Plagiarism Detection in Software Designs

B. Belkhouche[*]
EECS Department
Tulane University
New Orleans, LA 70118
bb@eecs.tulane.edu

Anastasia Nix
EECS Department
Tulane University
New Orleans, LA 70118
nix@eecs.tulane.edu

Johnette Hassell
EECS Department
Tulane University
New Orleans, LA 70118
hassell@eecs.tulane.edu

ABSTRACT

Detecting plagiarism in software is a computationally complex process. At the same time it is critical, for the lack of a deterrent through detection may result in various losses. Several systems to detect plagiarism have been proposed. However, their lexically-based analysis is not powerfull enough and can be foiled with minimal efforts. To address their shortcomings, we have devised a detection framework with the following salient features: (1) designs, instead of code, are compared; (2) multi–level abstractions of the design are generated; and (3) comparison follows a stepwise process according to the abstraction levels. A comparison with existing systems shows that this strategy results in simpler algorithms and more accurate analyses.

1. INTRODUCTION

The problem of code theft and software plagiarism has plagued the software industry and academia for years. Oftentimes code is copied wholesale, without even changing any variable names in the program. As software has become more and more pervasive, illegal copying has become more widespread. Consequently, billions of dollars are lost annually. Preventive measures and laws are being implemented to discourage software piracy. Simultaneously, manual and automated systems to detect copies are being developed. In fact, this endeavor was triggered very early within universities in an effort to thwart students from cheating on their programming assignments. The automated systems relied on various strategies to detect similarities between copies. A simple strategy consists of counting the numbers of unique operators/operands and the total numbers of these in a program [1]. The resulting counts are then compared for similarities. A refinement of this approach was subsequently proposed [2]. It took into account the numbers of each type of statement in a program. It also performed some data analysis based on assignment statements. These systems relied mostly on lexical items making them inadequate to capture structure and contexts. Still a further refinement included as part of its strategy the longest string matching. This system, named Plague, outperformed several other comparison methods [3]. An improvement in the efficiency and accuracy of Plague resulted in the YAP system [4]. YAP succeeded in finding as many matches out of a group of programs as Plague, was equally efficient, and proved to be easier to port to other languages. Another similar system known as Moss is used to find plagiarism in students' assignments [5]. Moss uses a combination of preprosessing to remove comments and unify identifiers, token counting, and string comparison. It is able to give a percentage of matching code and to show the longest matching region.

This work is also motivated by Nimmer's discussion of determining software copyright infringement in which he describes a three step examination of the software [6]. The steps are abstraction, filtration, and comparison and are applied to data structures as well as procedural code. The approach described herein provides for the abstraction of both data and program logic. Further, the abstraction of program logic is multi-level and provides for comparisons at both macro and micro levels. This approach also provides filtration of all syntactic information and filtration on many other aspects of program logic such as "read before you write". The work of an expert called to determine whether two programs/systems are substantially similar, as specified in the law, is greatly reduced because of the ability to automate a large part of the process. In the case of an expert called to testify, the replicatabilty of the abstraction and filtration process answers potential Dubert challenges.

The accuracy and efficiency of detecting similarities depend strongly on the level of the linguistic analysis. At one end of the spectrum is lexical analysis. Such an analysis involves token classification and token counting. Comparison is reduced to aggregate numbers that are stripped of any context. Because of its simplicity and its ease of implementation, it is adopted by most approaches. At the other end resides semantic analysis, the goal of which is to compare functionality. Unfortunately, this is not computationally feasible. Thus, accuracy and efficiency conflict with each other. A more accurate, and meaningful, comparison requires a deeper analysis, which, in turn, leads to higher complexity. To address these issues effectively, other strategies and heuristics must be adopted to warrant any substantial improvement in software comparison. Our approach is to break away from the linear/textual methods of comparison proposed so far. We accomplish this task in

[*] Corresponding author

ACM Southeast Conference '04, April 2-3, 2004, Huntsville, AL, USA
Copyright 2004 ACM 1-58113-000-0/00/0004. $5.00.

two stages. First, we abstract the software by translating the code into its corresponding graphical design, and then we perform multiple comparisons between the designs. By choosing the design as the representation, we are able to capture essential characteristics of the software under scrutiny and thus focus the comparison on these characteristics.

The structure of this paper is as follows. Section 2 describes the plagiarism detection process. There, we describe how design are generated from C programs and explain in detail our comparison technique. Section 3 introduces an example to illustrate the comparison process and its results. Section 4 compares the behavior of our system and other systems.

2. PLAGIARISM DETECTION PROCESS

The plagiarism detection algorithm takes two C programs as input and performs the following operations:

- Design generation: a C program is transformed into a corresponding structure chart design (reverse–engineering).
- Region delineation: a structure chart is partioned into strongly–coupled regions.
- Abstract comparison: the structures as defined by the partitions are compared for similarities.
- Micro comparison: corresponding regions and nodes are compared for similarities.
- Data dictionary comparison: entries in the data dictionary are compared for similarities.

2.1 Design Generation

The design generation process takes a C program and translates it into a structure chart. A structure chart is a graphical design method used to express algorithms at a fairly abstract level. A complete structure chart design consists of two components: a tree representing the algorithm and a data dictionary (symbol table) representing the variables and data structures used in the algorithm. The root of the tree specifies the header of the algorithm. Children nodes specify statements and control structures. The order of the nodes from left to right captures the statements sequencing order. Figure 1 shows a structure chart example, and Table 1 shows the data dictionary.

Rather than manipulating the source code directly, we translate it into its abstract design form. From the source code (C programs in this case), the translation process creates a parse tree and a symbol table. The parse tree is then mapped into an internal binary tree representation of the corresponding structure chart. To a certain extent, this process can be viewed as transforming a linear (textual) representation into a two–dimensional (graphical) representation. As a final step, the structure chart tree is encoded in the DaVinci format to be displayed in the DaVinci graphing program [7]. This tools takes our encoding and generates the actual graph.

2.2 Region Delineation

The structure chart is divided into three types of regions: groups of sequential statements, repetition segments and selection segments. A region must have at least one statement and up to as many statements as are in the chart. The partition is done recursively, so initially the regions will be the largest groups of control flow structures. Then these regions are divided into smaller regions and so on, until every region has only one control structure or a sequence of nodes with no possible control flow changes. Identifying regions is similar to the process of identifying basic blocks in dataflow analysis [8].

The partioning of the structure chart into regions is in fact an abstraction mapping whose input is the detailed structure chart and whose output is the abstract region tree. This abstraction process groups together strongly–coupled statements into one piece (region), and thus, abstracts details of the design. For example, the following set of statements

```
while (x < 0)
   {y = 5;
    x = x + 1;
    while (y > 0) y = y - 1;
   }
```

is initially alloted one region for the outside while statement. After all of the recursive steps, this code would consist of 4 regions: (1) A looping region for while($x < 0$) (R1). (2) A sequential regions for the first two statements inside that loop (R2). (3) A looping region for while($y > 0$) (R3). (4) A sequential region for the statement $y = y - 1$ (R4).

The structure of the corresponding region tree has region 1 as the root, region 2 as its child, region 3 as 2's sibling and 4 as 3's child.

2.3 Abstract Comparison

Root nodes in each region subtree are assigned a given type according to their syntactic category (repetition, selection, sequencing, I/O, etc.). At this stage, the goal is to find whether the charts are conceptually the same. If the control structures are basically similar, comparison is continued. Otherwise, we can reasonably rule out the possibility of plagiarism without having to go through the effort of the detailed comparison.

Region comparison involves generating an ordered sequence of regions at each level starting from the top. The ordered sequence captures the types of regions, the number of occurrences, and the order of occurrence. For example, in the first iteration, from the chart shown in Figure 3 we get the sequence $S_1 \;=\; < A1 : (\mathit{sequence}, 1), A2 : (\mathit{selection}, 1), A16 : (\mathit{sequence}, 1), A17 : (1, \mathit{selection}) >$, and from the chart shown in Figure 4 we get the sequence $S_2 \;=\; < B1 : (\mathit{sequence}, 1), B2 : (\mathit{selection}, 1), B16 : (\mathit{sequence}, 1), B17 : (\mathit{selection}, 1), B22 : (\mathit{sequence}, 1) >$. The premise is that, when copying a program, the overall structure (i.e., the logic) of the program is maintained in the two charts, and that major modifications in the logic are unlikely. The next step in the abstract comparison is to find the longest sequence of matching regions at each level. This consists in traversing the two sequences in locksteps and comparing the corresponding elements. A mismatch stops the search. The longest subsequence is then used to generate the next level sequences and the process is repeated until all levels are exhausted. The length of the matched sequences is used to determine a similarity percentage for this category. A skeleton algorithm for this stage is:

1. Recursively consider level i for tree 1 and tree 2.
2. Build the sequence S_1 of regions in tree 1 and S_2 of regions in tree 2.

3. Compare S_1 and S_2.

4. Generate the longest matching sequence from S_1 and S_2.

If the similarity percentage is below a certain threshold, say below 50%, the charts are considered not similar and comparison stops. To continue the process would result in an exponential explosion.

The final step in the abstract comparison concentrates on node types. Using the sequences generated previously the types of the corresponding nodes pairs are compared. If equal, an equal counter is incremented. The comparison stops after all the nodes are visited. Again a similarity percentage is computed. The skeleton algorithm for this stage is:

1. Consider two corresponding sequences S_1 and S_2.

2. While traversing them in lockstep consider nodes (N^i_{s1},N^j_{s2}).

3. Increment count if their types match

4. Next pair.

2.4 Micro Comparison

The comparison process is finally refined to address the details of the nodes by performing micro comparison. At this low level, each node represents an individual statement. Such a statement is itself represented as a subtree (evaluation tree). The purpose of this comparison is to compare the structure of these evaluation trees. The comparison consists of two steps: (1) shape comparison to determine whether the two subtrees are similar; and (2) token comparison to determine whether the operands are similar. Under the premise that it is more dangerous to change semantics than syntax, shape comparison carries a more important value than token comparison. Indeed, it is much easier to rename variables than to change expressions. The shape comparison algorithm is a tree–isomorphism algorithm.

2.5 Data Dictionary Comparison

The data dictionaries are also compared in two ways. First, the number of each type of element is calculated separately. So, one may find that there are 3 ints and two floats in the first data dictionary and only 2 ints with 4 floats in the second. Secondly, as we look at each element in the first data dictionary, a matching element in type and id is sought out in the second one. Each type–id pair with a corresponding type–id pair in the other data dictionary is computed. Two percentages are computed: one for type similarity and the other for type–id pairs similarity.

3. AN ILLUSTRATIVE EXAMPLE

Consider the two structure charts shown in Figures 1 and 2. The first is assumed to be the original and the second is the copy. Chart 2 differs from chart 1 in that two nodes have been added.

3.1 Abstract Comparison

First the two trees are divided into major regions, represented by the region trees shown in Figures 3 and 4. Regions are delineated by blobs. Then ordered sequences are generated and compared. In the first iteration the following

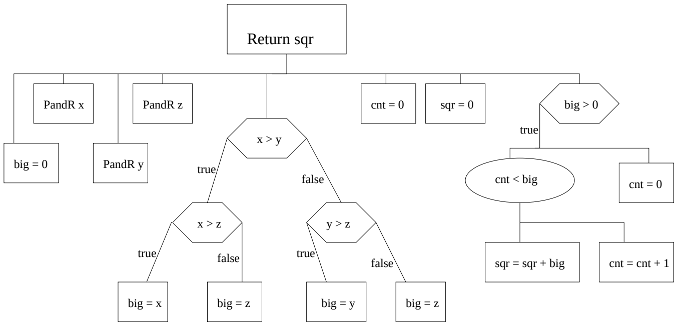

Figure 1: Structure Chart of the Original Program

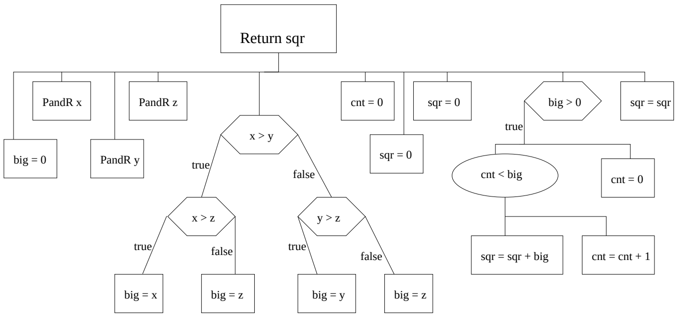

Figure 2: Structure Chart of the Copied Program

Chart 1 Data	Chart 2 Data
int x	int x
int y	int y
int z	int z
int sqr	int sqr
int cnt	int cnt
int big	int big

Table 1: Data Dictionaries for the two Structure Charts

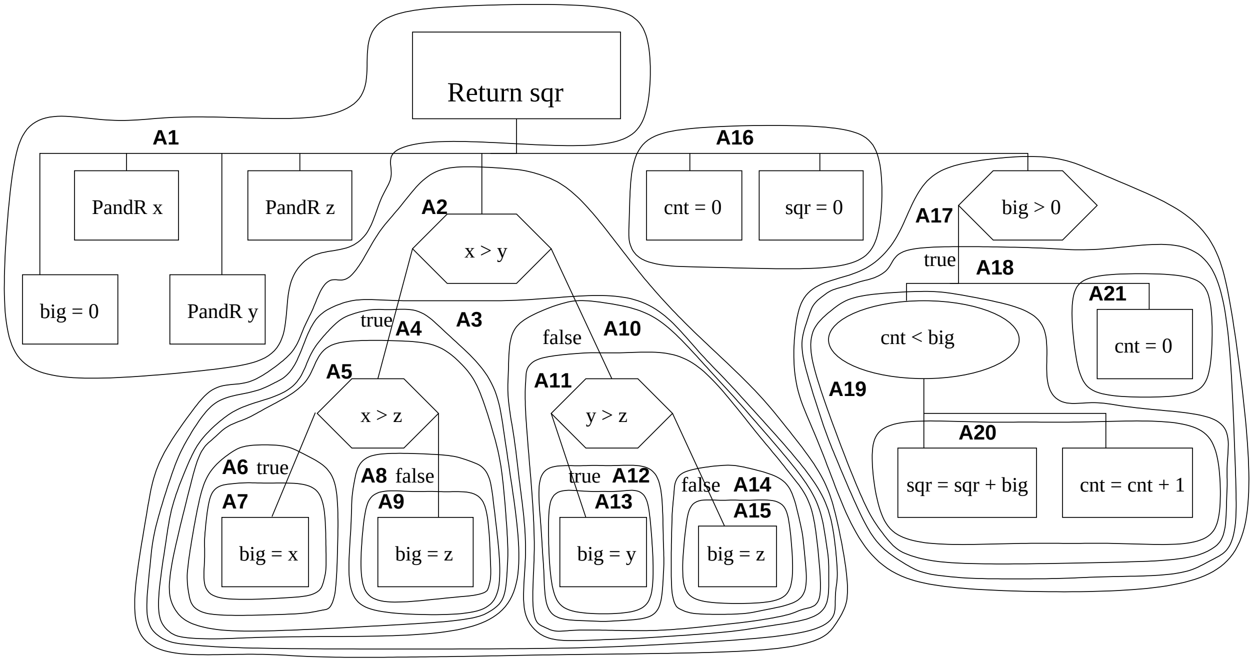

Figure 3: Region Tree for Chart 1

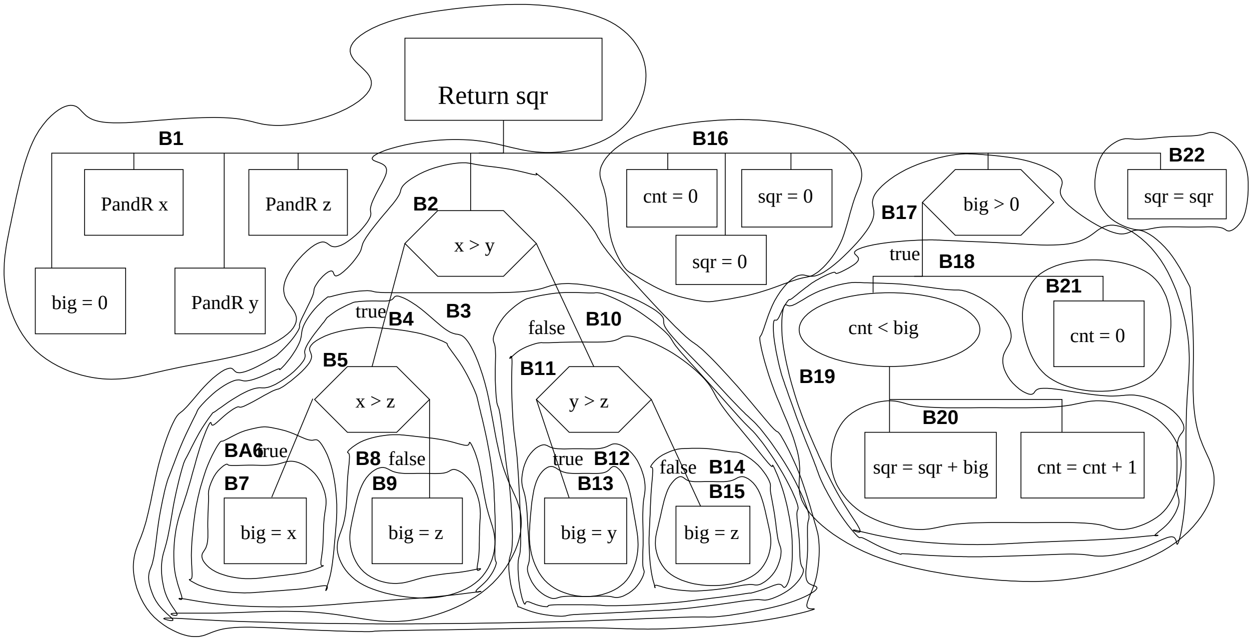

Figure 4: Region Tree for Chart 2

Region	General	Exact	Near	NoMatch1	NoMatch2
92.31	71.43	71.43	21.43	0.00	7.14

Table 2: Comparison Results

Change	YAP	MOSS	Brass
1	Immune	Immune	Immune
2	Immune	Immune	Notes-MC
3	Immune	Notes	Notes-MC
4	Immune	Immune	DD-Notes, Tree-Immune
5	Notes	Notes	Notes-MC
6	Notes	Notes	Notes
7	Notes	Notes*	Notes-MC
8	Notes	Notes	Notes-MC
9	Notes	Notes	Notes-MC
10	Notes	Notes	Notes
11	Notes	Notes	Notes-MC
12	Notes	Notes	Notes

Table 3: Performance of Similar Copy Detection Programs

sequences are generated: (1) for chart 1: $S_1 = <A1 : (sequence, 1), A2 : (selection, 1), A16 : (sequence, 1), A17 : (1, selection)>$; and for for chart 2: $S_2 = <B1 : (sequence, 1), B2 : (selection, 1), B16 : (sequence, 1), B17 : (selection, 1), B22 : (sequence, 1)>$. By comparing S_1 and S_2 it is determined that (A1, A2, A17, A17) matches (B1, B2, B16, B17) and that B22 has no match. Thus the longest string of matching regions between the two trees is set to be (B1, B2, B16, B17). The next iteration then concentrates on the sequences (A1, A2, A16, A17) and (B1, B2, B16, B17) by generating subsequences for each of corresponding regions pairs. In this case regions A1 and B1 do not have any subregions, so no further comparison is carried out on them. Comparison is moved to the next pair of regions A2 and B2. Their sequences are: A2: $S_{A2} =< A4A10 : (selection, 2) >$ and B2: $S_{B2} =< B4B10 : (selection, 2) >$. Again S_{A2} and S_{B2} are compared and the longest sequence (A4, A10) is identified. The process continues until all the subsequences are determined and compared.

The region comparison provides a good overview of how the two trees are structurally compatible. Being less extensive, this strategy is a quick overall check to determine whether a detailed comparison is warranted.

Now, node types comparison starts. It checks for exact matches in type and location of nodes in sequences generated previously. This kind of comparison mainly points out nodes that a user might want to take a closer look at. The comparison process traverses each region pair by visiting the nodes in a breadth firsth order. For example, by considering the region pair A1 and B1, the following lists are generated: for A1: (header, assign, I/O, I/O, I/O), and for B1: (header, assign, I/O, I/O, I/O). A comparison of these two lists results in complete similarity. After the comparison process is finished, the system generates a summary of the tree comparison – a set of percentages representing the matching nodes against the total number of nodes as shown in table 2.

4. ANALYSIS

Whale suggests twelve copy masking strategies when he compared his system against similar ones [3]. These include changing comments, formats, data types, identifiers, etc. We will use these here to show how our system (named Brass) performs in comparison to MOSS and YAP [5, 4]. The three systems were run on the same sample. Results are shown in Table 3. (in the table, Immune that the program will not notice this change in the copy; Notes means that the program will notice this change; and Notes-MC means that Brass does not notice the change until the micro–comparison part of the algorithm. Moss will not notice this change unless the statements have some difference other than just their identifiers.)

Brass is the only algorithm that tries to notice differences. The other two systems try to mask most changes that do not effect the functionality of the program. Because Brass uses structure charts, comments and formatting do not affect the matching. However, it does take into consideration every other type of change at some stage of its comparison. This provides much more informative information to the user. One can look at the structural comparison or just at the matching percentages for general data and the tree comparison will highlight specific areas for further study.

5. CONCLUSION

We presented a framework and an automated system whose the task is software plagiarism detection. Our framework constitutes a departure from traditional systems. It relies on designs and diivides up the detection process into levels of abstraction. The implemented system successfully notices similarities and differences between structure charts, and presents these in a useful form to the user. Compared to other plagiarism detection mechanisms, Brass provides more detailed information, allowing the user to draw accurate conclusions.

6. REFERENCES

[1] K. J. Ottenstein, "An Algorithmic Approach to the Detection and Prevention of Plagiarism," *CSD-TR 200*, AUGUST 1976.

[2] A.-M. L. John L. Donaldson and P. H. Sposato, "A Plagiarism Detection System," in *Twelfth SIGCSE Technical Symposium on Computer Science Education*, February 1981.

[3] G. Whale, "Identification of Program Similarity in Large Populations," *The Computer Journal*, vol. Vol 33, No. 2, pp. 140–146, 1990.

[4] M. J. Wise, "Detection of Similarities in Student Programs: Yap'ing may be Preferable to Plague'ing," in *ACM SIGCSE Bulletin Volume 24 , Issue 1*, March 1992.

[5] A. Aiken, "Moss.html," tech. rep., Computer Science Division of University of California, Berkeley, 2002.

[6] M. B. Nimmer and D. Nimmer, *Nimmer on Copyright*. Matthew Bender, 2002.

[7] M. Frohlich and M. Werner, *daVinci - Online Documentation V2.1*. University of Bremen, Germany.

[8] M. S. Hecht, *Flow Analysis of Computer Programs*. The Computer Science Library, North–Holland, Inc., 1977.