Software Code Clone Detection using AST

G. Anil kumar[#1] Dr. C.R.K.Reddy[*2] Dr. A. Govardhan[#3]

[#1]research Scholar Jntuh & Sr.Asst. Prof Cse Mgit, Hyderabad
[*2]prof. of CSE CBIT , Hyderabad

[#3]director & Prof. of CSE, Sit, JNTUH, Hyderabad

Abstract— The research which exists suggests that a considerable portion (10-15%) of the source code of large-scale computer programs is duplicate code. 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 lexemes, 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 500K 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

The data from the previous work says that duplicated code is a considerable fraction i.e.10-15% of the source of large computer programs[2][9]. Adhoc reuse is routinely performed by programmers by brute-force copying code fragments. These enable implementation of actions similar to their current need and helps in performing a cursory customization of the copied code to the new context.

The act of copying suggests that the programmer has the intention to reuse the implementation of some abstraction. Encapsulation which is the software engineering principle is broken by the act of pasting. The unstructuredness and commonness suggest that programmers should be offered tools which enable them to make use of implementations of abstractions without breaking encapsulation.

There can be a decrease in the cost of the software maintenance apart from the reduction in code size, if in-lined procedure calls, macros or other equivalent short hand methods are used to detect and replace redundant code. The present software engineering focuses on how to find small percentage process gains. This is a mechanical method inorder to achieve upto 10% savings.

A program fragment which implements a recognizable concept i.e. data structure or computation is an idiom. A fragment is a clone. A fragment which is nearly identical to another is a near miss clone. When an idiom is optionally edited and copied, clones occur with the production of exact or near miss clones.

Clone detection is not only helpful in producing more structured code but also in discovering domain concepts and their idiomatic implementations.

There were limitations to clone detection, previously clone detection used to detect textual matches or near miss clones only on complete function bodies. This paper enables practical methods with the help of abstract syntax trees. This helps in the detection of exact and near miss clones for the arbitrary fragments of program source code. By using conventional transformational methods clones can be factored out of the source code. This becomes feasible since detection is in terms of the program structure.

A tool which uses these detection techniques is applied to a java production software system of some 500k SLOC and the results showed detection levels of duplication found by previous work.

A variation of the well-known compiler method is used by the tool for the detection of common sub-expressions [1]. This enables exact tree matches essential to detect clones in the context of commutative operands, near misses and statement sequences.

II. OCCURRANCES OF CLONES

Software clones appear for many reasons:
1. Code reuse by copying existing codes.
2. Coding styles of the programmers.
3. Instantiations of definitional computations.
4. Failure to identify/use abstract data types.
5. Performance enhancement.

6. Accidental clones.

The design processes and formal reuse methods are structured by the state of the art software design. Less structured means are used to construct legacy code. New functionalities are implemented by programmers to find some code idiom. The idiom performs a computation which is identical to the one desired. It also copies and modifies the idiom. The ubiquity of this event is hastened by the functions of copy and paste.

Inorder to produce modules which have different variants, this method is used in large systems. When device drivers are built for operating systems, maximum code is boiler plate. The only part which needs to be changed is the device hardware of the driver. Usually, a device driver author copies an entirely an existing well-known, trusted driver and he will modify it simply. Generally, it is a good reuse practice. However, it exacerbates the problem of maintenance of fixing a bug found in the trusted driver by code replicating over many new drivers.

At times, a style for coding a regularly needed code fragment will arise like error reporting or user interface displays. Inorder to maintain the style, the fragment is copied. When this is done the fragment consists only of parameters. However, this is a good practice. Sometimes, the fragment unnecessarily contains more knowledge of some program data structure, etc.,

The repeated computations are simple and definitional. As a result, even if copying is not used, a programmer can use a mental macro to write. When he writes, the same code needs to be carried out. If at all if there is frequency in mental operation, he may develop a regular style of coding it. Near miss clones are produced by mental macros. The code has irrelevant order and variable names.

Some clones have complete duplicates of functions. These are intended to be used on another data structure of the same type. There are many systems with poor copies. They have insertion sort on different arrays scattered around the code. All such clones give an indication that the data type operation should be supported by reusing a library function instead of a pasting a copy.

There are justifiable performance reasons for which some clones exist. Systems which have tight time constraints are often handoptimized by replicating frequent computations. This is done when a compiler does not offer in living of arbitrary expressions or computations.

Finally, there are occasional code fragments that are accidentally identical, but actually they are not clones. When proper investigation is done, such clones are not intended to carry out the same computation. The number of accidents of this type will come down dramatically, as the size goes up.

When the accidental clones are ignored, the mass of the code increases because of the presence of clones in code unnecessarily. This compels programmers to inspect more code than necessary, and as a result the cost of software maintenance increased. Such clones can be replaced by invocations of clone abstractions, when the clones can be found with potentially great savings.

III. METHODOLOGY

A. Clone Detection Using ASTs

The discovery of code fragments which compute the similar result is the basic problem in clone detection. For this, first the program in parts must be fragmented before comparison. Then, it has to be determined as impossible, two arbitrary program fragments halting under the same circumstance is not determined. Hence, it is impossible theoretically to finalise that they compute identical results. The deep semantic analysis which is conservatively bounded by time limits is acceptable for equivalence detection as false negatives are unavoidable. There will be infrastructure requirement in the form of semantic definitions, theorem provers etc., Practically, detection of complete semantic equivalence should be given up because many clones evolve due to copy and paste editing processes.

When false positives are not produced simpler definitions of equivalent code may suffice. This denotes that clone detection can be done using more syntactic methods. The source lines of code can be compared. It is assumed that the cloning process has not introduced any changes as per source line equality. Clone detection is limited to exact matches without any changes in identifiers, comments, spacing, or other non –semantic changes. As a result, it fails to trace near miss clones. Hence, a practical possibility would be to compare program representation in which control and data flows are explicit, closer to full semantics.

The building of transformational tools in order to modify large software systems is semantic designs [4]. As a first step, such tools typically parse source programs into ASTs before transformation. The comparison of syntax trees is chosen for investigation because of the early product state of our tools. This had the advantage of directly avoiding uninteresting changes at the lexical level.

There are some steps in the process of clone detection. First, the source code is parsed and an AST is produced for it. Next, three main algorithms are applied to find clones. The purpose of the basic algorithm, which is the first algorithm, is to detect sub –tree clones. The second algorithm is sequence detection algorithm. This algorithm helps in the detection of variable –size sequences of sub –tree clones. This is helpful in the detection of statement and in the declaration of sequence clones. The third algorithm attempts to generalize combinations of other clones and looks for more complex near miss clones.

Clone removal is not carried out, but the remaining detected clones can be printed.

B. 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 [1]. 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 subtree 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 algorithm 1. The Basic algorithm is straightforward. In Step 2, the hash 4 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.

The algorithm is shown in the algorithms section Basic *Sub-tree Clone Detection Algorithm.*

C. 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 statement sequence clones in ASTs using the Basic algorithm as a foundation.

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 subtrees 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 1, 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 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 i ()          void j ()
{                  {
p=0;               y=2;
q=1;               q=1;
r=2;               r=2;
s=3;               s=3;
w=4;               h=5;
}                  }
```

Fig. 1 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 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.

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

The program has three sequences.

List Structure:

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

Fig. 2 Example of list structure

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 2 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.

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 1, 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.

The algorithm is shown in the algorithms section's *Sequence detection algorithm*.

D. Generalization

After finding exact and near-miss clones, we use another method (Figure 5) to detect more complex nearmiss 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 sub trees, sequence detection and this generalization method. This new version has better performance and detects any kind of near miss 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 5 shows an example of near miss sequence clones detected by the tool in the application discussed in the next section. Figure 5 shows the macro generated by the clone detector for the clones in Figure 5. Trivial syntax modifications can turn this into legal C pre-processor 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.

The algorithm is shown in the algorithms section as Detecting more complex clones.

IV. ALGORITHMS

Basic Sub-tree Clone Detection Algorithm

compare every subtree to every other sub-tree for 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. Rather than comparing trees for exact equality, we compare instead for similarity, using a few parameters

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. 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.

Detecting more complex clones

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.

V. RESULTS AND DISCUSSIONS

The clone detector tool was applied to a process control system having java code. The figure 3 shows the overview of our tool which takes the files (java only) as inputs. The results are shown in detail after the processing of our tool in the region provided below the processing space.

The figure 4 shows how the tool will represent and show the result of a particular detection between 2 java source files. The coding is in such a way that the ASTs will be automatically built in the tool, processed and the line by line detection of the clones will be produced in the space of the tool as shown in the figure.

Here we have applied the tool to the files of type I and II. The type I clones are the clones which are exact to each other in every means i.e the both clones are mirror images of each other. These are very easy to detect through our tool which constructs the ASTs and compares the both clones. The type II clones are the codes which nearly similar to each other i.e near miss clones. To use the tool we simply click on the browse button on the tool and select our destination file, then click on the process button to detect the clones in our source files. The results are shown in the fig.4.Thus we have applied our tool on the java platform and in the future we could also go for comparison and detection of type III and type IV clones.

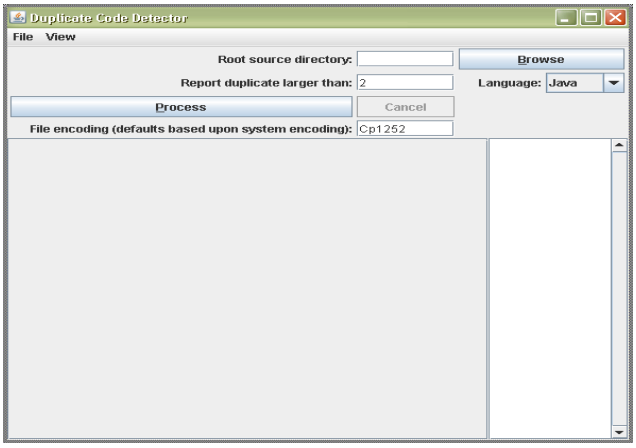

Fig. 3 The clone detector tool

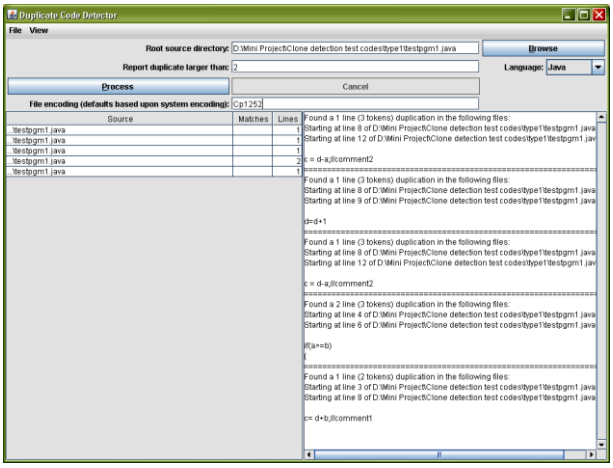

Fig. 4 clone detection applied

VI. CONCLUSIONS

The

References

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

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

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

[5] Jean-Marc DeBaud, DARE: Domain- *Augmented Reengineering*, Working Conference on Reverse Engineering, 1997, IEEE.

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

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

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

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