Volume 5, Issue 3, March 2015 ISSN: 2277 128X
International Journal of Advanced Research in Computer Science and Software Engineering
Research Paper
Available online at: www.ijarcsse.com

Clone Detection Using Abstract Syntax Trees

Ritu
Departament of Computer Science & Engineering
Chandigarh Group of College,
Landra, India

Abstract: Existing research suggests that a considerable fraction (5-10%) of the source code of large-scale computer programs is duplicate code (" clones"). Detection and removal of such clones promises decreased software maintenance costs of possibly the same magnitude. Previous work was limited to detection of either near-misses differing only in single lexems, or near misses only between complete functions. This paper presents simple and practical methods for detecting exact and near miss clones over arbitrary program fragments in program source code by using abstract syntax trees. Previous work also did not suggest practical means for removing detected clones. Since our methods operate in terms of the program structure, clones could be removed by mechanical methods producing in-lined procedures or standard preprocessor macros.

A tool using these techniques is applied to a C production software system of some 400K source lines, and the results confirm detected levels of duplication found by previous work. The tool produces macro bodies needed for clone removal, and macro invocations to replace the clones. The tool uses a variation of the well-known compiler method for detecting common sub-expressions. This method determines exact tree matches; a number of adjustments are needed to detect equivalent statement sequences, commutative operands, and nearly exact matches. We additionally suggest that clone detection could also be useful in producing more structured code, and in reverse engineering to discover domain concepts and their implementations.

Keywords: Software maintenance, clone detection, software evaluation, Design Maintenance System

I. INTRODUCTION

Data from previous work [Lague97, Baker95] suggests that a considerable fraction (5-10%) of the source of large computer programs is duplicated code. Programmers routinely perform ad hoc code reuse by brute-force copying code fragments that implement actions similar to their current need, and performing a cursory (often empty!) customization of the copied code to the new context.

The act of copying indicates the programmer's intent to reuse the implementation of some abstraction. The act of pasting is breaking the software engineering principle of encapsulation. While cloning may be unstructured, it is commonplace and unlikely to disappear via fiat. *Its very commonness suggests we should offer programmers tools that allow them to use implementations of abstractions without breaking encapsulation.*

In the meantime, detection and replacement of such redundant code by subroutine calls, in-lined procedure calls, macros, or other equivalent shorthand that effect the same result, promises decreased software maintenance costs corresponding to the reduction in code size. Since much of present software engineering is focused on finding small-percentage process gains, a mechanical method for achieving up to 10% savings is well worth investigating.

We define an *idiom* as a program fragment that implements a recognizable concept (data structure or computation). A *clone* is a program fragment that identical to another fragment. A *near miss clone* is a fragment, which is nearly identical to another. Clones usually occur when an idiom is copied and optionally edited, producing exact or near-miss clones.

Previous clone detection work was limited to detection of either exact textual matches, or near misses only on complete function bodies. This paper presents practical methods, using abstract syntax trees (ASTs), for detecting exact and near miss clones for arbitrary fragments of program source code. Since detection is in terms of the program structure, clones can be factored out of the source using conventional transformational methods.

A tool using these detection techniques is applied to a C production software system of some 400K SLOC and the results confirm detected levels of duplication found by previous work. The tool uses a variation of the well-known compiler method for detecting common sub-expressions [Aho85], which determines exact tree matches essentially by hashing. A number of adjustments are needed to detect clones in the face of commutative operands, near misses, and statement sequences.

We additionally suggest that clone detection could also be useful in producing more structured code, as well as discovering domain concepts and their idiomatic implementations.

Section 2 discusses the causes of clones. Section 3 discusses why we chose AST-based clone detection. Section 4 describes the basic AST clone detection algorithm. Section 5 builds on the basic method to detect clone sequences. Section 6 discusses detection of near-miss clones generalized from previously discovered clones. Section 7 discusses the problems of engineering a clone detector for scale. Section 8 reports the results of applying the clone detector to a running software system, and analyzes the results. Section 9 discusses the relation between clones and domain concepts. Section 10 reports possible future work. Section 11 describes related work.

II. WHY DO CLONES OCCUR?

Software clones appear for a variety of reasons:

- Code reuse by copying pre-existing idioms
- Coding styles
- Instantiations of definitional computations
- Failure to identify/use abstract data types
- Performance enhancement
- Accidents

State of the art software design has structured design processes, and formal reuse methods. Legacy code (and, alas, far too much of new code) is constructed by less-structured means. In particular, a considerable amount of code is or was produced by ad hoc reuse of existing code. Programmers intent on implementing new functionality find some code idiom that perform a computation nearly identical to the one desired, copy the idiom wholesale and then modify in place. Screen editors that universally have " copy" and " paste" functions hasten the ubiquity of this event.

In large systems, this method may even become a standard way to produce variant modules. When building device drivers for operating systems, much of the code is boilerplate, and only the part of the driver dealing with the device hardware needs to change. In such a context, it is commonplace for a device driver author to copy entirely an existing, well-known, trusted driver and simply modify it. While this is actually good reuse practice, it exacerbates the maintenance problem of fixing a bug found in the " trusted" driver by replicating its code (and reusing its bugs) over many new drivers.

Sometimes a " style" for coding a regularly needed code fragment will arise, such as error reporting or user interface displays. The fragment will purposely be copied to maintain the style. To the extent that the fragment consists only of parameters this is good practice. Often, however, the fragment unnecessarily contains considerably more knowledge of some program data structure, etc.

It is also the case that many repeated computations (payroll tax, queue insertion, data structure access) are simple to the point of being definitional. As a consequence, even when copying is not used, a programmer may use a mental macro to write essentially the same code each time a definitional operation needs to be carried out. If the mental operation is frequent, he may even develop a regular style for coding it. Mental macros produce near-miss clones: the code is almost the same ignoring irrelevant order and variable names.

Some clones are in fact complete duplicates of functions intended for use on another data structure of the same type; we have found many systems with poor copies of insertion sort on different arrays scattered around the code. Such clones are an indication that the data type operation should have been supported by reusing a library function rather than pasting a copy.

Some clones exist for justifiable performance reasons. Systems with tight time constraints are often hand-optimized by replicating frequent computations, especially when a compiler does not offer in-lining of arbitrary expressions or computations.

Lastly, there are occasional code fragments that are just accidentally identical, but in fact are not clones. When investigated fully, such apparent clones just are not intended to carry out the same computation. Fortunately, as size goes up, the number of accidents of this type drops off dramatically.

Ignoring accidental clones, the presence of clones in code unnecessarily increases the mass of the code. This forces programmers to inspect more code than necessary, and consequently increases the cost of software maintenance. One could replace such clones by invocations of clone abstractions once the clones can be found, with potentially great savings.

III. CLONE DETECTION USING ASTS

The basic problem in clone detection is the discovery of code fragments that compute the " same" result. To do this, we must first fragment the program in parts we are willing to compare, and then determine if fragment pairs are equivalent. Since determining that even a single fragment halts is impossible, we cannot determine that two arbitrary program fragments halt under the same circumstance, and thus it is impossible in theory to decide that they compute identical results. Since false negatives are acceptable (as engineers we have no choice), then deep semantic analysis conservatively bounded by time limits could be used for equivalence detection. However, considerable infrastructure may be required— in the form of semantic definitions, theorem provers, etc. In practice, we are willing to give up detecting complete semantic equivalence because many clones come about due to copy-and-paste editing processes.

Simpler definitions of code equivalence may suffice if too many false positives are not produced. This suggests clone detection by more syntactic methods. One can go as far as comparing source lines. Source line equality assumes that the cloning process introduced no changes in identifiers, comments, spacing, or other non-semantic changes, and thus limits clone detection to exact matches. Consequently, it fails to detect near-miss clones. Closer to full semantics but still a practical possibility would be to compare program representations in which control and data flows are explicit.

Semantic Designs is building transformational tools (DMS) to help modify large software systems [Baxter97]. Such tools typically parse source programs into ASTs as a first step before transformation. Due to the early product state of our tools, we chose to investigate comparing syntax trees. This had the attraction of directly avoiding confusing but uninteresting changes at the lexical level.

As a first step in the clone detection process, the source code is parsed and an AST is produced for it. After that, three main algorithms are applied to find clones. The purpose of the first algorithm, which we call the Basic algorithm, is to detect sub-tree clones. The second one, which we call the sequence detection algorithm, is concerned with the detection of variable-size sequences of sub-tree clones, and is used essentially to detect statement and declaration sequence clones. The third algorithm looks for more complex near-miss clones by attempting to generalize combinations of other clones. The resulting detected clones can then be prettyprinted. We did not carry out clone removal.

VI. FINDING SUB-TREE CLONES

In principle, finding sub-tree clones is easy: compare every subtree to every other sub-tree for equality. In practice, several problems arise: near-miss clone detection, sub-clones, and scale. Near misses we handle by comparing tress for similarity rather than exact equality. The sub-clone problem is that we wish to recognize maximally large clones, so clone subtrees of detected clones need to be eliminated as reportable clones.

The scale problem is harder. For an AST of N nodes, this comparison process is O(N^3), and, empirically, a large software system of M lines of code has N=10*M AST nodes (if we consider comparing sequences of trees, the process is O(N^4)!). Thus, the amount of computation becomes prohibitively large.

In order to tackle this problem it is possible to partition the sets of comparisons by categorizing sub-trees with hash values. The approach is based on the tree matching technique for building DAGs for expressions in compiler construction [Aho86]. This allows the straightforward detection of exact sub-tree clones. If we hash sub-trees to B buckets, then only those trees in the same bucket need be compared, cutting the number of comparisons by a factor of B. We choose a B of approximately the same order as N; in practice, B=10% N means little additional space at great savings in terms of computation. We have found that the cost of comparing individual trees averages close to a constant, rather than O(N), and so hashing allows this computation to occur in practice in time O(N).

This approach works well when we are finding exact clones. When locating near-miss clones, hashing on complete subtrees fails precisely because a good hashing function includes all elements of the tree, and thus sorts trees with minor differences into different buckets. We solved this problem by choosing an artificially bad hash function. This function must characterized in such a way that the main properties one wants to find on near-miss clones are preserved. As we described in Section 2, near miss clones are usually created by copy and paste procedures followed by small modifications. These modifications usually generate small changes to the shape of the tree associated with the copied piece of code. Therefore, we argue that this kind of near-miss clone often has only some different small sub-trees. Based on this observation, a hash function that ignores small sub-trees is a good choice. In the experiment presented here, we used a hash function that ignores only the identifier names (leaves in the tree). Thus our hashing function puts trees which are similar modulo identifiers into the same hash bins for comparison.

Rather than comparing trees for exact equality, we compare instead for similarity, using a few parameters. The similarity threshold parameter allows the user to specify how similar two sub-trees should be. The similarity between two sub-trees is computed by the following formula:

Similarity = 2 x S / (2 x S + L + R) where:

S = number of shared nodes

L = number of different nodes in sub-tree 1 R = number of different nodes in sub-tree 2

The mass threshold parameter specifies the minimum sub-tree mass (number of nodes) value to be considered, so that small pieces of code (e.g., expressions) are ignored.

We combine these methods to detect sub-tree clones, giving the Basic clone detection algorithm in Figure 1. The Basic algorithm is straightforward. In Step 2, the hash codes for each sub-tree are computed to place them in the respective hash bucket. This step ignores small subtrees, thus implementing the mass threshold in a way that further reduces the number of comparisons required considerably, as the vast majority of trees are small. After that, every pair of sub-trees located in the same hash bucket is compared, if the similarity between them is above the specified threshold, the pair is added to the clone list, and all respective sub-clones are removed.

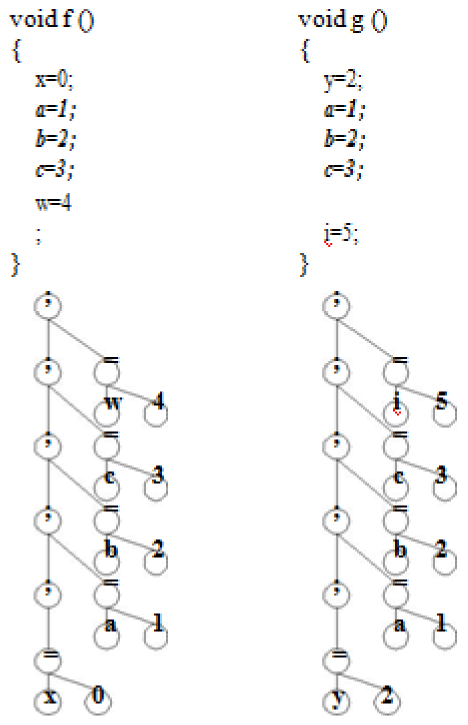
```
1. Clones=□
2. For each subtree i:
     If mass(i)>=MassThreshold Then hash i to bucket
3. For each subtree i and j in the same bucket If CompareTree(i,j) > SimilarityThreshold
        Then { For each subtree s of i If IsMember(Clones,s)
                    Then RemoveClonePair(Clones,s) For each subtree s of j
                    If IsMember(Clones,s)
                    Then RemoveClonePair(Clones,s) AddClonePair(Clones,i,j)
               }
```

Figure 1 - Basic Subtree Clone Detection Algorithm

A minor enhancement to subtree clone detection is the detection of similar trees containing commutative operators such as add (“ +”). The value is in detecting re-ordered operands in “ mental macros” ; in reused-code, it is rare for a programmer to swap operands while editing. Implementing this requires merely that such tree node types be identified as commutative, that the hashing function produces identical values on commutative trees, and that the similarity function tries all child orderings on commutative subtrees.

V. FINDING CLONE SEQUENCES

The preceding section shows how to detect clones as trees, and is purely syntax driven. In practice, we are interested in code clones that have some semantic notion of sequencing involved, such as sequences of declarations or statements. In this section, we show how to detect because of the order in which the parse reductions are done as determined by the controlling grammar rule. In this example, nodes labeled with a " ;" are sequence nodes for statements belonging to a compound statement. Because a generic clone detector has no idea which tree nodes constitute sequence nodes, these nodes must be explicitly identified to the clone detector.

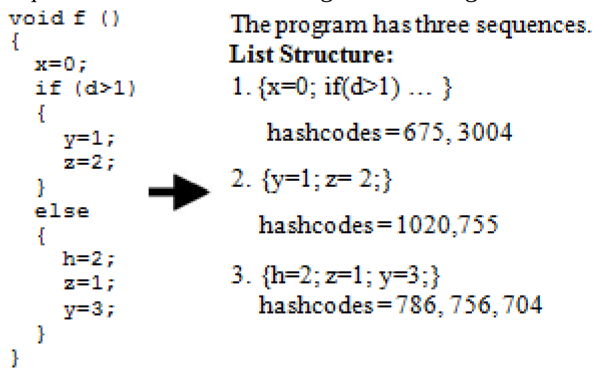


```
void f ()
{
   x=0;
   a=1;
   b=2;
   c=3;
   w=4
   ;
}
```

```
void g ()
{
   y=2;
   a=1;
   b=2;
   c=3;

   i=5;
}
```

Figure 2 – Example of clone sequence

Such sequences of sub-trees are not strictly trees, and consequently require a special treatment. In Figure 2, the Basic algorithm finds three clones corresponding to the assignment statements for variables a, b and c. But, it is unable to detect the clone sequence, because it is not a single sub-tree, but rather a sequence of sub-trees. The sequence detection algorithm copes with this problem by comparing each pair of sub-trees containing sequence nodes, looking for maximum length sequences that statement sequence clones in ASTs using the Basic algorithm as a foundation.



```
void f ()
{
  x=0;
  if (d>1)
  {
    y=1;
    z=2;
  }
  else
  {
    h=2;
    z=1;
    y=3;
  }
}
```

The program has three sequences.

List Structure:

1. {x=0; if(d>1) … }
   hashcodes=675, 3004
2. {y=1; z= 2;}
   hashcodes=1020,755
3. {h=2; z=1; y=3;}
   hashcodes=786, 756, 704

Figure 3 – List structure example

Such sequences show up in ASTs not as arbitrary trees, but rather as right- or left-leaning trees with some kind of identical sequencing operator as root. Sequences of sub-trees appear in AST as a consequence of the occurrence in the dialect grammar of rules encapsulating sequences of zero or more syntactic constructs. These sequence rules are typically expressed by the use of left or right recursion on production rules. When a parser generator produces parsers that automatically generate AST, it is common, as in our case, that the trees have a left-leaning shape.

Consider Figure 2, which shows a pair of short sequences of statements along with their corresponding trees. Note that the left-leaning tree reverses the order of the statements encompasses previously detected clones. Short sequences (especially those of length one) are not interesting sequence clones. A minimum-sequence length threshold parameter controls the minimum acceptable size of a sequence.

To find sequence clones, we build a list structure where each list is associated with a sequence in the program, and stores the hash codes of each sub-tree element of the associated sequence. Figure 3 shows an example of the list structure that is built. This list structure allows us to compute the hash code of any particular subsequence very quickly.

Figure 4 gives the sequence detection algorithm. This algorithm compares each pair of sub-trees containing sequence nodes looking for the maximum length of possible sequencing that encompasses a clone. Whereas the Basic algorithm finds *three* clones in Figure 2, the sequence detection algorithm finds the sequence comprising the assignments for variables a, b and c as a *single* clone. Following the requirement that larger clones subsume smaller ones, detecting this sequence immediately invalidates the clone status of the atomic statements found as clones by the Basic algorithm.

```
1. Build the list structures describing sequences
2. For k = MinimumSequenceLengthThreshold
         to MaximumSequenceLength
3. Place all subsequences of length k
   into buckets according to subsequence hash
4. For each subsequence i and j in same bucket If CompareSequences (i,j,k) >
                SimilarityThreshold
   Then { RemoveSequenceSubclonesOf(clones,i,j,k) AddSequenceClonePair(Clones,i,j,k)
         }
```

Figure 4 – Sequence detection algorithm

The current tool does have a method for detecting near miss sequence clones in which the sequences are of unequal length. This causes us to miss clones in which statements have been inserted or deleted.

VI. GENERALIZING CLONES

After finding exact and near-miss clones, we use another method (Figure 5) to detect more complex near-miss clones. The method consists of visiting the parents of the already-detected clones and check if the parent is a near miss clone too. We also delete subsumed clones. Note that the details related regarding sequence handling have been omitted for clarity.

A significant advantage of this method is that any near-miss clones must be assembled from some set of exact sub-clones, and therefore no near-miss clones will be missed. (Since acceptance of the paper, we have developed a new version of the clone detector that uses only exact clone hashing on small subtrees, sequence detection and this generalization method. This new version has better performance and detects any kind of near miss clones. Details will have to wait for another paper.)

```
1.  ClonesToGeneralize=Clones
2.  While ClonesToGeneralize□ □
3.    Remove clone(i,j) from ClonesToGeneralize
4.    If CompareClones(ParentOf(i), ParentOf(j))
          > SimilarityThreshold
      Then { RemoveClonePair(Clones,i,j) AddClonePair(Clones,
                    ParentOf(i), ParentOf(j)) AddClonePair(ClonesToGeneralize,
                    ParentOf(i),ParentOf(j))
           }
5.  End While
```

Figure 5 – Detecting more complex clones

The detected clone set is the union of sequence clones and the results of the clone generalization process. After all clones were detected, we generate a macro that abstracts each pair of clones. Figure 6 shows an example of near miss sequence clones detected by the tool in the application discussed in the next section. Figure 7 shows the macro generated by the clone detector for the clones in Figure 5. Trivial syntax modifications can turn this into legal C preprocessor directives, and the detected clones could be removed since the tool knows their source.

In the last step, the tool tries to group instances of the same clone in order to provide additional feedback on the number of instances of each clone. The clones are divided in-groups following the first fit approach; i.e. a clone is inserted in the first group where it is a clone of all instances already inserted.

VII. DETECTOR ENGINEERING

To build a practical clone detector one must address several issues:

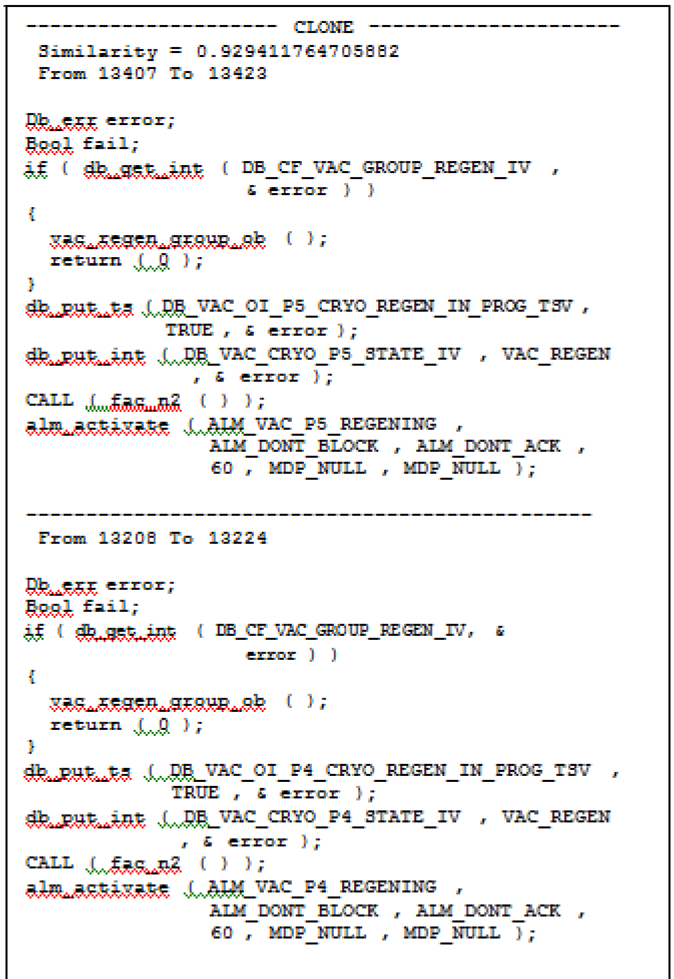
- Parsing and building the AST
- Preprocessor directives
- Reporting results
- Industrial scale source codes

Parsing the program suite of interest requires a parser for the language *dialect* of interest. While this is nominally an easy task, in practice one must acquire a tested grammar for the dialect of the language at hand. Often for legacy codes, the dialect is unique and the developing organization will need to build their own parser. Worse, legacy systems often have a number of languages and a parser is needed for each. Standard tools such as Lex and Yacc are rather a disappointment for this purpose, as they deal poorly with lexical hiccups and language ambiguities. In our Design Maintenance System™ (DMS™) tool [Baxter97] we use a variation of a Tomita-style parser [Tomita91], that can parse ambiguous grammars (such as C++) with impunity, cutting the time to develop a usable parser significantly. In particular, we use the parsing algorithm of [Wagner97], which also conveniently builds the parse tree. However, when using a Tomita-style parser, special nodes called Symbol nodes, representing ambiguous parses, must be taken into account in the clone detection algorithm. We ignore ambiguous sub-trees by choosing a canonical son based on grammar rule number.

Preprocessor directives always complicate processing source files. For the sample system in C++, we handled them in various ways:

- Added preprocessor directives to the language syntax, effectively extending the language to include well-structured directives;
- Ignored (processed) INCLUDE directives, and concatenated INCLUDE files onto the input; and
- Ran the preprocessor to get rid of " unstructured" directives.

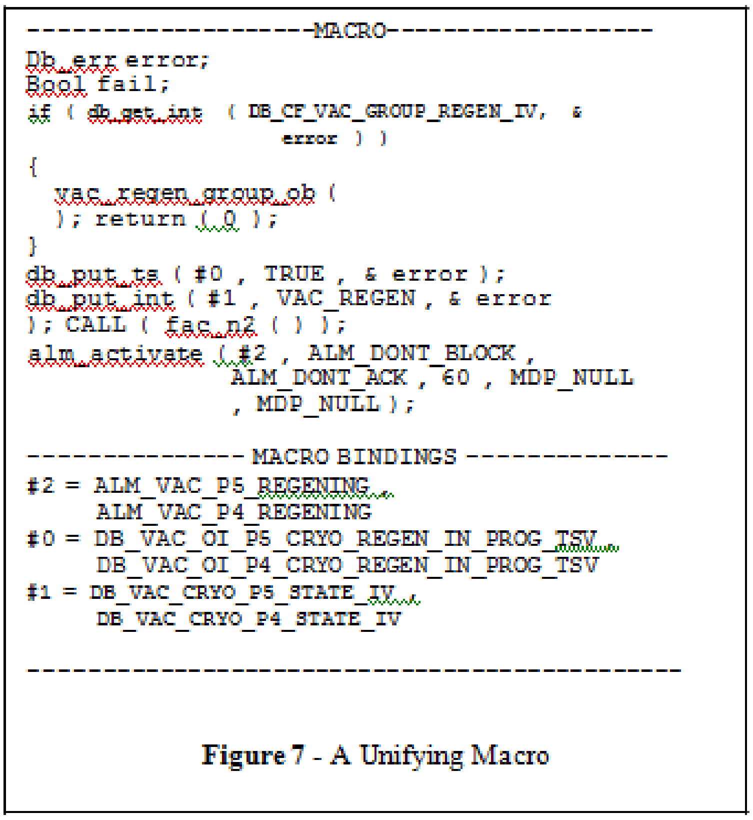
When detecting clones in the presence of INCLUDE files, the included text must be treated as part of the include file, not part of the including file.



```
--------------------- CLONE ---------------------
 Similarity = 0.929411764705882
 From 13407 To 13423

Db_err error;
Bool fail;
if ( db_get_int  ( DB_CF_VAC_GROUP_REGEN_IV  ,
                   & error ) )
{
  vac_regen_group_ob  ( );
  return ( 0 );
}
db_put_ts ( DB_VAC_OI_P5_CRYO_REGEN_IN_PROG_TSV ,
           TRUE , & error );
db_put_int ( DB_VAC_CRYO_P5_STATE_IV  , VAC_REGEN
             , & error );
CALL ( fac_n2  ( ) );
alm_activate  ( ALM_VAC_P5_REGENING  ,
               ALM_DONT_BLOCK , ALM_DONT_ACK  ,
               60 , MDP_NULL , MDP_NULL );

-----------------------------------------------
 From 13208 To 13224

Db_err error;
Bool fail;
if ( db_get_int  ( DB_CF_VAC_GROUP_REGEN_IV,  &
                   error ) )
{
  vac_regen_group_ob  ( );
  return ( 0 );
}
db_put_ts ( DB_VAC_OI_P4_CRYO_REGEN_IN_PROG_TSV  ,
           TRUE , & error );
db_put_int ( DB_VAC_CRYO_P4_STATE_IV  , VAC_REGEN
            , & error );
CALL ( fac_n2  ( ) );
alm_activate  ( ALM_VAC_P4_REGENING  ,
               ALM_DONT_BLOCK , ALM_DONT_ACK  ,
               60 , MDP_NULL , MDP_NULL );
```

Figure 6 – Sequence clones found (from Application)

Program scale is a problem for any clone detection scheme. For our experiment, we were limited to 100,000 line chunks in 600Mb RAM because of artificially large memory-based data structures in our prototype DMS. One unavoidable problem is the retention of comments to enable re-printing of detected clones in their exact source form; most conventional lexers throw comments away as being " useless" whitespace. We believe one can perform clone detection on some 10 million lines in 2Gb with careful design and one tree node per processor cache line. A disk-based implementation could handle larger size, but batch performance is likely to be much worse because one would replace RAM-based hash bucket access and tree walking times with disk I/O times. A reasonable possibility is to batch compute once, and then incrementally compute the hash codes and do clone detection for changed modules.



```
--------------------MACRO-------------------
Db_err error;
Bool fail;
if ( db_get_int  ( DB_CF_VAC_GROUP_REGEN_IV,  &
                       error  ) )
{
  vac_regen_group_ob (
  ); return ( 0 );
}
db_put_ts ( #0 , TRUE , & error );
db_put_int ( #1 , VAC_REGEN , & error
); CALL ( fac_n2 ( ) );
alm_activate ( #2 , ALM_DONT_BLOCK ,
               ALM_DONT_ACK , 60 , MDP_NULL
               , MDP_NULL );

--------------- MACRO BINDINGS --------------
#2 = ALM_VAC_P5_REGENING ,
     ALM_VAC_P4_REGENING
#0 = DB_VAC_OI_P5_CRYO_REGEN_IN_PROG_TSV ,
     DB_VAC_OI_P4_CRYO_REGEN_IN_PROG_TSV
#1 = DB_VAC_CRYO_P5_STATE_IV ,
     DB_VAC_CRYO_P4_STATE_IV

---------------------------------------------
```

Figure 7 - A Unifying Macro

VIII. CLONE DETECTION APPLIED

The clone detector was applied to the source code for a process-control system having approximately 400 KSLOC of C code. The system was created 7 years ago by reusing and modifying a then 3-year-old code of a system with similar functionality. 15 programmers presently maintain it. The programmers who did the port were not those who developed the original system. The software is partitioned into subsystems according to function.

Figure 8 shows the percentage of cloned code in 19 different subsystems, computed as the ratio of redundant SLOC and subsystem SLOC. Three subsystems have approximately 28% cloned code. (Subsystem 3 is believed to be an erroneous data point because the clone detector at present erroneously reports overlapping sequences of self-similar code as clones). Subsystem 15 was created by cloning device driver code capable of handling one I/O port to allow handling of a second port. Done under schedule pressure, it was already known to be highly redundant. We observe that schedule pressure has also prevented this redundancy from ever being fixed, and the clone tool could do this almost automatically. Subsystem 16 is code that services a number of devices and was created similarly to subsystem 15, as confirmed by the developers. The average clone percentage is 12.7% for all subsystems.

If software maintenance costs were distributed evenly across source, this suggests 12% savings in maintenance costs. In practice, some systems are more troublesome than others. Subsystem 2 is difficult to maintain and accounts for a large percentage of defects written against the system. A reduction in code here could have disproportionate savings in costs.

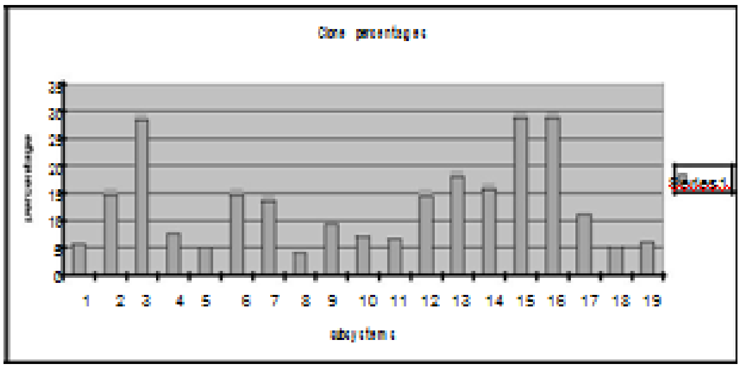


Figure 8 –Clone percentages

An interesting question is how often code is copied, and how much is copied. Figure 9 shows the number of times that clones of various sizes (SLOC) were found. The three largest subsystems are represented in this graph. Clone sizes larger than 25 are rare and are therefore not shown. However, within subsystem 15 an unexpected clone of size 497 was found! We conclude that most clones are relatively small in size, on the order of 10 SLOC.

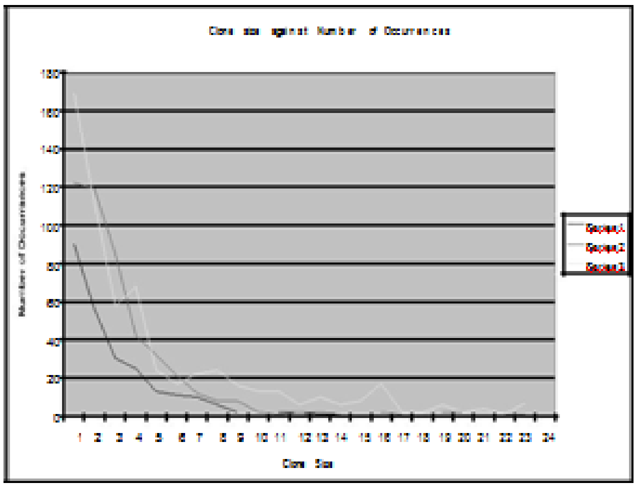


Figure 9 – Clone size against number of occurrences

One hopes that as a system matures, it becomes better organized. Figure 10 shows the changes in percentage of clones for four fairly volatile subsystems over four releases with a periodicity of approximately 6 months. Three of the four subsystems show a steady improvement in clone density, but one shows a sharp increase. All could clearly use a clone removal pass.

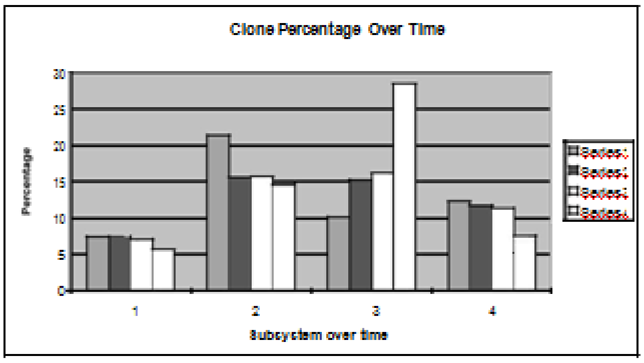


Figure 10 – Clone percentage over time

We had hypothesized that larger files might have larger clone percentages, because of opportunity and complexity. Figure 11 compares clone percentage with subsystem size in KSLOC. Although the 3 largest files have the largest number of clones, no pattern is recognizable. We conclude that clone ratios are relatively independent of subsystem size.

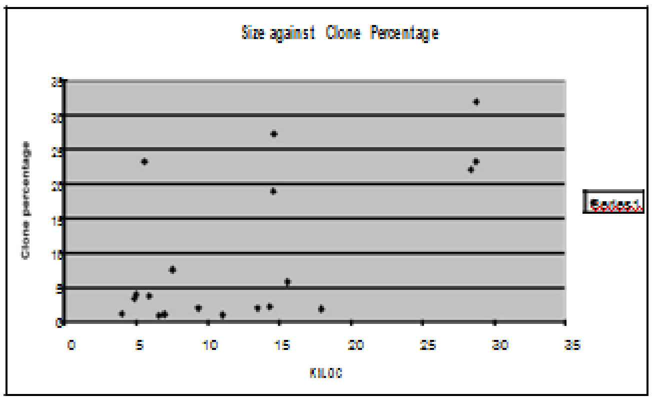


Figure 11 – Size against clone percentage

We draw the following conclusions from the data:

1. Higher percentages of cloned code were found in newer subsystems, which might be expected to be freer of clones because they are closer to a clean, original design. However, it appears that cloning is used to get the code to work, the code is released as soon as it is working, and then improved as it stabilizes. [Lague97] also found this.
2. We examined the clones in the code for subsystems having higher clone percentages. Many clones were separate functions performing identical operations on different data types or devices having left versus right symmetry. This appears to have been a stylistic choice, perhaps made in the interests of apparent modularity or clarity (although the cost of maintaining all those copies argues against actual modularity). It is interesting to notice that subsystems

having a greater degree of this stylistic copying are also the subsystems which change most infrequently, largely low-level utility programs (data access, data initialization, etc). This supports finding 1) in that utility code is often stable, therefore rarely modified and therefore not 'cleaned up'.

3. There is no correlation between subsystem size and clone percentage. One might expect to have more clones in larger subsystem because of the conceptual complexity of grasping the entire subsystem. It appears that deadline pressures, similarities in functionality, and programmer style have more affect on clone density than code size.

IX. CLONES AS DOMAIN CONCEPTS

We believe that a clone detector tool can be of great value for a domain analyst capturing domain concepts from the source code of similar systems. A key issue in domain analysis is the need to understand commonalities in concepts and implementations across families of software systems. As defined earlier, idioms are program fragments that implement specific concepts in a given application domain. It is reasonable to believe that some of the clones found in a system are realizations of these concepts, precisely because they are in repeated demand for carrying out the application. Such understanding can be used as an interesting starting point for the abstraction of domain concepts from source code [DeBaud97].

A domain-analysis process step might include running a clone analysis for each of several similar systems, followed by interviews with domain specialists. During these interviews, the cloned code is shown and the specialists are asked to provide possible abstractions motivating such implementations. The results would be domain concepts and their abstract implementations as generalized from the clone instances. The tool presented in this paper represents a step forward in the development of auxiliary tools for the hard task of finding commonalties in similar systems.

```
device.mode = MOVE_ABS; device_xf.position = (int) db_get_int(
                    DB_XFER_ARM_LOAD_IV, &dbstat); move_swap (& device_xf);
data = (char *) & device_xf; data_size = sizeof(Msg_device_move); msg_send((Process_id)
DI_DEVICE_XCR,
                    data ,(Uint16) data_size, MSG_NO_FLAGS , MSG_XCR_DEVICE_XFER_MOVE);
printf("Xfer Arm rotated to load \n");
```

Figure 12 - A domain idiom found by clone detection

An example of a clone representing a domain idiom is shown in Figure 12. These seven lines were repeated eight times within one source file. Here the control system is commanding an arm, which transfers product through the system, to move to the load position, a basic command for that device.

X. FUTURE WORK

The most obvious next step is to automate the removal of detected clones from the source (Since paper acceptance, we have done this). The organization maintaining the example application code currently has high software engineering costs, and appears motivated to carry this process out manually, given the clone detector shows the replacement macros. A follow on step might be to insert the clone detector into the software engineering check-in process to eliminate clones as they arise.

There are a number of improvements one could make to the clone detector itself, including performance, the use of DAGs and/or dataflow graphs instead of trees and operations.

The current detector uses some 600Mb RAM and 120 minutes of CPU time to compute exact and near miss clones for 100K SLOC code. The space demand is determined essentially by the size of the abstract syntax trees and the hash tables. The AST node size in our implementation could be reduced by a factor of 4, and removing ambiguities in the C++ grammar that we used to parse the C code could reduce the demand by another estimated 30%, reducing the storage costs to 100Mb. Much of the clone detection is done by comparing sub-trees, and is easily parallelizable. Since our clone detector is sequentially coded in a parallel language, PARLANSE™ , this should be easy to accomplish. We expect a nearly linear speedup to the 8 processors available on commercial servers, reducing the estimated runtime to 15 minutes.

At present, the detector uses thresholds to eliminate comparisons of small trees. This prevents it from detecting near miss clones in which one clone instance is small, and the other is large. We would like to remove this limitation.

The algorithms we presented here will work more effectively on DAGs, with no change, than on trees. For DMS, we plan to convert identifiers in tree leaves into cross-links to the syntactic construct defining that identifier, forming DAGs from the parse trees. This has the advantage of preventing false-positive exact clones using accidentally the same textual name, but which actually refer to different identifiers according to the language scoping rules (Near miss clones that are parameterized by these non-identical names will still get found). Type information could be used to prevent detection of accidental clones, as many such clones use different types in their computation. In order to properly build DAGs, it is required that preprocessor primitives, especially INCLUDE directives, be handled in a structured fashion; arbitrary " editing" of source with a preprocessor other prevents correct type analysis and name resolution.

What one would often like is the identification of two blocks of code that were cloned, and then patched differently. If the patch is the replacement of one language construct by another, our detector will find it. If the patch is the insertion of new code in the middle of a clone, forming a " split clone" , our present detector will not be able to find the entire clone, but can identify the preceding and following fragments as clones. It would be convenient if the detector identified such split clones, as they are often an indication that a clone was defective, and only one instance was fixed.

A program representation in which control and data flows are explicit would allow a more semantic-oriented clone detector that would be insensitive to variable names and statement ordering. In particular, this would detect, as monoliths, clones that have been split by irrelevant insertions. It would also not be fooled by reordered data declarations. For DMS, we will construct such representations, and so should be able to build more sophisticated detectors in the future.

XI. RELATED WORK

A fast string-based method, DUP, for detecting exact clones and simple near-miss clones, was considered by [Baker95], who reported 13-20% clones, for a large application (1M SLOC). When producing clones, programmers often change white spacing (blanks, tabs, newlines) and comments, which will disable recognition based purely on strings. A simple lexical processor can overcome this particular problem. DUP actually compares strings of lexemes rather than strings of characters to combat this problem.

Their lexeme-based algorithm seems easy to implement and operates nicely on scale, but has some shortcomings. First, it fails to detect clones which differ by other than a trivial substitution of one lexeme for an identifier. Second, based on the description, we believe that such a detector will also falsely detect nonsensical clones, such as the lexeme sequence frequently found in C programs at the end of a function: " **}; int SomeName(**" Furthermore, DUP's matching process cannot detect exact clones with commutative operators. An easy way to cure these problems is to take the language syntax into account, as we do. Our detector finds clones, which differ in complex language constructs, such as expressions or statements. In any case, more sophisticated clone detection, using name resolution, cannot be accomplished without parsing and applying deeper knowledge of the language scoping rules.

The method used by [Lague97] apparently parses the program text, and then computes a hash code for function bodies by forming a vector of predefined software metrics (e.g. SLOC, Halstead, etc. While this can be used for recognizing function clones, it appears to make the hashing function unnecessarily complicated without necessarily providing a good hash function. Implementing and running the clone detection process is not likely to be nearly as fast as a straightforward implementation like ours, which is important for the scale in which clone detection is interesting. We remark that comparing just function bodies, however, cuts the volume of comparisons required, so perhaps this is not a problem if one wants to limit clone detection to just function bodies. Furthermore, it is limited to language constructs on which all the metrics apply, hinting at the reason it operates only on function bodies. Such a scheme probably would be ineffective for detecting data declaration clones (which could be removed in " C" by using TYPEDEFs). If a wide variety of metrics were used, it would be likely parsing is necessary to support one of them, and our scheme could be used directly instead. If parsing is not required, then lexical differences may damage clone detection ability as already suggested. Finally, one has to augment simple hashing schemes to reasonably detect near misses, or sequence clones. Our method will detect cloned expressions, statement sequences, data declarations, etc., anything which has a syntactic structure in a language.

Neither Baker nor Lague constructed macros to enable clone removal.

XII. CONCLUSIONS

A practical method for detecting near-miss and sequence clones on scale has been presented. The approach is based on variations of methods for compiler common-subexpression elimination using hashing. The method is straightforward to implement, using standard parsing technology, detects clones in arbitrary language constructs, and computes macros that allow removal of the clones without affecting the operation of the program. We applied the method to a real application of moderate scale, and confirmed previous estimates of clone density of 7-15%, suggesting there is a " manual" software engineering process " redundancy" constant. Automated methods can detect and remove such clones, lowering the value of this constant, at concomitant savings in software engineering or maintenance costs. Clone detection tools also have good potential for aiding domain analysis.

We expect to report on more advanced clone detection and removal methods in the near future.

REFERENCES

[1] [Aho86] Alfred Aho, Ravi Sethi and Jeffrey Ullman, **Compilers, Principles, Techniques and Tools**, Addison-Wesley 1986.

[2] [Baker95] Brenda Baker, On Finding Duplication and Near-Duplication in Large Software Systems, Working Conference on Reverse Engineering 1995, IEEE.

[3] [Barson95] P. Barson, N. Davey, S. Field, R. Frank, D.S.W. Tansley, Dynamic Competitive Learning Applied to the Clone Detection Problem, Proceedings of International Workshop on Applications of Neural Networks to Telecommunications 2, 0-8058-2084-1, Lawrence Erlbaum, Mahwah, NJ 1995

[4] [Baxter97] Ira Baxter and Christopher Pidgeon, Software Change through Design Maintenance, International Conference on Software Maintenance, 1997, IEEE.

[5] [DeBaud97] Jean-Marc DeBaud, DARE: Domain-Aumented ReEngineering, Working Conference on Reverse Engineering, 1997, IEEE.

[6] [Kontoginnis96] K. Kontogiannis, R. DeMori, E. Merlo, M. Galler, and M. Bernstein, Pattern Matching for Clone and Concept Detection, Journal of Automated Software Engineering 3, 77-108, 1996, Kluwer Academic Publishers, Norwell, Massachusetts

[7] [Johnson94] J.H. Johnson, Substring Matching for Clone Detection and Change tracking, Proceedings of the International Conference on Software Maintenance 1994, IEEE.

[8] [Johnson96] H. Johnson, Navigating the Textual Redundancy Web in Legacy Source, Proceedings of CASCON '96, Toronto, Ontario, November 1996.

[9] [Lague97] B. Lague, D. Proulx, E. Merlo, J. Mayrand, J. Hudepohl, Assessing the Benefits of Incorporating Function Clone Detection in a Development Process, International Conference on Software Maintenance 1997, IEEE.

[10] [Tomita91] **Generalized LR Parsing**, Masaru Tomita ed., 1991, Kluwer Academic Publishers, Norwell, Massachusetts.

[11] [Wagner97] Tim Wagner and Susan Graham, Incremental Analysis of Real Programming Languages, Proceedings 1997 SIGPLAN Conference on Programming Language Design and Implementation, June 1997, ACM