Automatic Generation of Plagiarism Detection Among Student Programs

Rachel Edita Roxas, Nathalie Rose Lim and Natasja Bautista

Abstract—A system for the automatic generation of plagiarism detectors that find similar programs in a set of student programs is presented. Existing plagiarism detectors are either applied to a programming language or a pre-defined set of programming languages. The general purpose one usually employs string matching to perform similarity measures that are based on plagiarism detection among documents in general, and not in programs in particular, thus, losing much of the structure and logic of programs in the process. On the other hand, plagiarism detectors for specific languages only cater to that particular set of languages.

This study provides a means for the user to specify the programming language of the student programs to be analyzed. Moreover, an automatic plagiarism detector system must be immune to the transformations that students perform on copied programs. These transformations are usually dependent on several factors namely: the type of programming problems and correspondingly, the complexity of the project to be implemented by the students, and also the programming language paradigm of the programs. Thus, the similarity measures employed by the system should be determined by these factors and can be specified by the professor. He/she has the option to specify how the similarities among the student programs will be captured. The system provides an interface for the specification of the particular programming language in which the student programs are implemented, and a knowledge-base of similarity measures that the user would like to include in the analysis of the student programs. Hence, the system provides flexibility in the programming language of the student programs to be analyzed and the similarity measures that the professor wishes to employ. Initial qualitative and quantitative evaluations illustrate a flexible, convenient and cost-effective tool for building plagiarism detectors for effective detection of programs in various imperative and procedural programming languages. The approach also addresses some of the changes that students perform on copied programs which JPlag fails to handle, thus, allowing for improved accuracy in terms of the reduction of false-positives, increasing the chance of catching plagiarized programs. These changes include modification of control structures, use of temporary variables and sub-expressions, in-lining and re-factoring of methods, and redundancy (variables or methods that were not used).

Comprehensive tests on other programming languages under various programming language paradigms such as object-oriented, logic and functional languages, considering the different changes that the students employ to copied programs (such as the tests done in JPlag) are also recommended for empirical evaluation.

Index Terms—Plagiarism detection, transformations, similarity measures.

I. INTRODUCTION

PROGRAM plagiarism has been an area of concern in the computer science community, most especially, in institutions teaching computer science courses. It has been emphasized that the process of learning computer programming is experienced through the actual exposure of the individual to various problem-solving scenarios. Since it is totally vital for a student to develop his own skills in programming, it is imperative that a measure be implemented in order to discourage students to plagiarize other people’s work. Thus, schemes for detecting plagiarism must be formulated.

From a set of student programs that are handed to the professor for a particular programming assignment, one way to detect plagiarism is to find pairs of programs that may be similar to each other to detect possible copied programs within the group of students. It is believed that the most frequently resorted sources of copied programs are the internet and peers/classmates.

As the number of computer science students increases, the detection of plagiarism among their peers/classmates becomes a difficult and complicated task. The use of the manual system of inspecting student programs becomes an unreliable and slow process; thus, automated systems are introduced. These systems are used to compare all pairs of programs, and to
Existing automated plagiarism detectors perform analysis of programs written in particular programming languages and implement specific plagiarism detection schemes. Because of the increasing demand to develop these various plagiarism detectors, a generalized plagiarism detector is presented. It offers flexibility in the specification of the programming language on which the programs being analyzed are written and the application of specific plagiarism detection scheme.

Plagiarism detectors have been implemented to analyze programs written in various programming languages. It is believed that the particular paradigm of the programming language of the student assignments to be analyzed would greatly affect the detection scheme to be applied.

Existing plagiarism detectors implement various plagiarism detection schemes. These systems aim to quantify or extract the characteristics of programs written to measure program content and are compared to detect program similarities. In previous studies, program similarities were measured using the quantitative or structural approaches.

Earlier systems used quantitative approaches, while later ones used structural approaches or a combination of both approaches. In 1987, a lexical detector [1] has been devised to count the occurrences of all the words in each of the programs. Similarity is then captured using this word occurrence scheme. It was based on the fact that program writers have their own unique way of representation and expression, thus similar use of words could mean plagiarized programs. Note that this system only detects copies which have common words used in programs. Because of the fact that students introduce changes to copied programs to disguise a copy, this system would be very limited to the lexical level.

Almost simultaneously, other quantitative approaches used software science parameters [2], [3] or variants thereof. The software science parameters are the number of unique operators, number of unique operands, total number of occurrences of operators, and total number of occurrences of operands. Program content is measured by a tuple of values of these parameters, and program similarity is done by comparing tuples of the suspected programs. These systems abstract from the actual identifiers to the operations performed on operands and the number of operands. The changes in identifiers and its data types do not affect this attribute counting, thus, are immune to these possible changes. Transformations such as introduction of unnecessary or redundant statements, though, may greatly affect the results.

Other parameters were used in other systems and adjustments made on counting the parameters to eliminate sensitivity to these introduced statements. Certain features [4], [5] are also attached to these parameters, and are set according to the expected characteristics of the group of programs to be analyzed.

But then, these systems attempted to capture the structure of the program using the quantitative method, losing much of the actual control structure details of the programs. Other detection schemes [6], [7], [1] introduced ways to avoid this by maintaining certain program constructs such as iteration, selection, compound, goto, procedure call, assignment, input and output. Some systems preserved the control flow of blocks of the programs by using graphical techniques, while others using strings. The graphs and strings were further processed and transformed into standard forms to minimize the effects of possible changes introduced by students to copied programs.

Substantial similarity between programs must be established regardless of the changes done by the students to disguise a copy. Detected plagiarized copies are obtained by “applying low-order semantics-preserving transformations, and by small modifications yielding versions with minor differences in functionality” [1].

The more recent web-based plagiarism detection systems are JPlag [8] and MOSS [9], which employed structural approaches. In a comprehensive evaluation of JPlag using various sets of Java programs [10], each transformation made by the students to disguise plagiarized copies was considered. It was shown that JPlag does not effectively detect plagiarized copies when the students introduce changes to the copied programs such as modification of control structures, use of temporary variables and sub-expressions, in-lining and refactoring of methods, redundancy (variables or methods that were not used).

A transformation-based plagiarism detection system such as JPlag does not effectively detect plagiarized copies when the students introduce changes to the copied programs such as modification of control structures, use of temporary variables and sub-expressions, in-lining and refactoring of methods, redundancy (variables or methods that were not used).

A transformation-based plagiarism detection system such as JPlag does not effectively detect plagiarized copies when the students introduce changes to the copied programs such as modification of control structures, use of temporary variables and sub-expressions, in-lining and refactoring of methods, redundancy (variables or methods that were not used).

II. THE PDG SYSTEM

The PDG incorporates a parser generator and a transformer generator to create the parser and the transformer, respectively (as shown in Fig. 1).

PDG uses the corresponding user-defined programming language definition of the programs being analyzed and abstract syntax tree (AST) construction library for the generation of the parser. This generated parser accepts the text programs and creates their corresponding abstract syntax trees (which are represented as a graph structure). PDG uses the user-defined transformation to be implemented and the AST transformation library for the generation of the transformer. This generated transformer accepts these graph structures and performs the transformations on them. String comparison follows for similarity measures of each pair of programs. Prospective plagiarized copies are manually checked to confirm plagiarism.
III. PARSER GENERATOR

The parser generator accepts the definition of the programming language for which the input programs are written and constructs the parser through the abstract syntax tree (AST) construction library. The parser generator is based on a syntax-directed translation scheme such that the parser is constructed by associating the operations in the construction library with the productions in the programming language specification.

The parser generated must be able to accept text programs and output their corresponding abstract syntax trees. The generated parser has two main parts: lexical analyzer and syntax analyzer. The lexical analyzer accepts text programs and groups legal strings into tokens. The syntax analyzer accepts a stream of tokens and groups legal sequences of tokens into syntactic structures.

A. Generalized Abstract Syntax Tree

The generalized plagiarism detector provides a library of functions to represent, access and manipulate common structures found in most imperative and procedural programming languages. These can be categorized as follows: composition statements, alternation statements, iteration statements and procedure definition and calls. These structures, identified and grouped according to functionalities, have a great influence on the transformations to be performed on the group of programs being analyzed. (These are further discussed in Section IV).

Composition statements, in general, do not have a specific control structure function. These include the assignment, input, output, label and goto statements, and procedure calls. Composition statements could be grouped into a compound statement.

Alternation statements express selection of statements or sequences of statements. It could be a selection of one or two choices or branches (such as an if-then/if-then-else statement), or a multi-branch statement (such as a nested if-then/if-then-else statement or a case statement).

Iteration statements could be a simple repetition, repetition while condition holds, repetition until condition holds, repetition while incrementing a counter, or indefinite repetition.

B. AST Construction Library

The operations were defined to perform the construction of the abstract syntax tree (AST) on individual nodes and on sequences of nodes (initialize a sequence and add a member to a sequence).

For instance, the AST definition for a case statement is as follows:

\[
\text{casestat} \rightarrow \text{condition: expression, branches: Seq of branch;}
\]

where expression and branch are defined elsewhere. The C function for the AST construction for casestat is as follows:

\[
\text{casestat make_casestat (theCondition, theBranches)} \\
\text{expression theCondition;} \\
\text{SEQbranch theBranches;} \\
\{ \\
\text{casestat C;} \\
\text{C = Ncasestat;} \\
\text{C->condition=theCondition;} \\
\text{C->branches = theBranches;} \\
\text{return C;} \\
\}
\]

C. Parser Generation

The parser generation is implemented using flex and bison, which are variants of Lex and YACC. Flex is used as a word recognizer and bison as a programming language definition recognizer. Flex recognizes input strings satisfying some patterns defined as regular expressions.

Bison is a compiler generator that accepts a specification defined in a grammar with actions associated with productions attached to it. This is a tool used with the scanner generator, Flex, to recognize a definition.

Error recovery mechanisms in bison attempt to prevent the drastic option of aborting parsing when a syntax error on the program being parsed has been found. It provides an error
token to represent anything so that the user could incorporate error tokens into the definition input. This handles an error such that when all the expected patterns are not satisfied, then the parser treats the error token as just another expected pattern and takes that token to represent anything, and thus, recovers from the error, and continues with the parsing of the next terminal symbol.

D. Parser Generation: Illustrated
Mini-languages control and scope [12] illustrate the implementation of imperative and procedural languages, respectively. Mini-language control provides composition, alternation and iteration statements, while mini-language scope provides sub-procedure definition and invocation, in addition to the statements provided by mini-language control.

For both mini-languages, the following is a partial bison input:

```
statement_list : statement {$$=add_stmt(init_stmtseq(), $1);} |
                statement_list statement {$$=add_stmt($1, $2);} |
                error { $$=init_stmtseq();} |
                statement_list error {$$=$1;}
```

where the procedures are defined and provided in the AST construction library.

For control, the expressions are discarded, while for scope, expressions are represented in the AST and all the declarations are kept for further processing in the transformer stage.

IV. TRANSFORMER GENERATOR

The second main part of PDG is the transformer generator. It accepts the user-defined transformation specifications to be applied to the AST of the student programs and constructs the transformer using the procedures defined and provided in the AST transformation library. This generated transformer accepts ASTs and outputs the corresponding transformed ASTs.

A. Generalized Transformer

Structural changes are employed by students on a plagiarized program to disguise the copy by using different programming language commands to perform the same tasks among others. Note that the introduction of these changes still maintains the control structure of the copied program. And because of these, plagiarism detection systems which are transformation-based, attempt to extract the control structure content of programs by examining the corresponding equivalences of programming language structures, and the way of transforming these equivalent structures into a canonical form. This is the identified means of coping up with the changes introduced by the students.

B. Equivalence of Program Structures

Equivalent program structures include:
1. changing to equivalent structural constructs (selection and iteration constructs)
2. reordering of blocks of statements in branches of selection statements
3. replacing expressions with equivalents
4. replacing procedure/function calls in in-line code. Recursion is a special case in that recursive relations could be reduced to iterative relations.
5. adding redundant statements. These include expansion of output statements and the addition of unnecessary initializations.
6. nested selection statements. A selection statement can be expanded to several selection statements.
7. factoring or de-factoring expressions or statements
8. equivalence of GOTO statements and other constructs in composition, selection and iteration.

C. Transformation Rules

Based on these program statement equivalences, the programs are converted into some standard form. Application of the transformation rules, the transformation phase, is divided into two parts: first stage and second stage. In the first stage, instances of the AST with the structure of the initial AST definition are accepted and are transformed into a construct list. This stage involves all the structural changes. The output construct list is characterized as a structure that contains the following constructs: assignment, select-option, loop, sequence, and sub-program call.

At the second stage, several passes are necessary to perform all the transformations since some transformations are needed to be performed on the construct list before others can be successfully carried out. The passes distinctly identify which transformations are to be performed at certain stages in the transformation, and ensure that they are performed as desired.

The following transformations were implemented in PDG: structural changes, expansion, compression, absorption, and in-line substitution.

1) Structural change: Selection and iteration statements are transformed to standard forms so that the change of equivalent control statements would not affect the analysis.

2) Expansion: Procedure/function in-line substitution is performed. This transformation requires a symbol table ADT which is also provided in the transformation library. The symbol table is used to keep track of procedure definition and scoping information.

3) Compression: Sequences of assignment and/or input and/or output statements are compressed to a standard form.

4) Absorption: Null compound statements (or null control blocks) are removed.

We recommend here a sequence of the performance of the transformations. Transforming the AST into the construct list using the structural equivalences is performed first. This particular transformation is embedded into the PDG as the first
transformation performed and the user transformer is made to undergo this stage. The succeeding transformations are as follows (in this particular order): procedure call expansion, compression, absorption, and reordering. Procedure call expansions should be performed before any compression or absorption since constructs of the body of the procedure definition will eventually be in sequence with other constructs in the position of the procedure call. This means that these constructs may miss being compressed or absorbed when procedure call expansion is performed later than compression or absorption.

Using the same argument, absorption must be performed before compression. This is because redundant compound statements will hinder the necessary compression of other contiguous redundant constructs (such as assignment, input and output constructs).

Similarly, the reordering transformation must be performed after redundant constructs have been removed, that is, after performing compression and absorption. If not, reordering will obtain erroneous weight counts or length measures due to this un-removed redundant constructs. In addition to this, procedure call expansions must be performed before the reordering transformations so that the computation of the weight counts or the length measures will be accurate.

Expression transformations are performed based on its occurrence in the program.

1) If-then-else Statement: Any subprogram call expansion in the condition is performed before the if-then-else statement, such that:

If condition then statement-sequence

is transformed to:

<condition>
Select
<statement-sequence>

The notation <X> denotes the transformation of X.

2) Case Statement: Similarly for case statements, any subprogram call expansion in the condition is performed before the case statement, such that:

Case condition
Branch option-1: statement-sequence-1
Branch option-2: statement-sequence-2
Branch otherwise: statement-sequence-3

is transformed to:

<condition>
Select
Option: <statement-sequence-1>
Option: <statement-sequence-2>

3) While Statement: Since the evaluation of the condition of a while statement is performed at every iteration of the while loop, then the subprogram call expansion is done inside the loop, such that:

While condition do
Statement-sequence
endloop

is transformed to:

loop
<condition>
<statement-sequence>
endloop

4) Repeat-Until Statement: The evaluation of the condition of the repeat-until statement is also performed at every iteration of the loop, but in contrast to the while statement, it is performed after the execution of the body of the loop, such that:

Repeat
Statement-sequence
Until condition

is transformed to:

loop
<statement-sequence>
<condition>
Endloop

5) For-Do Statement: The evaluation of a for statement condition is performed once before the iteration of the loop, such that:

For condition
Statement-sequence

is transformed to:

<condition>
Loop
<statement-sequence>
D. Symbol Table

A symbol table facility is used to keep track of user-defined identifiers and their associated information. Although the information in the symbol table could be directly obtained from the declarations in a program (if there are any), the organization of this information greatly depends on the scoping mechanism of the programming language being implemented.

For example, BASIC maintains a global symbol table and local symbol tables for functions which are visible only within the function.

In block-structured languages such as Pascal, C and Modula, identifiers are only visible within its scope, that is within the block in which the identifiers are declared. These languages maintain a symbol table structure that keeps track of the scopes of the identifiers. Since Pascal strictly requires that identifiers are declared before its use, the parser could check the current state of the symbol table to search for the identifier, thus, needing only a single pass.

Languages such as Algol do not require a declaration before use restriction, such that parsing would require two passes over a program: the first one dealing with the collection of the declaration information; and the second one for attaching reference for every occurrence of these identifiers.

PDG attempts to provide a representative scheme that could handle these mechanisms and enables the user to specify the choice and make corresponding additions to accommodate scoping implementation.

E. Transformer Generation: Illustrated

A sample partial transformation specification is shown as follows:

```plaintext
construct => assignment | select | ... ;
construct.select => options: Seq of option
{ Pass 3 : Transform Weigh [#];
  Pass 4: *Transform Reorder [#];
} Option => list : Seq of construct
{ Pass 1: *Transform Absorb [#];
  Pass 2: Transform Compress [#];
  Pass 3: Transform Weigh [#];
}
```

where the transformations Weigh, Absorb, Compress, and Reorder are provided in the AST Transformation Library.

F. Extensions to the Transformation Library

Definition of new transformations simply requires the incorporation of the transformation subprogram (and other supporting routines) into the library. This routine must be defined to conform to the generic AST specification and the routine name must be distinct from the names in the libraries.

V. RESULTS

The following changes that students employ on copied programs will not affect the analysis of the plagiarism detectors generated.

A. Addition of Comments

The system merely ignores such lexical changes and proceeds with the analysis of the statements in the programs.

B. Changing of Identifiers

Students usually change the identifiers in copied programs to disguise the plagiarism. For example, identifiers x and y are changed to a and b. Again, the system merely ignores such lexical changes on variables (excluding subprogram calls) and proceeds with the analysis of the statements in the programs.

C. Reordering of Independent Statements

To disguise a copied program, students may perform re-ordering of independent statements such as input and assignment statements. PDG deals with this by allowing the user to specify how to convert these independent statements into a standard ordering.

D. Addition of Useless Statements

An empty compound statement is an example of a useless statement. PDG removes such useless statements.

E. Addition of Redundant Statements

An example redundant statement can be output statements. An output statement is expanded to several output statements so that plagiarism would not be obvious to the professor who may check manually program plagiarism among student programs. Adding unnecessary initializations are also considered redundant statements.

F. Alternation Structural Change

Students break down a selection statement into several selection statements. For instance:

```plaintext
If (x=0 and y=0) then
  statement-1
else
  statement-2
end
```

is changed to:

```plaintext
if x=0 then
  if y=0 then
    statement-1
  else
    statement-2
else
  statement-2
end
```

G. Iteration Structural Change

Iteration statements could be transformed into a simple repetition, repetition while condition holds, repetition until condition holds, repetition while incrementing a counter, or indefinite repetition.
H. Re-ordering of Program Segments

This reordering may pertain to branches in a case statement or subprogram definition that is not used in the program. For instance, the following statement segments are equivalent:

Case
Branch condition-1: statement-sequence-1
Branch condition-2: statement-sequence-2
Branch otherwise: statement-sequence-3

and

Case
Branch condition-3: statement-sequence-3
Branch condition-2: statement-sequence-2
Branch otherwise: statement-sequence-1

Comparison of relative equivalence of particular branches are measured by computing the weights or complexities of each branch, and reordering branches in either ascending or descending order of complexities. The complexities may be computed in two ways:

1. Assign specific weights to each of the constructs in the AST. Gotcha [13] used this scheme with the following weights: assignment=2; case=4; loop=6; read=4; write=2.

2. Assume equal weights for each of the constructs in the AST which, in effect, computes for the length of the branch.

I. In-Line Substitution

In-line substitution of subprogram calls instead of defining the subprograms, or the other way around (defining subprograms rather than the actual code in the program) is another possible change that students introduce to copied programs. A special case must be examined in procedure and function calls, and that is, recursion. All recursive relations could be reduced to recurrence or iterative relations, and [14], [15] confirmed this claim. To illustrate, the recursive procedure for the factorial computation is as follows:

PROCEDURE factorial (n)
If n=1 then return 1
Else return (n*factorial(n-1));
End

with the corresponding equivalent iterative segment as follows:

Factorial := 1;
For i:=1 to n do
Factorial := factorial * i;

Though obtaining equivalent iterative constructs seem straightforward in this example, it may be very complex when the direct recursion (procedure calling itself) becomes complicated or when it is an indirect recursion (procedure P calls another procedure which in turn, or eventually, calls P).

The user may specify to perform in-line substitution depending on certain criteria such as frequency of calls of a certain subprogram or the length of the subprogram definition body (that is, if a definition exceeds a certain length then the user may not wish to perform in-line substitution), and recursive calls are simplified by substituting a recursive token in place of the recursive call. For more theoretical bases, a mechanical way of obtaining an iterative equivalent of recursive functions into flowcharts (often called removal of recursions) is presented by [15].

Note that because of this in-line substitution scheme, all subprogram calls are maintained in the AST. Identifiers in expressions are candidate subprogram calls. In-line substitution of subprogram calls depends on the type of statement for which it has been invoked.

J. Conversion of Goto Statements

Another possible change that students employ to copied programs includes the conversion of composition, selection and iteration into goto statements. In composition, for instance:

Statement-0
Goto label-1
Label-2: statement-2; Goto label-3;
Label-1: statement-1; Goto label-2;
Label-3: statement-3

is equivalent to:

Statement-0
Statement-1
Statement-2
Statement-3

In selection, for instance:

Statement-0
If a=0 then goto label-1;
Statement-1;
Goto label-2;
Label-1: statement-2
Label-2: statement-3

is equivalent to:

Statement-0
Statement-1
Statement-3

or

Statement-0
Statement-2
Statement-3
In iteration, for instance:

Statement-0
Label-1: If a=0 then goto label-2;
Statement-1;
Goto label-1;
Label-2: statement-2

is equivalent to:

Statement-1
Statement-2

or

Statement-0
Statement-1
Statement-2

Since evaluation of the detection and possible implementation of equivalent control constructs corresponding to these statements cannot be done in a straightforward manner and requires additional theoretical study, for this version of PDG, the transformation of GOTO and label statements into composition, selection and iteration was not implemented. Furthermore, non-local goto equivalences are more complicated to detect and to implement.

The system was tested on three student program sets: one set on C programs and the other two on Java programs. Comparisons were made on the resulting transformed programs using the common longest substring algorithm. The first set of files consisted of four simple C program files that computed for the average of two input numbers (see Table I). The original file was the basic program structure, and several copies of it were re-created using techniques that the plagiarism detector handles such as compression, substitution and re-ordering. The similarity matches were made on the codes 1 and 2 without transformations (or called immediate), and on codes 1 and 2 after each of the transformations, namely: compression, substitution and reordering. From the test results (see Table II), only the application of the transformation reordering yielded improvements to the similarity matches of program pairs. This may be because the original program is very short and simple, containing only 24 tokens. Thus, changes to the program are not too varied and therefore, the original program and its variants may not be too different from each other.

Percent improvements to similarity matches of programs where transformation reordering is applied range from 3.3 to 4.2%.

Also note that similarity matches that exceed 100% are due to inline substitution of code that counts the tokens of bodies of the subprogram more than once depending on the number of subprogram calls.

For the second set of test data, an original java file was duplicated four ways, with each file exhibiting an aspect that the PDG should detect: absorption, compression, substitution, and re-ordering (see Table III). The results are more significant in this data set in contrast to set 1 since the original program has more tokens (102 tokens) than the original program used in set 1 (24 tokens). When all transformations are applied, the system was able to detect similarities from 92 to 100% (see Table IV). Transformation reordering also consistently yielded higher similarity scores. Most of the transformations showed improvements (or no improvements) to the similarity matches of program pairs. As expected the plagiarized program with absorption when applied with the transformation absorption yielded a 100% similarity match, and program with reordering when applied with reordering yielded 93%. The only transformations that yielded negative improvements are transformation substitution on the plagiarized programs with absorption or substitution applied, and for transformation compression with reordering applied. Some of these irregularities may be attributed to the particular nature of the original program and not on the transformations, per se.

For the third set of programs, there were 19 Java programs and each one is compared with the other programs, thus, there were 171 program pairs compared. When all transformations are performed on the programs, 6 program pairs out of 171 (or 3.5%) yielded at least a 95% similarity. One program pair, in particular, had the highest percent improvement (76%) on the similarity score, from 19% similarity score before transformations (immediate) and 95% similarity score after all transformations have been applied. A manual evaluation of the program pair shows a possible plagiarized copy.

88% of the pairs yielded positive similarity improvements, while only 12% of the pairs yielded negative similarity improvements, all of which have an immediate similarity measure of at least 96%. On the average, programs that underwent transformation reordering yielded an 82% similarity match, while transformations substitution, compression and absorption yielded 28, 25 and 25% similarity measures.

In all data sets considered, reordering consistently yielded a higher similarity score when applied to test program pairs.

VI. SUMMARY, CONCLUSIONS AND RECOMMENDATIONS

The approaches taken are the generator paradigm, framework building blocks provided and the transformation-based plagiarism detector scheme.

The generator paradigm provided a convenient interface for the user such that it requires less coding by the user to be able to construct a plagiarism detector for a particular language and a specific programming assignment. This is also made possible by the libraries provided that contain the basic building blocks of constructing a plagiarism detector. Additional facilities could be integrated to the framework provided.

PDG also provides the theoretical basis for the detection of plagiarized copies by using the transformation-based approach.

1 Due to the large volume of data in results of this test set, the actual values are not presented in this paper. This data set was lent by JavaBugs (c/o Ms. Merlin Cruz-Suarez, De la Salle University-Manila, Philippines).
The libraries provided. Though this system could also be used languages, based on the transformation rules made available in new addition to the canonical form). Statements could be transformed to the canonical form (or to a analysis of its functionality could illustrate how these the analysis of options of the selection statement. Further study is recommended on the detection and implementation of the equivalences of Goto and labels and iteration or selection.

Exception handling, handled by some programming languages such as Ada and ML, provides a means of cleanly completing the execution of programs when a run-time error was identified by performing the associated exception code. This is a way of preventing the code from getting cluttered with error checks. The examination of this transformation needs to be evaluated considering these properties. This transformation may be found to be as complex as the recursion problem since errors could be done at various parts of the program.

Concurrent code segments (implemented in co-routines, Modula; par processes, Occam) could be treated similarly to the analysis of options of the selection statement. Further analysis of its functionality could illustrate how these statements could be transformed to the canonical form (or to a new addition to the canonical form).

The study has the following limitations. PDG is currently applicable to imperative and procedural programming languages, based on the transformation rules made available in the libraries provided. Though this system could also be used to analyze other programming languages which fall under various programming language paradigms, the orientation needs to be redefined. Other transformation rules should be devised based on these other PL paradigms, and incorporated into the libraries. Other plagiarism detection schemes could also be considered.

Also, the relationship of the complexity and length of the student programs to the effectiveness to detect plagiarized copies is to be established.

Lexical tests on student programs such as the checking of identical identifiers used, and identical long comment lines or identical spelling mistakes in comments, can be furthered studied and added into the system. Comprehensive tests on other programming languages under various programming language paradigms such as object-oriented, logic and functional languages, considering the different changes that the students employ to copied programs (such as the tests done in JPlag [9]) are also recommended for empirical evaluation.

VI. REFERENCES

Subthesis: Department of Computer Science, Australian National
University, 1986.


Journal of Computer and System Sciences, No. 5, 1971, pp 254-
285.

<p>| TABLE I |
| AC PROGRAM AND ITS &quot;PLAGIARIZED COPIES&quot; |</p>
<table>
<thead>
<tr>
<th>Program</th>
<th>Number of Tokens</th>
</tr>
</thead>
<tbody>
<tr>
<td>original</td>
<td>24</td>
</tr>
<tr>
<td>compression</td>
<td>36</td>
</tr>
<tr>
<td>substitution</td>
<td>31</td>
</tr>
<tr>
<td>re-ordering</td>
<td>24</td>
</tr>
</tbody>
</table>

<p>| TABLE II |
| SIMILARITIES OF PROGRAMS IN SET 1 |</p>
<table>
<thead>
<tr>
<th>Code 1</th>
<th>Code 2</th>
<th>Immed</th>
<th>Comp</th>
<th>Sub</th>
<th>Reord</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original</td>
<td>Comp</td>
<td>0.800</td>
<td>0.800</td>
<td>0.800</td>
<td>0.833</td>
</tr>
<tr>
<td>Original</td>
<td>Subst</td>
<td>0.873</td>
<td>0.873</td>
<td>0.873</td>
<td>0.909</td>
</tr>
<tr>
<td>Original</td>
<td>Reord</td>
<td>1.000</td>
<td>1.000</td>
<td>1.000</td>
<td>1.042</td>
</tr>
<tr>
<td>Comp</td>
<td>Subst</td>
<td>0.716</td>
<td>0.716</td>
<td>0.716</td>
<td>0.866</td>
</tr>
<tr>
<td>Comp</td>
<td>Reord</td>
<td>0.800</td>
<td>0.800</td>
<td>0.800</td>
<td>0.833</td>
</tr>
<tr>
<td>Subst</td>
<td>Reord</td>
<td>0.873</td>
<td>0.873</td>
<td>0.873</td>
<td>0.909</td>
</tr>
</tbody>
</table>

*Immed = Immediate comparison; Comp = Compression; Sub = Substitution; Reord = Reordering.

<p>| TABLE III |
| JAVA PROGRAM AND ITS &quot;PLAGIARIZED COPIES&quot; |</p>
<table>
<thead>
<tr>
<th>Program</th>
<th>Number of Tokens</th>
</tr>
</thead>
<tbody>
<tr>
<td>original</td>
<td>102</td>
</tr>
<tr>
<td>absorption</td>
<td>111</td>
</tr>
<tr>
<td>compression</td>
<td>102</td>
</tr>
<tr>
<td>substitution</td>
<td>102</td>
</tr>
<tr>
<td>re-ordering</td>
<td>102</td>
</tr>
</tbody>
</table>

<p>| TABLE IV |
| SIMILARITIES OF PROGRAMS IN SET 2 |</p>
<table>
<thead>
<tr>
<th>Code 1</th>
<th>Code 2</th>
<th>Immed</th>
<th>Abs</th>
<th>Comp</th>
<th>Sub</th>
<th>Reord</th>
<th>All</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original</td>
<td>Absorp</td>
<td>0.47</td>
<td>1.00</td>
<td>0.48</td>
<td>0.47</td>
<td>0.94</td>
<td>1.01</td>
</tr>
<tr>
<td>Original</td>
<td>Comp</td>
<td>0.47</td>
<td>0.47</td>
<td>0.49</td>
<td>0.47</td>
<td>0.91</td>
<td>0.92</td>
</tr>
<tr>
<td>Original</td>
<td>Subst</td>
<td>0.64</td>
<td>0.62</td>
<td>0.65</td>
<td>0.62</td>
<td>0.96</td>
<td>0.93</td>
</tr>
<tr>
<td>Original</td>
<td>Reord</td>
<td>0.38</td>
<td>0.38</td>
<td>0.37</td>
<td>0.38</td>
<td>0.93</td>
<td>0.94</td>
</tr>
</tbody>
</table>

*Immed = Immediate comparison; Abs = Absorption; Comp = Compression; Sub = Substitution; Reord = Reordering.