental program testing using program dependence graphs. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 384–396, Jan. 1993.

[5] I. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. Clone detection using abstract syntax trees. In *Int. Conf. on Software Maintenance*, pages 368–378, 1998.

[6] J. Choi and J. Ferrante. Static slicing in the presence of goto statements. *ACM Trans. on Programming Languages and Systems*, 16(4):1097–1113, July 1994.

[7] K. Cooper and N. McIntosh. Enhanced code compression for embedded RISC processors. In *Proc. ACM Conf. on Programming Language Design and Implementation*, pages 139–149, May 1999.

[8] S. Debray, W. Evans, R. Muth, and B. D. Sutter. Compiler techniques for code compaction. *ACM Trans. on Programming Languages and Systems*, 22(2):378–415, Mar. 2000.

[9] J. Ferrante, K. Ottenstein, and J. Warren. The program dependence graph and its use in optimization. *ACM Trans. on Programming Languages and Systems*, 9(3):319–349, July 1987.

[10] W. Griswold and D. Notkin. Automated assistance for program restructuring. *ACM Trans. on Software Engineering and Methodology*, 2(3):228–269, July 1993.

[11] R. Komondoor and S. Horwitz. Using slicing to identify duplication in source code. In *Proc. Int. Symposium on Static Analysis*, pages 40–56, July 2001.

[12] R. Komondoor and S. Horwitz. Effective, automatic procedure extraction. *Submitted to Int. Conf. on*

Compiler Construction, 2003.

[13] K. Kontogiannis, R. Demori, E. Merlo, M. Galler, and M. Bernstein. Pattern matching for clone and concept detection. *Automated Software Engineering*, 3(1–2):77–108, 1996.

[14] D. J. Kuck, R. H. Kuhn, D. A. Padua, B. Leasure, and M. Wolfe. Dependence graphs and compiler optimizations. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 207–218, Jan. 1981.

[15] S. Kumar and S. Horwitz. Better slicing of programs with jumps and switches. In *Proc. Fundamental Approaches to Software Engineering*, Apr. 2002.

[16] 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 Maintenance*, pages 314–321, 1997.

[17] V. Waddle and A. Malhotra. An e log e line crossing algorithm for leveled graphs. In *Lecture Notes in Computer Science*, volume 1731, pages 59–71. Springer-Verlag, 1999.

[18] M. Zastre. Compacting object code via parameterized procedural abstraction. Master's thesis, Department of Computer Science, University of Victoria, British Columbia, 1995.