Abstract Syntax Tree Analysis for Plagiarism Detection

Erik A. Nilsson

Linköping 2012
Master’s Thesis

Abstract Syntax Tree Analysis for Plagiarism Detection

Erik A. Nilsson

Linköping 2012

Supervisor: Jonas Wallgren
IDA, Linköpings universitet

Examiner: Christoph Kessler
IDA, Linköpings universitet
Analys av abstrakta syntaxträd för detektion av plagiat.

Abstract Syntax Tree Analysis for Plagiarism Detection

Författare
Erik A. Nilsson

Nyckelord
abstract syntax tree, plagiarism, clone detection, fingerprint
Abstract

Today, universities rely heavily on systems for detecting plagiarism in students’ essays and reports. Code submissions however require specific tools. A number of approaches to finding plagiarisms in code have already been tried, including techniques based on comparing textual transformations of code, token strings, parse trees and graph representations. In this master’s thesis, a new system, cojac, is presented which combines textual, tree and graph techniques to detect a broad spectrum of plagiarism attempts. The system finds plagiarisms in C, C++ and Ada source files. This thesis discusses the method used for obtaining parse trees from the source code and the abstract syntax tree analysis. For comparison of syntax trees, we generate sets of fingerprints, digest forms of trees, which makes the comparison algorithm more scalable. To evaluate the method, a set of benchmark files have been constructed containing plagiarism scenarios which was analyzed both by our system and Moss, another available system for plagiarism detection in code. The results show that our abstract syntax tree analysis can effectively detect plagiarisms such as changing the format of the code and renaming of identifiers and is at least as effective as Moss for detecting plagiarisms of these kinds.

Sammanfattning

Acknowledgments

I would like to thank my supervisor Jonas Wallgren for his indispensable help, both during the project and with the report itself. I extend thanks to my coworkers Eric Elfving for his help with $\LaTeX$ and $\text{Bib}$TeX, Klas Arvidsson for his help regarding C++ programming and Torbjörn Jonsson for supplying the resources we needed to get started. I would like to thank Niklas Holma, the codeveloper of COJAC and who has been writing his master’s thesis in parallel with mine, for his commitment to the project. Finally, I thank my loving mother, for her help and support during this project.
# Contents

1 Introduction 1
   1.1 Project Background 1
   1.2 Common Terms 2
   1.3 Plagiarism Definitions 2
   1.4 Overview of the Thesis 3

2 Related Work 5
   2.1 Clone Detectors 5
   2.2 Clone/Plagiarism Detection Approaches 6
      2.2.1 Text Based Techniques 6
      2.2.2 Metrics Based Techniques 6
      2.2.3 Token Based Techniques 6
      2.2.4 Syntax Tree Techniques 6
      2.2.5 Graph Based Techniques 7
   2.3 Comparisons of Clone/Plagiarism Detectors 7
   2.4 Using gcc for parsing C/C++ 8
   2.5 Moss 8
      2.5.1 How Moss Works 8
      2.5.2 Output from Moss 9

3 Goals and Requirements 11
   3.1 Basic Functionality 11
   3.2 Modularity 12
   3.3 Performance 13
   3.4 Documentation 13

4 Overview of the System 15
   4.1 COJAC 15
      4.1.1 Source Code Units 15
   4.2 Overview of Detect 16
      4.2.1 Configuration Parameters 17
      4.2.2 Important Data Objects 18
      4.2.3 Main Module 18
      4.2.4 Textual Diff Module 19
      4.2.5 Front Ends 19
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.2.6</td>
<td>AST Analysis</td>
<td>20</td>
</tr>
<tr>
<td>4.2.7</td>
<td>PDG Factories and PDG analysis</td>
<td>20</td>
</tr>
<tr>
<td>5</td>
<td>Front Ends</td>
<td>21</td>
</tr>
<tr>
<td>5.1</td>
<td>Tree Representation</td>
<td>21</td>
</tr>
<tr>
<td>5.2</td>
<td>C/C++ Front End</td>
<td>22</td>
</tr>
<tr>
<td>5.2.1</td>
<td>The GCC Plugin</td>
<td>23</td>
</tr>
<tr>
<td>5.2.2</td>
<td>Format of the Dump File</td>
<td>23</td>
</tr>
<tr>
<td>5.2.3</td>
<td>Dump File Parser</td>
<td>25</td>
</tr>
<tr>
<td>5.2.4</td>
<td>Tree Reconstruction</td>
<td>25</td>
</tr>
<tr>
<td>5.2.5</td>
<td>Tree Simplification</td>
<td>26</td>
</tr>
<tr>
<td>5.2.6</td>
<td>Setting Node Control Types</td>
<td>27</td>
</tr>
<tr>
<td>5.2.7</td>
<td>Tree Examples</td>
<td>28</td>
</tr>
<tr>
<td>5.2.8</td>
<td>C vs. C++ Front End</td>
<td>33</td>
</tr>
<tr>
<td>5.2.9</td>
<td>Generalizing Control Structures</td>
<td>33</td>
</tr>
<tr>
<td>5.2.10</td>
<td>Finding Uses and Definitions in the Tree</td>
<td>37</td>
</tr>
<tr>
<td>5.3</td>
<td>Ada 95 Front End</td>
<td>39</td>
</tr>
<tr>
<td>5.3.1</td>
<td>The Symbol Table</td>
<td>39</td>
</tr>
<tr>
<td>5.3.2</td>
<td>The yacc grammar</td>
<td>39</td>
</tr>
<tr>
<td>5.3.3</td>
<td>Tree Examples</td>
<td>40</td>
</tr>
<tr>
<td>5.3.4</td>
<td>Generalizing Control Structures</td>
<td>42</td>
</tr>
<tr>
<td>5.3.5</td>
<td>Replacing Case Statements</td>
<td>43</td>
</tr>
<tr>
<td>5.3.6</td>
<td>Lifting Up Nested Statement Lists</td>
<td>44</td>
</tr>
<tr>
<td>5.3.7</td>
<td>Finding Uses and Definitions in the Tree</td>
<td>44</td>
</tr>
<tr>
<td>6</td>
<td>AST Analysis Module</td>
<td>45</td>
</tr>
<tr>
<td>6.1</td>
<td>Module Overview</td>
<td>45</td>
</tr>
<tr>
<td>6.2</td>
<td>Fingerprinting</td>
<td>45</td>
</tr>
<tr>
<td>6.2.1</td>
<td>Hashing Subtrees</td>
<td>45</td>
</tr>
<tr>
<td>6.3</td>
<td>Matching Algorithm</td>
<td>47</td>
</tr>
<tr>
<td>6.4</td>
<td>Output of the Module</td>
<td>49</td>
</tr>
<tr>
<td>6.4.1</td>
<td>Pairs of Greatest Similarity</td>
<td>49</td>
</tr>
<tr>
<td>6.4.2</td>
<td>Similarity Matrix</td>
<td>50</td>
</tr>
<tr>
<td>6.4.3</td>
<td>List of Subtree Matches</td>
<td>51</td>
</tr>
<tr>
<td>7</td>
<td>Method of Evaluation</td>
<td>53</td>
</tr>
<tr>
<td>7.1</td>
<td>Plagiarism Scenarios</td>
<td>53</td>
</tr>
<tr>
<td>7.1.1</td>
<td>Format Alteration Scenarios</td>
<td>54</td>
</tr>
<tr>
<td>7.1.2</td>
<td>Identifier Renaming Scenarios</td>
<td>54</td>
</tr>
<tr>
<td>7.1.3</td>
<td>Declaration Reordering Scenarios</td>
<td>55</td>
</tr>
<tr>
<td>7.1.4</td>
<td>Statement Reordering Scenarios</td>
<td>55</td>
</tr>
<tr>
<td>7.1.5</td>
<td>Code Insertion Scenarios</td>
<td>55</td>
</tr>
<tr>
<td>7.1.6</td>
<td>Control Replacement Scenarios</td>
<td>55</td>
</tr>
<tr>
<td>7.1.7</td>
<td>Scenarios of Other Methods</td>
<td>56</td>
</tr>
<tr>
<td>7.2</td>
<td>Running Moss on the Scenarios</td>
<td>56</td>
</tr>
<tr>
<td>7.3</td>
<td>Running the AST Diff on the Scenarios</td>
<td>56</td>
</tr>
</tbody>
</table>
# Contents

8 Analysis Results ........................................... 59
  8.1 Moss compared to the AST Diff .......................... 59
  8.2 Comparison of the Front Ends ........................... 61
  8.3 Execution Time ........................................ 65

9 Discussion ................................................. 67
  9.1 Format Alteration Scenarios ............................. 67
  9.2 Identifier Renaming Scenarios ........................... 67
  9.3 Declaration Reordering Scenarios ......................... 68
  9.4 Statement Reordering Scenarios ........................ 68
  9.5 Code Insertion Scenarios ................................ 68
  9.6 Control Replacement Scenarios .......................... 68
  9.7 Other Scenarios ........................................ 68
  9.8 Overall ................................................. 68
  9.9 Execution Time ........................................ 69

10 Conclusions ................................................ 71

11 Future Work ............................................... 73
  11.1 Improvements to DETECT ................................. 73
    11.1.1 Extended Language Support ......................... 73
    11.1.2 Serialization of Parsed Code Units ................ 73
    11.1.3 Handling Commutative Nodes ........................ 74
    11.1.4 Other Forms of Output ............................. 74
    11.1.5 Matching Algorithm for Detecting Near Matches ... 74
  11.2 Improvements to COJAC ................................. 75
    11.2.1 Graphical User Interface for Setting up Batch Runs ... 75
    11.2.2 Web Interface for Code Submissions ................ 75

A Implementation Details .................................... 81
  A.1 Tree Representation Interface .......................... 81
  A.2 GCC Plugin .............................................. 86

B List of Requirements ....................................... 88

C Default Configuration File ................................ 91

D Configuration File for Benchmarking ....................... 92
Chapter 1

Introduction

1.1 Project Background

Today, systems for plagiarism control are heavily used by teachers to detect students who copy material from the web or from other students. URKUND [27] is a system used by Swedish universities for this purpose. Such a system can find plagiarisms in papers and essays of various formats and is therefore useful in many courses in many different areas. However, such a system is not suitable for code submissions. This is because a student can make a quick overhaul which greatly changes the appearance of the code but makes no difference in its functionality.

Nevertheless, finding plagiarisms in code is of value for teachers teaching basic and intermediate programming courses. Students attending such courses are equally tempted to copy material from the web or from their fellow students. There are a number of systems for plagiarism control of source code (both free and commercial), though not many systems check submitted material against the web. TickNik [26] is a Swedish commercial system set up as a web interface, which supports a wide range of programming languages. Moss [2] is a free system developed at Stanford University.

Currently no system for plagiarism control in code submissions is in use at the Department of Computer and Information Science at Linköping University. At an internal conference UPP, the group in charge of the basic programming courses at the department, discussed the need for a system for plagiarism control. The group proposed a master’s thesis project to develop such a system which could rival other available systems. Apart from being free, a locally developed system would also be customizable to the group’s specific needs.

In this master’s thesis project a system for plagiarism detection, COJAC, has been constructed. The system was developed by Erik Nilsson and Niklas Holma for the Division of Software and Systems at the Department of Computer and Information Science. Torbjörn Jonsson and Klas Arvidsson, two employees at the department, belonging to the UPP group, are the customers of the system. At the time of this project both Erik Nilsson and Niklas Holma were employed by the department, working in this group as part time teachers.
The focus of the project has been COJAC’s major component DETECT, the tool for comparing two units of code. Future work will make use of this tool, improve its usability and availability for teachers teaching programming or project courses. This will be further discussed in chapter 9.

1.2 Common Terms

In this report specific terms are used, which are explained below.

**Plagiarism** A piece of code (ranging from specific passages within a file to content spanning several files) which has been copied from an original and may have been altered to obscure the act of plagiarizing. We assume that such plagiarisms strive not to change the semantics of the code. We make this assumption based on the fact that copying code and deliberately changing the semantics would defeat the purpose of plagiarizing.

**Abstract Syntax Tree** Abstract syntax tree is abbreviated as AST. In our program this is in fact an acyclic directed graph (a DAG).

**Program Dependence Graph** Program dependence graph is abbreviated as PDG. However, in our system we deal with approximate PDGs, or APDGs. An APDG consists of an approximate data dependence set (ADDS) and an approximate control dependence set (ACDS). APDGs are further discussed in Niklas Holma’s report [12].

1.3 Plagiarism Definitions

When discussing plagiarisms, we categorized them according to the ways they alter the code. We have adopted the definitions used by Liu et al. [17] and have added some of our own:

**Format Alteration (FA)** A unit of code which has been plagiarized by format alteration has undergone changes in whitespaces and/or commenting. For example; changing the style of indentation of a file, random insertion of spaces and empty lines, breaking lines where the syntax allows it and removing/editing/adding comments.

**Identifier Renaming (IR)** A systematic replacement of variable, function and parameter names in a code unit is considered to be identifier renaming. This technique together with format alteration are perhaps the most readily used since modern integrated development environments have easy-to-use refactoring features and automatic code formatting.
1.4 Overview of the Thesis

**Declaration Reordering (DR)** Switching the order of function declarations or definitions, the order of function parameters or the order of locally declared variables is considered to be cases of Declaration Reordering. (This is included in the Liu et al. [17] definition of statement reordering). This also includes moving function definitions to separate include files.

**Statement Reordering (SR)** Statements which are not dependent on each other can be reordered while still maintaining the program correctness. For example two assignments of different variables which are both set to a constant value.

**Control Replacement (CR)** In programming languages which have multiple control structures for repetition and selection, one such structure can usually be replaced by an equivalent, such as replacing a for statement by an equivalent while, or a switch statement by an equivalent if. Another common technique is to negate a predicate in an if statement and then exchanging the then and else clauses.

**Code Insertion (CI)** Code which does nothing can easily be inserted without changing the semantics of the code. For example, one may define a variable which is never read, call a function which does nothing, add “dead” code after a function’s return etc. It may not only be simple statements which are inserted in this way. A plagiarizer may insert loops with empty bodies, or loops which will never run. Another technique is to surround code blocks with control structures which always execute exactly once. E.g. a while statement with a predicate which always evaluates to true but where the last statement is a break.

**Other Methods (OM)** The categories above cover a broad area of plagiarism techniques. However, there are other methods which do not really fall into any of these categories. For instance changing the type of a numeric variable from integer to real where it does not affect the semantics. Another example is inlining function bodies or the opposite; moving code into subroutines. There are many instances where the programming language offers equivalent constructs, such as the += operator in C++, which is easily converted into a regular incrementing assignment. These techniques have been put into this category since they do not really fit in anywhere else.

1.4 Overview of the Thesis

The remainder of this thesis is organized as follows: In chapter 2, related work in this field is presented. Chapter 3 presents the goals of this master’s thesis and the requirements for the project as a whole. An overview of the developed system is presented in chapter 4 and its front end modules are discussed in chapter 5. Chapter 6 explains the AST analysis and how the results of the analysis is presented to the user of the system. In chapter 7, a comparison is made between the developed system and another tool for plagiarism detection, the results of which
are presented in chapter 8 and further discussed in chapter 9. The conclusions of this master’s thesis are presented in chapter 10. In chapter 11, future work is discussed and improvements to the developed system are suggested.
Chapter 2

Related Work

2.1 Clone Detectors

Very close to the subject of detecting plagiarisms in code is the subject of detecting clones, two pieces of code which are similar. Depending on the definition of “similar”, clones are categorized as different types. According to Koschke et al. [16], the definition of these types are as follows: Type 1 is an exact copy with only modifications in whitespace and comments. Type 2 is a syntactically identical copy, where only identifiers have been changed. Type 3 is a copy with other modifications, such as removed or added statements. These definitions are close to the plagiarism categories explained in section 1.3. A type 1 clone can be viewed as a copy which has been modified by format alteration. A type 2 clone is a copy which has been modified with identifier renaming (and perhaps FA as well). A type 3 clone can be seen as a copy where other techniques have been applied, such as code insertion, statement/declaration reordering and control replacement.

As opposed to plagiarisms, clones do not only arise in malicious copying. According to Baxter et al. [4], clones occur for various reasons; code reuse by copying pre-existing idioms, using coding styles, failure to use/identify abstract data types, performance enhancements and by accident. It is of interest to corporations to detect such clones to decrease the maintenance cost of their systems. The problem of detecting clones is similar to the problem of finding plagiarisms in code, and thus the methods for doing so are interesting to us. However clone detectors detect clones for a different reason than plagiarism detectors. When finding clones it is important to locate exactly where these pieces of code are in order to remove those clones. Some clone detection systems (such as CloneDr [4]) even come with support for automatic removal of clones. For plagiarism detection it is more important to gauge how similar two pieces of code are, since such tools will run on many code units and the purpose of the tool is to flag which units may be plagiarisms. However, it may also be the case that the users of plagiarism detection tools want to see what parts were found to be similar, and indeed many tools offer this functionality.
2.2 Clone/Plagiarism Detection Approaches

Many approaches have been tried for detection of clones and plagiarisms in code. The methods vary from simple text based approaches to complex graph analyzing methods.

2.2.1 Text Based Techniques

A simple, straightforward approach to detect plagiarisms would be to simply run students’ code through a textual differ. Such a program could report how many lines of codes were found equal in the two files. Such a technique would be fast, reliable and support all kinds of programming languages but would be very easy to fool. Instead, one can do some processing of the text before comparison. One such program is Dup [3], which can systematically substitute variable names and constants. Another was proposed by Ducasse et al. [9] which processes each line by removing comments and whitespaces before comparison. The tool Moss works by transforming files into a sequence of hashes [23]. Moss is further discussed in section 2.5.

2.2.2 Metrics Based Techniques

Some tools extract various metrics from the code and compare such metrics vectors rather than the code itself. Some extract metrics from the code, others parse the code and gather metrics from the resulting parse tree or other intermediate representations.

A metrics based tool was presented by Mayrand et al. [18] which gathers metrics of functions from the abstract syntax tree and various graph representations. The metrics are grouped in four categories: function name, layout (such as average variable name length and number of non-blank lines), expressions (such as number of calls to other functions and number of declaration statements) and control flow (e.g. number of nodes and number of control statements).

2.2.3 Token Based Techniques

Tools such as CCFinder [14] first conduct a lexical analysis to obtain a sequence of tokens, or lexemes, which can then be used for comparison. Such a lexical analysis uses lexical rules of a language and is therefore language dependent.

2.2.4 Syntax Tree Techniques

Clone detection has also been conducted by the use of abstract syntax trees. To do this, the source code is first parsed and the resulting parse tree is used for further comparison. The idea of finding clones in trees is simple enough; compare each subtree to all other subtrees and record which trees are equal. However comparing trees of size $N$ is an $O(N)$ operation, so comparing all subtrees of a tree to all subtrees of another tree is $O(N^3)$. The problem of finding clones in
ASTs thus becomes a question of how to make an algorithm scalable. Baxter et al. [4] proposed an algorithm for finding clones in ASTs where subtrees are categorized by hash values, and then only comparing trees in the same category. A deliberately bad hash function, which ignored identifier nodes, was used in order to find not only exact, but also near exact subtrees.

The search for approximate clones comes with the downside of being time consuming. The fewer categories of subtrees there are, the longer it will take to find all subtree clones. Chilowicz et al. [7] instead proposed to use a good hash function, one which reflects the structural properties of the tree. Their method is based on representing each subtree with a digest form, called a fingerprint. The fingerprint is a tuple containing the subtree’s weight, hash value and pointers to the original tree. When finding subtree matches, their algorithm considers only fingerprints of equal weight and compares the hash values of the fingerprints. Trees with different hash values cannot be equal (in the isomorphic sense, not identical), however trees with equal hash values could be unequal if a hash collision has occurred. This fact cannot be ignored. In those cases the algorithm traverses the tree and compares a fixed number of childrens’ fingerprints as well, which reduces the probability of false positives being detected.

2.3 Comparisons of Clone/Plagiarism Detectors

Recently there have also been some thorough comparisons between these techniques. Roy and Cordy [22] conducted a comparison in which they evaluated clone/plagiarism detection tools by testing them with specific scenarios. The 22 editing scenarios covered changes in code which corresponds to format alteration, identifier renaming, declaration reordering, statement reordering, control replacement, and code insertion. From their evaluation they concluded that different techniques were effective for detecting different clone scenarios. For instance, token based techniques were found effective against scenarios of format alteration and identifier renaming (variable renaming), while the tool GPLAG was confirmed to be strong against scenarios of statement reordering and control replacement.

A large scale comparison was conducted by Bellon et al. [5], in which the Bellon Benchmark was used to evaluate six clone detectors. The benchmark consists of running the code of four systems (of different sizes) written in C and four systems written in Java through the clone detectors and recording the result. The result is then compared to a set of oracled (manually detected) clones which makes up about 2% of the original code. They concluded that text and token based tools had
Related Work

a higher recall, a metric describing the ratio of detected clones (in one program) divided by the number of oracled clones (for that program). This suggests that, in terms of plagiarization, such tools would have a higher rate of false positives. Some AST based tools had higher precision which would suggest that such techniques are less prone to report false positives. However, these tools did suffer from poor execution time. They also concluded that the evaluated PDG based tool was only sensible to use for detection of type 3 clones.

2.4 Using gcc for parsing C/C++

Hsiao [13] constructed an AST based tool for plagiarism detection using gcc’s C compiler as a parser. By using a specific flag, gcc can be prompted to dump the parse trees of a compilation unit to file. The ASTs were then reconstructed from those files. A graph grammar learner, which takes a graph as input and yields grammar rules as output, was applied to the trees. Similarity between trees was then calculated by comparing the resulting grammars.

2.5 Moss

In chapter 7, a comparison of our system and the tool Moss is described. Moss (Measure of Software Similarity) is an automatic system for plagiarism detection in code [2]. It was developed by Alex Aiken in 1994 and is currently used for detecting plagiarisms in programming classes at Stanford University. The system is set up as an internet service. Users can submit files by running a script from the command line. The script communicates with the Moss server which generates HTML web pages with the results. The URL is sent back as a response to the user.

2.5.1 How Moss Works

Schleimer et al. [23] presents the method used by Moss. Moss can be described as a textual tool which performs some transformation of the text of a file into a set of fingerprints (as defined below) of the file. These sets of fingerprints are then compared for similarity. The transformation occurs in four steps:

1. Irrelevant features, such as whitespaces, are removed.
2. A sequence of $k$-grams, where a $k$-gram is a contiguous substring of fixed length $k$, is derived from the text.
3. Each $k$-gram in the sequence is hashed.
4. A subset of those hashes is selected by winnowing. Winnowing works by splitting the sequence into a sequence of windows of hashes. The windows are overlapping subsequences of size $w$ where $w$ is calculated from the constants $t$ and $k$. $t$ is a threshold for how long substring matches must be to be
2.5 Moss

guaranteed to be detected. $k$ is a threshold for how long substring matches can be and still be considered to be noise (small changes). From each window, the minimum hash is selected (if that hash was not already selected in another window). If there are several minimum hashes, the rightmost one is selected. The resulting subset of hashes is the fingerprints of the file. The minimum hash is selected since it is probable that the minimum in one window will be the minimum in adjacent windows. In this way the number of selected hashes is reduced.

2.5.2 Output from Moss

For two input files A and B, Moss gives two similarity ratings (as percentages) and the number of lines which are considered to be matched in A and B. Moss also displays what parts of A and B were considered a match, and by use of a “thermometer” informs how large that match was in relation to the entire file’s size. If more than two files are submitted to Moss, this output is presented for each pair of files.
Chapter 3

Goals and Requirements

The requirements of the system were formed together with the customer and stated in the project plan of this master’s thesis project. The full, detailed list of requirements from the project plan is presented in appendix B. Among the requirements described below, those requirements which are marked with an asterisk are not relevant for this thesis but for the project as a whole.

3.1 Basic Functionality

Running on a POSIX System. It was known from the beginning of the project that COJAC should run on a POSIX system.

Multiple Language Support. Initially it was required that the system could handle Ada, C and C++. It was conceivable that support for Java and MatLab would be useful as well, but this was left as a lower prioritized requirement.

*Comparing Several Submissions at Once. It was required of the system to be able to compare many units of code (not just two), e.g. the submissions from all students pertaining to a specific lab assignment. The system was not required to check submissions against material on the web.

*Comparison on Different Levels of Abstraction. To become a versatile tool, it was required that the system could compare code on different levels of abstraction. A comparison on a low level (such as a simple textual diff) is very effective for finding exact, or near exact plagiarisms but may however be fooled by simple plagiarism techniques like format alteration and identifier renaming. A comparison on a higher level of abstraction may be able to deal with more invasive plagiarism techniques. On the other hand, it may lack effectiveness and be more prone to false positives.
Another benefit of applying several methods is the ability to discern what plagiarism techniques may have been applied, since different methods of detection are effective against different plagiarism techniques, as seen in chapter 2.

Outputs of the System. Basically the outputs of the system were requested to be of three different types:

- *An output based on a textual comparison of the code units. In particular, it is interesting to know how many lines of code in one file matched code lines in the other file. It was also required that the output specified which lines matched and that the user could specify if whitespaces should be considered or not.

- A similarity ratio of the code units based on an analysis which is robust against format alteration, identifier renaming, and declaration reordering. This was motivated by the belief that these techniques are the most commonly used ones. Also, it matched what was expected to be achieved by an AST based approach. It was preferable that matching in this way could report source code locations for found matches.

- *A similarity ratio of the code units based on an analysis which is robust against statement reordering, code insertion and control replacement as well. This was motivated by the customer’s request of the system being able to detect a wider range of plagiarism techniques. Also, it matched what was expected to be achieved by a PDG based approach.

Keeping Records of Past Runs. Two requirements regarded COJAC’s ability to store information about previous runs in a database. The idea was to store each submitter’s history of submissions and output from DETECT. Moreover, COJAC was to flag submitters whose history contained frequent cases of high output metrics. Since the focus of this project was on DETECT itself, these requirements were of lower priority.

3.2 Modularity

The Department of Computer and Information Science at Linköping University offers an array of programming courses. There is no “one” programming language taught. Courses are given in not only C, C++, Ada but also in Java, Python, MatLab, Lisp, Ruby, R, Prolog, C#, etc. Needless to say, the more languages a system such as COJAC can support, the better. Since the methods of analysis which were intended to be used are language dependent, the only sensible choice is to design the system to be as modular as possible. In this way, the system can be made to support new languages by installing a new module. In this case, a front end.

Modularity also improves the maintainability of the system. New standards of the supported programming languages are imminent. Thus, the modules handling those languages must be easy to update or replace.
Since COJAC is a tool with multiple approaches to plagiarism detection it is also conceivable that new analysis modules may be integrated in the future. Or that already existing analysis modules should be replaced. These requirements prompt us to make the system as modular as possible.

3.3 Performance

A goal in this project is to develop a system which can compete with other available systems for plagiarism detection such as Moss and TickNik in terms of what forms of plagiarisms it can detect. There are no specific requirements on execution time.

3.4 Documentation

The development of this system will not conclude in conjunction with the completion of this master’s thesis project. The system will be used, owned and maintained by the department at which it was developed. As in all software engineering projects, the need for proper documentation is unquestionable. Specifically, documentation on how to use COJAC and how to add or remove front ends in DETECT is required.
Chapter 4

Overview of the System

4.1 COJAC

COJAC is a system for comparing multiple source code units against one another and finding pairs of code units which may contain plagiarism. The system uses a file system database to manage source code units. Internally COJAC relies on the tool DETECT for comparing two source code units at a time. The focus of this master’s thesis is on DETECT, therefore the description of the surrounding architecture is only rudimentary. A schematic of the system is shown in figure 4.1.

![Figure 4.1: Overview of COJAC.](image)

4.1.1 Source Code Units

A source code unit (also code unit or input unit) is a main file and a set of include files. The rules for how files may include one another depends on the programming language.

For C/C++ An include file must either be included from the main file or from another include file in such a way that if inclusion is a relation from the includer to the included, the transitive closure of that relation must not be reflexive.
For Ada  An included file may also include an implementation file. This is the case when a specification file (.ads-file) has an accompanying implementation file (.adb-file). In this case the implementation file is part of the same source code unit as the main file.

From the user’s point of view, a drawback when comparing projects of C/C++ code is that one project may be made up of multiple code units. If that is the case, these code units must be analyzed separately by DETECT. When comparing Ada project this can be avoided since another definition of the source code unit is used. The reason this was implemented for Ada and not for C/C++ was that implementation files in Ada always bear the same base name as the specification file.

The entire source code unit is considered to be of a certain programming language, therefore the files it contains must be of the same programming language. The system does not support comparison of code units of different languages. The system also requires the code to actually be parsable. It does not handle code units which produce errors upon parsing.

4.2 Overview of Detect

Figure 4.2 shows the modules of DETECT.
4.2 Overview of Detect

4.2.1 Configuration Parameters

Configuration parameters can be passed to detect by use of a configuration file. The configuration file is a text file with key-value pairs on each line. There is a default configuration file with default values for every key that is used by detect. The default configuration file is shown in appendix C. The values can be of boolean, integer or string type. Table 4.1 shows examples of configuration parameters controlling the main parts of detect. (There are many others.)
### Overview of the System

<table>
<thead>
<tr>
<th>Key</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ENABLE_DEBUG_MODE</td>
<td>boolean</td>
<td>Prints debugging messages (used during development).</td>
</tr>
<tr>
<td>ENABLE_CONFIG_DUMP</td>
<td>boolean</td>
<td>Prints the configuration settings from within DETECT.</td>
</tr>
<tr>
<td>ENABLE_TEXT_DIFF</td>
<td>boolean</td>
<td>Enables the textual diff analysis module.</td>
</tr>
<tr>
<td>ENABLE_AST_DIFF</td>
<td>boolean</td>
<td>Enables the AST analysis module.</td>
</tr>
<tr>
<td>ENABLE_PDG_DIFF</td>
<td>boolean</td>
<td>Enables the PDG analysis module.</td>
</tr>
<tr>
<td>AST_DUMP_DOT_GRAPHS</td>
<td>boolean</td>
<td>Outputs a PNG file depicting the tree generated by the frontend.</td>
</tr>
<tr>
<td>AST_DOT_GRAPH_DETAIL</td>
<td>boolean</td>
<td>Enables more detail in the PNG of the generated tree.</td>
</tr>
<tr>
<td>AST_DUMP GENERICIZED</td>
<td>boolean</td>
<td>Outputs a PNG file depicting the tree after generalizations have been done.</td>
</tr>
</tbody>
</table>

Table 4.1: Configuration parameters for the main parts of DETECT.

#### 4.2.2 Important Data Objects

DETECT has an object oriented C++ implementation. This section does not discuss implementation details, however, in table 4.2 important data structures which are used internally are presented to facilitate the discussion of the system.

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Config</td>
<td>A class for storing configuration parameters.</td>
</tr>
<tr>
<td>CodeUnit</td>
<td>A class for keeping track of source code units.</td>
</tr>
<tr>
<td>AST_interface</td>
<td>An interface for the front end’s tree representation.</td>
</tr>
<tr>
<td>ASTForest</td>
<td>A set of AST_interface references.</td>
</tr>
<tr>
<td>ASTMetrics</td>
<td>A class for recording the result of the AST analysis.</td>
</tr>
</tbody>
</table>

Table 4.2: Main data structures used internally in DETECT.

#### 4.2.3 Main Module

The main module instantiates a Config from the input config file. Those keys which are not specified in the input config file are taken from the default config file. Should the key ENABLE_TEXT_DIFF be set to TRUE, two CodeUnit objects will be instantiated. The code units will then be preprocessed (see section 4.2.4) before sent to the Textual Diff module.

Two other CodeUnit objects are then instantiated for parsing. The CodeUnits which were used for the Textual Diff are not used since the preprocessing modifies the paths within those objects. Depending on the code units’ language an appropriate front end object is instantiated and the front end’s parse method is
4.2 Overview of Detect

called for each code unit. If the AST diff is enabled (ENABLE_AST_DIFF is set to TRUE), the two ASTForest objects which result are then passed to the AST analysis module.

If the PDG analysis module is enabled (ENABLE_PDG_DIFF is set to TRUE) an appropriate PDGfactory will be instantiated and create a set of APDGs from the AST forests. These APDGs are then given as input to the PDG analysis module.

4.2.4 Textual Diff Module

Before running the textual diff module on the code units, these must be preprocessed. When preprocessing a code unit, the main file and all include files are pasted together into one (new) file.

Then non-empty lines are removed and each line is indexed, stripped from whitespaces and treated as a symbol. The result is a string of such symbols. The module then proceeds to find all nonoverlapping common substrings, and their mappings, of the two strings. The figure 4.3 illustrates the mappings of the lines in two source files. The textual diff module is further explained in Holma’s report [12].

![Figure 4.3: The line mappings of two files, matched by the textual diff module.](image)

4.2.5 Front Ends

The language specific front ends take a CodeUnit object and parses it into a ASTForest object. This module is more deeply explored in chapter 5.
4.2.6 AST Analysis
The AST analysis module takes two \texttt{ASTForests} and a \texttt{Config} object as input and searches for matches between forests. It produces an \texttt{ASTMetrics} object which can print an output to an output stream. This module is further described in chapter 6.

4.2.7 PDG Factories and PDG analysis
It was our ambition to make the creation of APDGs a language independent process. This however proved difficult since the APDG creation algorithm used is based on the syntax tree which can look very different depending on which language’s grammar was used. The traditional approach to constructing a program’s PDG is by using the postdominator tree of a program’s control flow graph. Our method is based on a technique presented by Harrold et al. [11], which constructs an approximate control dependence subgraph (ACDS), and an approximate data dependence subgraph (ADDS) from the AST of a procedure. The APDGs are then analyzed by the PDG analysis module which finds subgraph isomorphisms in the graphs. The construction and analysis of such APDGs is the topic of Holma’s report [12].
Chapter 5

Front Ends

The front ends of detect are responsible for parsing the code units and producing abstract syntax trees for each procedure in each code unit. There is an interface specifying the functionality of front end classes. Classes implementing the interface must implement a function parse which takes a CodeUnit object and returns an ASTForest object. Three front ends have been implemented; C, C++ and Ada. If detect is to be extended to support another programming language, a front end for that language which implements the front end interface must be developed.

The C/C++ front end uses gcc as a parser. For Ada a custom parser was created. The decision to create a custom parser was driven by two things: The advantage of being able to control the contents of the parse tree and not needing to take the detour of reconstructing the tree from another tool’s tree representation. For the C/C++ front end, much time was spent just understanding gcc’s tree. However, the fact that gcc has a working C/C++ parser was the dominating factor in the choice of using gcc. Creating a C/C++ parser is a difficult and time consuming task.

5.1 Tree Representation

To clarify, in this context, when we say “tree” we really mean a directed acyclic graph (a DAG). Our abstract syntax trees are DAGs since one node can be a child of two parents, e.g a funcion call node which uses a variable $A$ and an assignment node which assigns that same variable $A$, will both have a node representing $A$ as a child. Such common subtrees are not limited to leaves in the tree. For instance, two nodes may reference the same function declaration node, which in turn may have children (e.g. nodes representing formal parameters).

Throughout detect, the various modules which use ASTs, reference such trees through the class AST_interface. The interface represents one node in the tree. Each node has a type, a type number (used for hashing, see 6.2.1), a Control Type (further explained in section 5.2.6), and a use and definition list (see section 5.2.10). Moreover, each node has a set of labeled children and a set of attributes (key-value pairs). The code for the interface is shown in section A.1.
It is convenient for modules which are not themselves dependent on the source language to handle a generic tree. For instance, the AST analysis module handles objects of the \texttt{ASTForest} class, which is a set of \texttt{ASTInterface} objects. The front ends themselves however make use of specialized representations. There is a class for C/C++ tree nodes (usually just called C++ tree) and a class for Ada tree nodes. These classes implement the interface.

5.2 C/C++ Front End

The C and C++ front ends share much code. They are described here as one unit. At the end of this section their differences are discussed.

As mentioned in chapter 2, it is possible to get a textual representation of \texttt{gcc}'s raw C/C++ AST. This can be achieved by compiling the source code with the \texttt{-fdump-tree-original-raw} flag. We will call this representation the \texttt{dump}. However, since the format of the dump cannot be manipulated, the method used by Hsiao [13] was not sufficient for our needs. For example, the syntax trees of procedures in system header files were included in the dump, which made the dump files larger than necessary. Instead a custom plugin for \texttt{gcc} was developed (described in section 5.2.1) which could produce a dump more according to our needs (see section 5.2.2). The stages of the \texttt{parse} method are shown in figure 5.1.

The front end compiles the code (but does not actually link or assemble it) with the plugin, which produces the dump file. The file is then parsed by a part of the front end called the \texttt{dump parser} which reconstructs the nodes of the ASTs (outlined in section 5.2.3). These sets of sets of nodes are then passed on to the \texttt{tree maker} which reconstructs the trees (see section 5.2.4). Once the trees have been made, the front end will run a \texttt{simplification} procedure to remove unwanted parts of the tree (section 5.2.5), \texttt{Control Types} will be set (see section 5.2.6) and finally the tree will be checked for cycles.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{parse_function_c++_front_end.png}
\caption{Overview of the \texttt{parse} function in the C/C++ Front End.}
\end{figure}

The front end also contains routines for further processing of the trees which are called before the APDG construction. One such routine performs tree transformations (detailed in section 5.2.9), another traverses the tree and finds where
variables are used and defined (see section 5.2.10).

5.2.1 The GCC Plugin

The basis of the plugin was created from the information on the web tutorial by Boris Kolpackov [15]. As of version 4.5.0 GCC supported custom made plugins written as shared objects which can be hooked into specific stages in the compilation. The tutorial clarifies the GCC documentation [25] on plugins and describes how to construct a plugin which prints all the declarations which appear in the global namespace after the parsing of the source code is done. This was similar to our needs and the example code could be used as a foundation for our plugin.

According to GCC’s internal documentation, GCC has a set of predetermined events in the compilation process where a plugin can be called. A plugin must register a callback function at one of these events. The event plugin_pre_genericize permits the plugin to access the low level AST of each function in the C/C++ front end (of GCC). This event is triggered after parsing but before the major optimization passes, which makes it a good place to extract the AST. The callback function is called with a pointer to the AST as argument. Should any errors arise during parsing the plugin will not run, only print an error message on the standard error stream. To keep the plugin as simple as possible, the plugin was designed to produce a dump which resembles the one given by the -fdump-tree-original-raw flag. The plugin does this by calling the same function as the flag invokes but in an alternate header file. The header file was copied from the GCC source code and then altered. In this way the dump format could easily be manipulated. The code for the plugin can be seen in section A.2.

5.2.2 Format of the Dump File

The format of the dump has been modified in the following ways:

**Printing the file and function Name**  Originally the dump only printed the function name. In our case it is also important to keep track of which file held the actual source. Fortunately, there exists a GCC macro, DECL_SOURCE_LOCATION, which takes a tree pointer (the argument of the callback function) and returns an expanded_location object. Such an object keeps track of the source of the currently parsed function and has the function and file name. An end-of-function marker is also printed to make the parsing of the dump easier.

**Skipping system header functions**  Dumps created with the flag became huge when system header files were included in the source code. A simple “hello world” program could generate a 5K-line dump file. This is a problem, since our system should deal only with code which has actually been submitted (not code in system header files). Fortunately the expanded_location object not only contains the source code location of the function, but also a boolean representing whether this function is a system header function or not. This boolean is used to disregard those trees which come from functions in system header files.
Printing the type number for each node  Each node in the AST has a type, e.g. `function_decl`, `statement_list` etc. For the convenience of the analysis module the corresponding integer value of those types are printed for each node. The value is extracted from the nodes by GCC’s `TREE_CODE` macro.

Properly printing parameters  For function declarations, the original dump contains only the first parameter. For our purpose, we want to see all the parameters of a function. Unfortunately the parameters of a function are linked as a chain of parameters rooted at the function declaration node. It is more convenient to have each parameter of a function as a child node of that function declaration node. This is however fixed later in the front end. The plugin dumps the parameters as a chain. The parameters are accessed by calling GCC’s macro `TREE_CHAIN` on nodes of `parm_decl` type.

Printing line numbers whenever possible  Since being able to trace matched trees back to source code is important in this project, it is favorable for as many nodes as possible to have source code locations. It is not strange that some nodes lack this information since the compiler generally inserts nodes during the parsing which do not originate from the syntax itself but rather from the grammar rules. In the original dump, source code locations were only printed for declarations. For C++-trees there exists a `line` attribute for some nodes. The plugin enhances this greatly and prints line numbers for nodes which have a source code location. The macros `EXPR_HAS_LOCATION` and `EXPR_LINENO` are used to achieve this. This is one of the plugin’s more important functionalities.

Printing local declarations  For local declarations, the original dump only printed these as `decl_expr` nodes. The plugin makes use of the macro `DECL_EXPR_DECL` to find the entity which was declared, e.g. a `label_decl`.

Removing string constants  For nodes of type `string_cst`, the original dump printed the actual string. Obviously this could ruin the format of the dump file so the plugin does not print the actual string, only its length.

Resulting Format

The dump produced by the plugin is saved to a file. The file consists of an arbitrary number of function entries on the following format:

```
FUNCTION: <name>
FILE: <path-to-file>
<list of nodes>
END_OF_FUNCTION
```

Notice that the function name does not have to be unique. For C, `<name>` is simply the name of the function. For C++, `<name>` has the following format:

```
<return-type> <function name>(<param-type>, <param-type>, ...)
```
The list of nodes consists of an arbitrary number of node entries (each one on its own line) on the following format:

@<uid> <type num> <type> <list of attributes>

The fields are separated by at least one space character. The uid is a unique (within the current function) ID for the node. The ID is used when referencing the node. The type of a node describes that node’s meaning in the tree, the type num is the type’s corresponding value. The list of attributes is an associative list of data and edges to other nodes. Each key-value pair is separated by whitespace and the list can be broken over several lines. Figure 5.2 is an excerpt from a dump file, as an example of the format.

FUNCTION: void subprog2(int, int)
FILE: Testsuites/C++/Analys/plag1.cc
@1 60 cleanup_point_expr type: @2 line: 2 op 0: @3
@2 20 void_type name: @4 algn: 8
@3 258 expr_stmt type: @2 line: 2 line: 2 expr: @5
@4 36 type_decl name: @6 type: @2 srcp: <built-in>:0
note: artificial
@5 112 convert_expr type: @2 op 0: @7
@6 1 identifier_node strg: void lngt: 4
@7 52 modify_expr type: @8 op 0: @9 op 1: @10

Figure 5.2: Excerpt from a dump file.

5.2.3 Dump File Parser
The dump parser simply scans the dump file produced by the gcc + plugin compilation and constructs a sequence of dumped_function objects. Such objects have file and function name as data members as well as a set of dumped_node objects. Each node has a unique ID (unique within the node’s tree), a type, a type value and a set of attributes.

5.2.4 Tree Reconstruction
The function make_tree in the C/C++ front end takes a dumped_function object and reconstructs the AST it represents. First a node table is filled with C++ tree node objects, one for each dumped_node in the dumped_function object. Then for each dumped_node N, N’s list of attributes is processed. References to other dumped nodes are edges in the tree, and those edges are added among the C++ tree nodes in the node table. Other attributes are simply added to the attribute set of the C++ node which corresponds to N.

Some edges between nodes are omitted to ensure that no cycles arise during the reconstruction of the tree. Nodes which represent different types, such as integer
type, can themselves have a child type. Because of this, the node representing void
type has itself as a child, thus creating a cycle. These edges are therefore removed.
The type of a node is instead added as an attribute of that node. Some nodes
have a scope attribute which references another node. One example is parameters
having the function they belong to as scope. Since that function is a parent to
the parameter, this creates a cycle, thus the scope attribute has been removed in
those nodes.

Nodes representing function declarations (the function of the current tree and
others) appear internally within the tree. One example is when the formal pa-
rameter of the current function is referenced by another node (as in figure 5.3 and
5.4). For the construction of the ACDS in the APDG construction it is convenient
if the tree’s function declaration appears near the root of the tree. This is because
the function declaration contains declarations of the function’s formal parameters,
and those parameters should be declared before used in other nodes. If those pa-
rameters are not declared, the ACDS will not contain important data dependency
edges related to those parameters. A node is therefore placed as the new root of
the tree. A function declaration node, declaring the function for the current tree,
appearing among the dumped_nodes objects is added as child to this new root.
When the tree has been reconstructed entirely the old root is inserted as a child
of this new root (an example of the result can be seen in figure 5.5).

```c
void fun(int param) {
    param = 3;
}
```

Figure 5.3: A func-
tion where a param-
eter is assigned.

Figure 5.4: Conceptual
original parse tree of the
code in 5.3.

Figure 5.5: Conceptual
parse tree after tree re-
construction.

### 5.2.5 Tree Simplification

gcc’s C/C++ AST is a rich and complicated structure, in fact, it contains more
information than what is needed for the AST analysis. This is because the gcc tree
contains information which is needed for code generation. The function simplify
in the C/C++ front end recursively traverses the tree and makes the following
modifications to the tree in order to reduce its size.

**Removing size nodes** The byte sizes of variables, parameters etc. appear as
child nodes in the tree. For us, this information is not important and the node
could simply be removed. However, anticipating that the information may be needed in future development, the size has been stored as an attribute of the parent node instead.

Removing identifier nodes  Just like the size nodes, identifier strings appear as children of variable declarations, parameter declarations etc. This information has also been stored in the parent. It would have been possible to design the plugin so that these nodes (size and identifier) are never dumped. However, it was simpler to implement a removing of these nodes than to further modify the original GCC code.

Removing Mangled Names  Some nodes have children which contain mangled names. These children were deemed unimportant and were simply removed.

Removing convert_expr and nop_expr nodes  These nodes are placeholders for code generation for conversions which require, and do not require actual code, respectively [25, p. 195]. Since these nodes adhere to code generation, they are removed. The (single) child of a convert_expr node is added as a child to the parent of the convert_expr node. The same is done for nop_expr nodes. The node is not removed if it happens to be the root of the tree.

Removing cleanup_point_expr nodes  These nodes represent full expressions and are placeholders for calls to destructors of temporaries created within the expressions [25, p. 199]. Such nodes have been removed in the same manner as convert_expr and nop_expr nodes.

Removing bind_expr nodes  These nodes represent block structures and the variables they declare [25, p. 204]. For our purpose these nodes were not necessary to maintain. They were removed in the same manner as convert_expr, nop_expr and cleanup_point_expr nodes.

Removing cleanup_stmt nodes  These nodes represent actions which should be taken when exiting a certain block, such as calls to destructors [25, p. 220]. Such implicit calls are not important to maintain in the tree and are removed as well.

5.2.6 Setting Node Control Types

The construction of the ACDS depends on the Control Type of nodes. The control type indicates how the node affects the control flow of the source program. Before returning the ASTForest, the C/C++ front end recursively traverses the tree and sets the correct control type for each node. The control types in table 5.1 exist and are common for all front ends:
<table>
<thead>
<tr>
<th>Control Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NONE</td>
<td>The default control type. Nodes which do not affect control flow, or are of no other interest to the APDG construction are marked with this control type.</td>
</tr>
<tr>
<td>PROGRAM_BEGIN</td>
<td>Control type for the root of the AST.</td>
</tr>
<tr>
<td>ASSIGNMENT</td>
<td>Control type for assignment nodes.</td>
</tr>
<tr>
<td>DECLARATION</td>
<td>Control type for variable and function declaration nodes.</td>
</tr>
<tr>
<td>FCALL</td>
<td>Control type for function call nodes.</td>
</tr>
<tr>
<td>STATEMENT_LIST</td>
<td>Control type for nodes which contain other statements. If such a node is at the root of the tree, this control type overrides PROGRAM_BEGIN.</td>
</tr>
<tr>
<td>IF_ELSE_BEGIN</td>
<td>Control type for nodes representing if-statements.</td>
</tr>
<tr>
<td>CONTINUE</td>
<td>Control type for continue statement nodes.</td>
</tr>
<tr>
<td>BREAK</td>
<td>Control type for break statement nodes.</td>
</tr>
<tr>
<td>GOTO</td>
<td>Control type for goto statement nodes.</td>
</tr>
<tr>
<td>LABEL</td>
<td>Control type for label nodes.</td>
</tr>
<tr>
<td>DETECT_LOOP</td>
<td>Control type for detect loop nodes (see section 5.2.9).</td>
</tr>
<tr>
<td>PROGRAM_EXIT</td>
<td>Control type for return statement nodes.</td>
</tr>
</tbody>
</table>

Table 5.1: The control types and their descriptions.

5.2.7 Tree Examples

In this section some examples of ASTs are presented. For a detailed description of the gcc’s tree representation GENERIC, see gcc’s internal documentation [25]. Figure 5.6 shows the tree structure for a function with a for statement. Figure 5.7 shows the tree structure for a variable declaration and an if statement. Figure 5.8 shows a while statement and a call to a function bar. Figure 5.9 shows the tree structure of goto statements and labels, assignments, expression and return statements. Notice that the expression 3*15 has been folded into one constant by the gcc front end. For an example of a switch statement, see figure 5.11. The images were generated by detect. (The image generator colors nodes which have actual source code locations blue. The color of the nodes is of no importance in this context.)
void foo(int i) {
    for (int i = 0; i < 10; ++i) {
    }
}

Figure 5.6: Syntax tree for a function with a for statement.
```c
int a;
if (a == 5) {
} else if(a > 9) {
} else {
}
```

Figure 5.7: Syntax tree for an if statement and variable declaration.
while (true) {
    bar(15);
}

Figure 5.8: Syntax tree for a `while` statement with a function call to a function `bar`. The formal parameter `j` of `bar` (not visible in the code excerpt) is visible as a child to the node representing the function.
```c
int foo(int i) {
    goto L;
    i = 3 * 15 + i;
L: return i;
}
```

Figure 5.9: Syntax tree for a function with a `goto` statement, a label, an assignment and a `return` statement.
5.2 C/C++ Front End

5.2.8 C vs. C++ Front End

How is the C front end different from the C++ front end? Initially the source code is compiled using two scripts, which are similar. The C++ script compiles with `g++` whereas the C script compiles with the command `gcc`. Both scripts use the same plugin.

In addition to what has been described above, the C front end makes some small adjustments after the simplification step.

- **gcc**’s C tree represents `if` statements with the node `cond_expr`. In these nodes the control type is set to `IF_ELSE_BEGIN` and the labels of the children are adjusted to be similar to the `if_statement` node which appears in the C++ tree.

- **gcc**’s C tree represents `switch` statements with the `switch_expr` node instead of `switch_stmt` which occurs in the C++ tree. The structure of the statement is however the same. The type string `switch_expr` is therefore changed to `switch_stmt`.

- **gcc**’s C tree represents loop constructs with `if` statements and `goto` statements. Unifying of loop constructs (described in section 5.2.9) for C trees has not been implemented.

5.2.9 Generalizing Control Structures

For the APDG construction, similar control statements, such control statements which can readily be replaced by one another, should be represented in a unified way. This is to be able to detect plagiarisms where control replacement has been applied. The following subsections describe what operations are performed to generalize C++ trees.

Unifying Loop Constructs

These operations turn subtrees for `for`, `while` and `do-while` statements into a unified structure which we call a `detect loop` (see figure 5.10). A node with the control type `DETECT_LOOP` can have four children: `precondition`, `postcondition`, `body` and `after`. Precondition is a loop exit condition which is checked at the beginning of the loop (used by `for` and `while`). Postcondition is a loop exit condition which is checked at the end of the loop (used by `do-while`). Body is a statement list with the body of the loop. After is a statement list with statements which are executed at the end of each iteration, such as incrementing iterators. Fortunately the tree structures for loops (in C++) are already organized in this way. Therefore (in contrast to the Ada front end) this operation simply renames the children of `for`, `while` and `do-while` statements to match the respective parts in the detect loop. However, a detect loop does not need to have all four children, for instance only a detect loop generated from a `do-while` has the child Postcondition.
Replacing Switch Statements

This operation replaces switch statements by equivalent structures made up of if statements and gotos. This is done by traversing the switch’s statement list and constructing a set of if statements with gotos to the case labels found within. Whenever a break is encountered (which is not in a nested loop or another switch), that break is replaced by a goto to the end of the switch. An example of a switch statement and the corresponding parse tree is given in figure 5.11. Figure 5.12 shows the structure after the transformation (nested statement lists have also been transformed as explained below) and the code which corresponds to the transformed tree.
int var;
switch (var) {
case 1:
  1;
  break;
default:
  3;
}

Figure 5.11: Parse tree for the a `switch` statement. Note that the statements 1 and 3 are represented by the single node `@46integer_cst`. Because of this, the line information has been lost for the first statement.
int var;
if (var == 1) {
    goto L1;
} else {
    goto L2;
}

L1: 1;
goto L3;
L2: 3;
L3: 

Figure 5.12: Syntax tree after transformation of the tree in fig 5.11. Also, the use and definition sets have been calculated for the tree (see section 5.2.10). The code corresponding to the tree is shown below the tree.
Lifting Up Nested Statement Lists

Statement list nodes can be nested, as in figure 5.9. This operation lifts up the statements of a nested statement list. Figure 5.13 shows what the top part of the tree in figure 5.9 looks like after this transformation is done. This is not actually performed to generalize control structures, but to facilitate the construction of the ACDS.

![Figure 5.13: Part of the tree from figure 5.9 after nested statement lists have been transformed. In this tree, the use and definition sets have also been calculated.]

5.2.10 Finding Uses and Definitions in the Tree

For the construction of the ADDS each node must specify which variables/parameters it uses and defines. By use of a variable/parameter we mean an operation which will read the memory location of that variable/parameter. By definition of a variable we mean an operation which gives the memory location of that variable/parameter a value or an operation which declares that variable/parameter. Since the APDG constructor considers nodes which have a node control type other than none, the use and def sets are calculated for these nodes only.

There is a set of member functions in the C++ tree which traverse the tree and calculate these sets. To keep track of variables, the ID of the node is used (together with the name of the variable if it has one). Since these IDs are unique within each function, there can be no mixup of variables. However it is worth mentioning that the use and definition sets that are produced are only an approximation of the sets defined above. This is mainly because variables and parameters can be aliased through pointers. For instance, consider the scenario in figure 5.14.

By our implementation, a is defined on the first line, b is defined and a used on the second line, and b is defined on the third line, when in fact it is a which is being defined. Use-def relations for aliased variables can be correctly computed by
int a;
int* b = &a;
*b = 5;

Figure 5.14: On the third line, a is being assigned by dereferencing a pointer to a. DETECT is unable to find such variable definitions.

using Static Single Assignment form of the code [8], but this is beyond the scope of this master’s thesis and has not been implemented in DETECT.

Here is how the use and def sets are calculated for different node control types:

**Declarations**  For nodes with control type DECLARATION, if the entity being declared is a variable, that variable is added to the declaration’s definition set. If there is an initialization expression in that declaration the variables/parameters appearing in that expression are added to the use set.

If the declaration is a function declaration the parameter declarations are added to the definition set of the function declaration.

**Assignments**  For nodes with control type ASSIGNMENT (this includes pre- and post-, in- and decrementations) variables/parameters appearing in the right hand side are added to the use set. If the left hand side is an indexed array, the name of the array is added to the definition set. If the left hand side is an indirect reference (such as a dereferenced pointer) or a component reference (such as a data member in a class) the variable/parameter appearing in the left hand side is added to the definition set. If there are multiple variables/parameters appearing in the indirection expression, they are all added to the definition set. In reality, only one memory location is given a value but since we cannot determine which variable/parameter appearing in the left hand side is bound to this memory location we set them all as being defined. As discussed earlier, this leads to the definition set only being an approximation. If the left hand side is a regular variable or parameter, that variable/parameter is added to the definition set.

**Function Calls**  For nodes with control type FUNCALL, variables/parameters which occur in the subtrees are added to the use and def set of the node. Whether or not the arguments are passed by reference can be discerned from the C/C++ tree but this was not the case for the trees generated by the Ada front end. For Ada trees, an assumption is made that actuals can be passed by reference and thus be defined within those functions. Therefore the same approach is used in the C/C++ front end.

**Nodes of other Control Types**  Variables/parameters which occur in the subtrees of these nodes are added to the use sets of the nodes.
5.3 Ada 95 Front End

For the Ada front end a custom parser was built from resources found on the Ada Resource Association web site [1]. The yacc Ada 95 grammar and lexer posted there formed a simple Ada 95 syntax checker. This was the foundation of the parser. The lex file only needed a little work, some code was added to keep track of line numbers and the tokens’ original strings. Moreover, the main function (which ran the parser) was removed since the parsing was to be invoked by the Ada front end’s parse function. The grammar required more work since semantical rules needed to be added. There was also need for a simple symbol table.

Initially, the front end compiles the source code with gnatmake, the Ada 95 front end which is a part of GCC. This is to ensure that the code actually compiles. Should an error occur during this compilation the front end will abort. The front end will then push the path to the source code’s main file on a stack and then parse that file. If the parser encounters includes (Ada’s with statement) the paths to the included files will be pushed on the stack. The parsing continues until the stack is empty.

5.3.1 The Symbol Table

The Ada parser uses a symbol table. Throughout the grammar, once variables, parameters and constants are declared they are inserted into the symbol table. When a rule is used which uses a name, that name is looked up in the symbol table. A real compiler would throw an error if a symbol was undeclared, our parser is more tolerant. If a symbol is looked up and not found it is silently inserted with a new node into the symbol table. This can happen e.g. if a symbol from another file is used since the symbol table is reset every time a new file is parsed. The symbol table is reset because references to symbols beyond a function’s local scope are not considered. Some symbols are inserted into the symbol table before each file is parsed, such as the boolean constants true and false.

5.3.2 The yacc grammar

In the AST forest, each procedure should be represented by its own tree. Ada supports nested procedures which means that a parse tree from a correctly parsed Ada program can contain trees of many procedures. Generally speaking the parser will construct the tree until reducing to the non-terminal symbol for subprograms (and tasks). The tree will then be inserted into the forest. When encountering the tokens procedure, function or task a new scope will be opened. New scopes are also opened when parsing loops, protected objects and declare blocks. Node control types are set as soon as their equivalent non-terminals are parsed, thus eliminating the need to traverse the tree an extra time in the front end. The various pragma statements which are available in the Ada 95 language are not supported by the parser and are completely ignored. The grammar (and accompanying lex-file) can be found at http://www.ida.liu.se/~ernil/detect/code/.
5.3.3 Tree Examples

In this section, some examples of parse trees from the Ada front end are shown. The code in figure 5.16 gives the trees shown in figures 5.15 and 5.17.

Figure 5.15: Tree for `Main` in figure 5.16.
with Ada.Integer_Text_IO;
procedure Main is
    function Do_Stuff(I : in Integer) return Integer is
    begin
        while I > 0 loop
            null;
        end loop;

        return I;
    end Do_Stuff;

    X : Integer;
beging
    Ada.Integer_Text_IO.Get(X);
    Ada.Integer_Text_IO.Put(Do_Stuff(X));
    end Main;

Figure 5.16: Sample Ada code containing function calls, a nested function, a loop and a return statement.

Figure 5.17: Tree for Do_Stuff in figure 5.16.
5.3.4 Generalizing Control Structures

Unifying Loop Constructs

For all loops the control type is set to detect_loop. For for loops some transformations of the tree are performed. The tree structure of the for loop in Ada is quite different from its namesake in C++. In C++ the syntax more resembles the code which the compiler will generate. In Ada the syntax is more specific, see figures 5.18 and 5.19. First, the iterator variable used in the loop is declared in the surrounding statement_list node and an assignment is added to initialize that variable. A subtree which represents an incrementation (or a decrementation if the iteration was declared reverse) of the iterator is added in the After part of the detect loop structure. Then a condition is added to the Precondition part of the detect loop. The result of the generalizing of the tree in figure 5.19 is shown in figure 5.20. The code corresponding to the tree in figure 5.20 can be seen in figure 5.21. Keep in mind that this code is merely conceptual, not correct Ada 95 syntax (the for statement is erroneously written and variables must be declared before the statement part of a program). However, the APDG construction does not require that ASTs are correct parse trees of a program, this is why such transformations can be permitted.

for I in 1..10 loop
    null;
end loop;

Figure 5.18: Example of a for loop in Ada.
5.3 Ada 95 Front End

Figure 5.19: Parse tree of the code in figure 5.18.

Figure 5.20: Tree after generalizing.

```
I : Iterator;
I := 1;
for I <= 10 loop
  null;
  I := I + 1;
end loop;
```

Figure 5.21: Code corresponding to the tree in figure 5.20.

5.3.5 Replacing Case Statements

Contrary to C and C++, the `case` construct in Ada can be mapped to an equivalent `if` construct. For `case_stmt` nodes, the tree is therefore transformed. For each `when` branch a separate `elsif` is created with a condition using the expression appearing after the keyword `case` and the values listed between the keyword `when` and the `=>` symbol. The `else` branch is constructed from the `when others` branch. Figure 5.22 and 5.23 is an example of a `case` structure and what the code of the transformed tree were to look like. The expression in the `elsif` clause is not a syntactically correct expression. However, the APDG construction does not consider the structure of such conditions, only what variables are used within.
case X is
  when 3 =>
    null;
  when 5..6|9 =>
    null;
  when others =>
    null;
end case;

if X = 3 then
  null;
elsif X = 5..6|9 then
  null;
else
  null;
end if;

Figure 5.22: Example of case statement.

Figure 5.23: After transforming the parse tree in figure 5.22, we get a tree which represents the code above.

5.3.6 Lifting Up Nested Statement Lists
This is done in an analogous way to what is explained in section 5.2.9.

5.3.7 Finding Uses and Definitions in the Tree
This is done in an analogous way to what is explained in section 5.2.10.
Chapter 6

AST Analysis Module

6.1 Module Overview

The AST Analysis module consists of the classes ASTAnalysis and ASTMetrics. The ASTForest objects created by the front end are passed to the constructor of ASTAnalysis. The trees in the ASTForest are then fingerprinted, sorted descendingly according to weight and then fed into the matching algorithm together with an ASTMetrics object. Depending on the settings in the configuration file an output will then be generated by calling the appropriate output member function of the ASTMetrics object. An overview of the module is depicted in figure 6.1

6.2 Fingerprinting

As discussed in section 2.2.4, AST based techniques suffer from greater time complexities than for instance, text and token based techniques. In order to reduce the number of subtrees which have to be compared, fingerprinting has been implemented based on the method presented by Chilowicz et al. [7]. In the fingerprinting process each subtree of each tree is given a fingerprint. Each AST in the ASTForest is thus fingerprinted into a set of fingerprints. The fingerprint is a tuple containing the subtree’s weight (the number of nodes in the subtree), a hash value, a pointer to the corresponding node, and a pointer to that node’s parent. The weight is relevant since only subtrees of same weight should be compared. Two matching subtrees will have the same weight since we intend only to detect equal trees (in the isomorphic sense). Knowing this, it is reasonable to expect that our analysis will be weak against plagiarism scenarios of code insertion.

6.2.1 Hashing Subtrees

It is important that the hash function reflects the structural organization of the tree, since the analysis finds exact subtree matches of two procedures’ ASTs. Chilowicz et al. [7] described two methods for hashing in this manner, Dyck words
using a Karp-Rabin hash function and cryptographic hashing. After analyzing hash function efficiency with regard to hash collisions they concluded that both methods were close to theoretically perfect hash functions. Fingerprinting with cryptographic hashing has been used in DETECT since it is rather straightforward to implement.

Each tree is hashed bottom up in the following manner: Each node supplies a value based on its node type. Then, using a cryptographic hashing function $C$, an entire tree is hashed based on the hashes of the children of each node. So if the hash function used is called $H$, the type value for each node is given by a function $V$ and we are hashing a tree $t$ where the root $r$ has children $t_1, t_2, ... t_n$ we get:

$$H(t) = C(V(r) \cdot H(t_1) \cdot H(t_2) \cdots \cdot H(t_n))$$  \hspace{1cm} (6.1)

The dots in the equation above represent concatenation. In DETECT, MD5 [21] hashing is used by default as $C$. By manipulating the configuration parameter CRYPT_HASH_METHOD, SHA-1 [24] or SHA-256 [19] are used instead, should a longer hash value be required. The OpenSSL [20] implementation of these hash functions
6.3 Matching Algorithm

has been used. The AST analysis module can be configured to print the fingerprints with these hashes if the configuration parameter `AST_DUMP_FINGERPRINTS` is set to `TRUE`. The figures 6.2 and 6.3 show examples of such dumps.

Fingerprints for int main(int, char**)
88cac246e40cccb2623eb549d8c3cee46 w= 54 @290 statement_list
322f904d0c1a703b5db50951a9605f8b w= 30 @1 statement_list
159ab8f8c57e1b21eb749dad44f53c92 w= 25 @6 statement_list

Figure 6.2: Excerpt of dump of fingerprints using MD5 hashing. Each row presents the hash and weight of the subtree and the unique ID and type of the root of the subtree.

Fingerprints for int main(int, char**)
ce8655ac1391792c50f0b6271a58e282409f193e w= 54 @290 statement_list
e3c673681e60c5c0e465470b7ed8b40e221431a w= 30 @1 statement_list
d21dcbed0a83b4dd0ca409148d130121b213bb33 w= 25 @6 statement_list

Figure 6.3: Excerpt of dump of fingerprints using SHA-1 hashing.

6.3 Matching Algorithm

The matching algorithm in the AST analysis module takes two sets of sets of fingerprints as input and yields an `ASTMetrics` object as output. The `ASTMetrics` object contains several data: A container containing subtree matches which were found during the matching process (S in the pseudocode in figure 6.4) and a similarity matrix (M below). For each pair of procedures \(< P_1, P_2 >\) where \(P_1\) is from the first code unit and \(P_2\) is from the second, the matrix contains a similarity ratio for how similar \(P_1\) is to \(P_2\). Equation 6.2 shows how the similarity ratio is calculated.

\[
\text{Similarity} = \frac{\text{Number of subtrees in } P_1 \text{ which had matches in } P_2}{\text{Total weight of } P_1} \tag{6.2}
\]

The `ASTMetrics` object has a separate container for the pairs of procedures \(< P_1, P_2 >\) which have the highest similarity ratio (called G below). That container also stores the similarity ratio of those pairs. If there are several pairs which have the highest similarity, the pair where the tree of \(P_1\) is of greatest weight is stored.

In the pseudocode in figure 6.4, the weight of a fingerprint \(F\) is denoted \(F.weight\), the hash of a fingerprint as \(F.hash\) and the corresponding node of a fingerprint as \(F.node\). A subtree match is a pair of fingerprints \(< F_1, F_2 >\), where \(F_1\) is a fingerprint from an AST from code unit 1 and \(F_2\) is a fingerprint from an AST from code unit 2. The set \(T\) is for keeping track of which nodes in one fingerprint set (the fingerprints of one procedure) have matches in another fingerprint set.
The constant **thresh** is a threshold value which can be set in the configuration file by the key `AST_WEIGHT_THRESHOLD`. Subtrees which are of weight below **thresh** will not be recorded in S in the matching algorithm, however they are counted for calculating the similarity (and therefore are stored in T). The corresponding procedure of a fingerprint set $S_i$ is denoted as $S_i.proc$.

**Input:** Two sets of fingerprint sets $C_a$ and $C_b$.

**Output:** Set of subtree matches $S$, similarity matrix $M$, set of pairs of most similar procedures $G$.

1: $S := \emptyset$
2: for each set $S_a$ in $C_a$ do
3:   for each set $S_b$ in $C_b$ do
4:     $T := \emptyset$
5:     for each fingerprint $F_a$ in $S_a$ do
6:       for each fingerprint $F_b$ in $S_b$ do
7:         if $F_b.weight = F_a.weight$ then
8:           if $F_b.hash = F_a.hash$ then
9:             if $Equals(F_a.node, F_b.node)$ then
10:                $T := T \cup F_a.node$
11:             end if
12:           end if
13:         end if
14:       end for
15:     end for
16:     end if
17:     $S := S \cup F_a,F_b>$
18: end for
19: $M_{S_a.proc,S_a.proc} := |T| / |S_a|$
20: end for
21: $G := G \cup <S_a.proc,S_x.proc>$, where $x$ maximizes $M_{S_a.proc,S_x.proc}$
22: end for

Figure 6.4: Pseudocode for the matching algorithm.

Two equal subtrees cannot give two different hashes, thus the condition in line 8 suffices to prove that two trees are unequal. However, since hash collisions are possible (however unlikely) it is not enough to prove that two trees are equal. The $Equals$ function on line 9 takes the roots of two subtrees and confirms whether or not they are in fact equal. The function makes a depth first traversal of both trees checking in each node that the types of the nodes are equal and that the labels of the children are equal.

Subtree matches are stored in $S$. However, it is not desirable to find all subtree matches. If a subtree match $<t_a,t_b>$ is found, obviously the matches for each subtree in $t_a$ and $t_b$ are also found. We call those matches submatches. These matches are redundant and we are not interested in keeping them. Therefore, when a new subtree match is added to $S$ in line 12, we must check that it is not
a submatch. For this purpose, a set called $U$ is used (this is omitted from the pseudocode for the sake of readability). There is one such set $U$ for each pair of procedures $<P_1, P_2>$ where $P_1$ is from the first code unit and $P_2$ is from the second. If the parents of $t_a$ and $t_b$ appear in subtree matches in $U$, then the subtree match $<t_a, t_b>$ is added only to $U$. Otherwise, the subtree match is not a submatch and is therefore added to both $S$ and $U$.

### 6.4 Output of the Module

The AST analysis provides three types of output, the similarity matrix, the pair of procedures of greatest similarity and a list of all subtree matches. The output can be controlled by manipulating the configuration parameters in table 6.1.

<table>
<thead>
<tr>
<th>Key</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AST_MATRIX_OUTPUT</td>
<td>boolean</td>
<td>Enable the output of the similarity matrix.</td>
</tr>
<tr>
<td>ENABLE_COLORFUL_OUTPUT</td>
<td>boolean</td>
<td>Use colors for highlighting ratios in the similarity matrix output.</td>
</tr>
<tr>
<td>COLOR_RED_THRESHOLD</td>
<td>integer</td>
<td>Set the threshold ratio for red highlighting in the similarity matrix output.</td>
</tr>
<tr>
<td>COLOR_YELLOW_THRESHOLD</td>
<td>integer</td>
<td>Set the threshold ratio for yellow highlighting in the similarity matrix output.</td>
</tr>
<tr>
<td>AST_ALL_MATCHES_OUTPUT</td>
<td>boolean</td>
<td>Enable the output of the list of subtree matches.</td>
</tr>
<tr>
<td>AST_WEIGHT_THRESHOLD</td>
<td>integer</td>
<td>Set the weight threshold for subtree matches which should be included in the List of Subtree Matches output (see section 6.4.3).</td>
</tr>
<tr>
<td>AST_GREATEST_OUTPUT</td>
<td>boolean</td>
<td>Enable the output of pairs of greatest similarity.</td>
</tr>
<tr>
<td>AST_DETAILED_OUTPUT</td>
<td>boolean</td>
<td>See section 6.4.1</td>
</tr>
</tbody>
</table>

Table 6.1: Configuration parameters for output of AST analysis. The default settings can be found in appendix C.

#### 6.4.1 Pairs of Greatest Similarity

For each procedure of the first input unit, this output presents the greatest similarity found, and which procedure in the second input unit which the procedure from the first was similar to. When AST_DETAILED_OUTPUT is set to TRUE, the output is a table with one row per procedure in input unit 1. The table has the following columns: Number of matched nodes in the tree ($|T|$ in the pseudocode in figure 6.4), the total number of nodes in the tree of the procedure, the similarity ratio (the ratio of the former two), the name (file and function name) of the
procedure in code unit 1 and the name of the matched procedure in code unit 2. When AST_DETAILED_OUTPUT is set to FALSE, this is simply a list of similarity ratios (as percentages) separated by spaces and sorted descendingly. In the example of output in figure 6.5, code unit 1 had four procedures and code unit 2 had one procedure.

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>81</td>
<td>100</td>
<td>81.0%</td>
<td>orig21.cc:int main(int, char**)</td>
<td>plag21.cc:int main(int, char**)</td>
</tr>
<tr>
<td>11</td>
<td>19</td>
<td>57.9%</td>
<td>orig21.cc:int factorial(int)</td>
<td>plag21.cc:int main(int, char**)</td>
</tr>
<tr>
<td>11</td>
<td>20</td>
<td>55.0%</td>
<td>orig21.cc:int power(int, int)</td>
<td>plag21.cc:int main(int, char**)</td>
</tr>
<tr>
<td>4</td>
<td>8</td>
<td>50.0%</td>
<td>orig21.cc:int charToInt(char)</td>
<td>plag21.cc:int main(int, char**)</td>
</tr>
</tbody>
</table>

Figure 6.5: Example of output of pairs of greatest similarity with detail.

The nondetailed version of the output in figure 6.5 is simply:

81.0 57.9 55.0 50.0

This output format is useful for determining how similar one code unit is to another code unit. For example, the undetailed output can be directly sent to a script which calculates the average, median, maximum and minimum of these values.

6.4.2 Similarity Matrix

The similarity matrix presents the similarity ratio for each procedure pair. Thus the ratios in the pairs of greatest similarity output can be found in the similarity matrix. In figure 6.6, both code units had five procedures.

<p>| | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>100.0%</td>
<td>30.0%</td>
<td>94.4%</td>
<td>33.3%</td>
<td>2.8%</td>
<td>4.3%</td>
<td>95.7%</td>
<td>55.3%</td>
<td>8.5%</td>
<td>51.1%</td>
</tr>
<tr>
<td>14.4%</td>
<td>30.1%</td>
<td>97.3%</td>
<td>19.2%</td>
<td>37.0%</td>
<td>18.2%</td>
<td>22.7%</td>
<td>40.9%</td>
<td>100.0%</td>
<td>36.4%</td>
</tr>
<tr>
<td>1.0%</td>
<td>37.8%</td>
<td>70.4%</td>
<td>33.7%</td>
<td>92.9%</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 6.6: Example of output of similarity matrix.
Since the names of the procedures can be long, the indices of the matrix have symbols, one for each procedure name. The matrix is sorted so that the greatest similarities can be found along the diagonal of the matrix. If the terminal emulator in which the output is printed supports colors (ANSI escape sequences [10]) and the configuration parameter `ENABLE_COLORFUL_OUTPUT` is set to `TRUE`, ratios above the the value of `COLOR_RED_THRESHOLD` are colored red, ratios between the value of `COLOR_YELLOW_THRESHOLD` and `COLOR_RED_THRESHOLD` are colored yellow and ratios below `COLOR_YELLOW_THRESHOLD` are colored green. By default the values for the red and yellow threshold are 98% and 90% respectively.

### 6.4.3 List of Subtree Matches

The former two output variants focus on the similarity ratio. However, in some cases the similarity ratio may be uninformative or misleading. Since each subtree in the first tree which has a match in the other tree will be counted, leaves in the tree will match very often. One example is identifier nodes, which appear as leaves in the tree. These nodes almost always have a match in the other tree since it is very rare for the other tree to lack identifier nodes. Thus, small procedures will frequently match each other, generating approximately 100% similarity ratios. Therefore, the similarity of procedures is only interesting if the procedures’ trees have enough weight. This is why the weight and total weight is included in the pairs of greatest similarity output.

Another example of where the similarity ratio may be misleading is the case where only a part of a procedure is plagiarized. Even if the part which has been plagiarized is of substantial weight the similarity ratio will be off, since it is based on the total weight of the procedure.

Considering this, a third kind of output is presented for this module, the list of subtree matches. The list contains all subtree matches (which were of weight greater than `AST_WEIGHT_THRESHOLD`) which were found during the matching process. A list of subtree matches are output for each pair of procedures and these lists are sorted descendingly by weight. The weight of the match, the names of the matching procedures and the approximate line numbers of the match are output.

For this output to be helpful `AST_WEIGHT_THRESHOLD` should be set sufficiently high (perhaps to tree weights of 40 or 50) so that leaves and small subtrees are not matched. If it, for instance, is set to 1, each identifier node in the trees of the first code unit will match all identifier nodes in the second code unit, thus leading to an explosion of matches. Figure 6.7 shows an example of this output.
If the configuration parameter `AST_DETAILED_OUTPUT` is set to false, this output is reduced to only the weights of the subtree matches printed on a single line in descending order. For the example in figure 6.7, it would look as follows:

59 27 26 23
Chapter 7

Method of Evaluation

We have chosen Moss (see section 2.5) for comparison to detect since it is a realistic alternative to our system. It supports multiple languages, is easy to use and is free. In this chapter, and in chapter 8 and 9, we compare detect’s AST analysis module (the AST diff) to Moss.

In the comparison to Moss, both detect and Moss have been run on a set of benchmark source files. Unfortunately the developers of Moss have not presented any scenarios or data with which we can compare. Because of this, the set of benchmark files were designed and created by me in collaboration with Niklas Holma. The files make up a set of pairs where each pair tests a certain plagiarism scenario. In each pair there is one original and one plagiarized file (with a few exceptions where some scenarios share files). This was a convenient way of setting up the tests, since Moss calculates similarity between files. C++ was chosen as the source language for the benchmark files. One set of equivalent benchmark files for C and another for Ada have also been constructed. These are used to illustrate any differences between the front ends in detect. The files have approximately 50 to 100 lines of code and are constructed so that as much code as possible is part of the test cases. This is in order to simplify the comparison and get clear results.

7.1 Plagiarism Scenarios

Table 7.1 contains a summary of the benchmark files. They are ordered by plagiarism type. Each scenario contains a number of cases. In many scenarios the plagiarism has not changed the semantics of the code. To detect how well the plagiarism system can deal with false positives some scenarios are included where the semantics were changed by unsystematic or “reckless” plagiarism. These scenarios are not plagiarisms in the strict sense (see our definition of plagiarism in section 1.2). The benchmark files can be found at http://www.ida.liu.se/~ernil/detect/bench/.
7.1.1 Format Alteration Scenarios

In scenario 1, the indentation of the plagiarized file has been changed. The indentation was changed from two spaces to one tab for C/C++ and from three spaces to one tab for Ada. In scenario 2, the positions of braces have been changed. An example is shown in figures 7.1 and 7.2. Since braces are not used in Ada, the positions of the is, then, loop keywords have been changed in a similar manner. In scenario 3, unnecessary braces have been removed (such as braces on if branches containing only one statement). This scenario is not applicable for Ada. Scenario 4, 5 and 6 manipulate the comments of a file. These scenarios share original file and plagiarism file.

```c
void main() {
    if (T) {
        S;
    } else {
        S;
    }
}
```

Figure 7.1: Original for scenario 2.

```c
void main() {
    if (T) {
        S;
    } else {
        S;
    }
}
```

Figure 7.2: Plagiarism for scenario 2.

7.1.2 Identifier Renaming Scenarios

In scenarios 7 and 8 variable names have been replaced. In scenario 7, each variable was systematically renamed throughout the file in order to preserve the semantics of the code. In scenario 8, each variable name was substituted with an arbitrary identifier, albeit one belonging to a declared variable. In scenario 9, parameter names were replaced in 5 functions. Here a mix of long variable names (more than 20 characters) and short names (only one or two characters) were used. In scenario 10 function names have been replaced. This was done systematically, for each function which was renamed, also the calls to that function were replaced by that name. In scenario 11 each function name (declarations and calls) was replaced by an arbitrary one, albeit one which was declared. This was done in 8 functions which all took one argument of the same type, in order to get maximum alteration of the semantics. After the replacement of function names, one such function became recursive. In scenario 12 typedefs have been introduced to replace declarations which used templates. 14 cases where template types were used were replaced by typedefs. Since this had no equivalent in C and Ada, this scenario is only applicable for C++.
7.1 Plagiarism Scenarios

7.1.3 Declaration Reordering Scenarios
In scenario 13, the order of function definitions have been changed. This was done for 8 functions (4 pairs were swapped). In scenario 14, the order of parameters have been changed. This was done in 4 functions. One function had two integer arguments, two functions had 11 integer arguments and one function had five arguments of different primitive types. The main function (not one of the 4) had calls to these functions. In scenario 15 the order of the formal parameters of two functions was changed where the order of the actuals was not, in the other two cases it was the opposite.

7.1.4 Statement Reordering Scenarios
In scenario 16, the order of variable definitions has been changed. This was done in 4 functions. In this scenario only statements which did not depend on one another were rearranged. In scenario 17, these dependencies were disregarded when rearranging the variable definitions. In scenario 18, the order of switch cases has been changed. This was done for 6 switch cases, three pairs were swapped. In scenario 19, variable definitions inside loops have been moved outside the loop and vice versa (where it did not affect the semantics of the code). Since declarations must come before the statement part of a program in Ada this case was not applicable for Ada.

7.1.5 Code Insertion Scenarios
In scenario 20, statements have been inserted in the code which did not affect the semantics, 13 statements were inserted. In scenario 21, the same number of statements were inserted, but these statements did affect the semantics of the code. In scenario 22 and 23, control structures (7 and 10 structures respectively) have been inserted which did not affect the semantics. For example, placing copied code in a control structure which always executes once or introducing loops which contain code which will never execute. In scenario 24, constants have been inserted into 10 expressions. For example multiplying an arithmetic expression by one, or adding 0 or adding one and then subtracting one. In scenario 25, extra parentheses have been added to 10 expressions. In some expressions this changed the order of evaluation but no changes were made to the semantics of the code.

7.1.6 Control Replacement Scenarios
In scenario 26, if branches have been swapped by negating conditions. Double negations were also introduced in one if statement. In scenario 27, if statements have been replaced by switch statements and vice versa. In scenario 28, loops have been replaced. Since there are three loop constructs in C/C++, and Ada, in total 6 constructs were replaced.
7.1.7 Scenarios of Other Methods

Some special cases which did not really fit into the classifications above have also been constructed. In scenario 29, functions have been inlined, this case also doubles for placing code in separated functions. This was done for 3 functions. In scenario 30, unnecessary parentheses have been removed from expressions which had a surplus of parentheses. In scenario 31, the \(+=, -=, *=\) and \(/=\) operators have been exchanged for equivalent “regular” assignment statements. This case is not applicable for Ada since these operators do not exist for that language. Scenario 24, 25, 30 and 31 shared original and plagiarism files. In these scenarios each scenario is in a separate function, so the similarity rating for these scenarios are the similarity ratings of those functions.

7.2 Running Moss on the Scenarios

For each scenario, the original and plagiarized files were submitted to Moss. The similarity ratio of these files was recorded. Then the passages of code which Moss highlighted as matches were examined to evaluate how many of the test cases had been detected. Since the AST diff in Detect does not calculate similarity depending on matched lines, the matched lines output of Moss was not considered for the comparison.

7.3 Running the AST Diff on the Scenarios

For each scenario, the original and plagiarized files were sent as input arguments to Detect with only the AST analysis module enabled. The contents of the config file is shown in appendix D. The list of subtree matches was used to evaluate how many of the test cases had been detected. For comparison to Moss’s similarity ratio, the average of the ratios from the pairs of greatest similarity output was calculated as an average of the entire file. To obtain two similarity ratios (one for how similar A is to B and one for the opposite), Detect was run twice: Once with original file as input unit 1 and plagiarized file as input unit 2, and a second time with the arguments reversed. This process was performed for all three supported languages.
### Table 7.1: Summary of the benchmark files with plagiarism scenarios. See section 1.3 for the definitions of plagiarism types.

<table>
<thead>
<tr>
<th>No</th>
<th>Name</th>
<th>Type</th>
<th>Cases</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Changing indentation of code</td>
<td>FA</td>
<td>2 functions</td>
</tr>
<tr>
<td>2</td>
<td>Change position of braces</td>
<td>FA</td>
<td>16 braces</td>
</tr>
<tr>
<td>3</td>
<td>Remove unnecessary braces</td>
<td>FA</td>
<td>14 braces</td>
</tr>
<tr>
<td>4</td>
<td>Add comments</td>
<td>FA</td>
<td>12 comments</td>
</tr>
<tr>
<td>5</td>
<td>Remove comments</td>
<td>FA</td>
<td>12 comments</td>
</tr>
<tr>
<td>6</td>
<td>Alter comments</td>
<td>FA</td>
<td>12 comments</td>
</tr>
<tr>
<td>7</td>
<td>Systematically replace var. names</td>
<td>IR</td>
<td>17 variables</td>
</tr>
<tr>
<td>8</td>
<td>Unsystematically replace var. names</td>
<td>IR</td>
<td>17 variables</td>
</tr>
<tr>
<td>9</td>
<td>Systematically replace param. names</td>
<td>IR</td>
<td>5 functions</td>
</tr>
<tr>
<td>10</td>
<td>Systematically replace function names</td>
<td>IR</td>
<td>8 functions</td>
</tr>
<tr>
<td>11</td>
<td>Unsystematically replace function names</td>
<td>IR</td>
<td>8 functions</td>
</tr>
<tr>
<td>12</td>
<td>Replace templates by typedefs</td>
<td>IR</td>
<td>14 types</td>
</tr>
<tr>
<td>13</td>
<td>Change order of function def’s</td>
<td>DR</td>
<td>8 functions</td>
</tr>
<tr>
<td>14</td>
<td>Systematically change order of param’s</td>
<td>DR</td>
<td>4 functions</td>
</tr>
<tr>
<td>15</td>
<td>Unsystematically change order of param’s</td>
<td>DR</td>
<td>4 functions</td>
</tr>
<tr>
<td>16</td>
<td>Systematically change order of var. def’s</td>
<td>SR</td>
<td>4 functions</td>
</tr>
<tr>
<td>17</td>
<td>Unsystematically change order of var. def’s</td>
<td>SR</td>
<td>4 functions</td>
</tr>
<tr>
<td>18</td>
<td>Change order of switch cases</td>
<td>SR</td>
<td>6 branches</td>
</tr>
<tr>
<td>19</td>
<td>Move var. def’s inside/outside loop</td>
<td>SR</td>
<td>8 variables</td>
</tr>
<tr>
<td>20</td>
<td>“Careful” insertion of statements</td>
<td>CI</td>
<td>13 statements</td>
</tr>
<tr>
<td>21</td>
<td>“Reckless” insertion of statements</td>
<td>CI</td>
<td>13 statements</td>
</tr>
<tr>
<td>22</td>
<td>Insertion of control structures (exec. once)</td>
<td>CI</td>
<td>7 structures</td>
</tr>
<tr>
<td>23</td>
<td>Insertion of control structures (exec. never)</td>
<td>CI</td>
<td>10 structures</td>
</tr>
<tr>
<td>24</td>
<td>Insertion of constants in expressions</td>
<td>CI</td>
<td>10 expr’s</td>
</tr>
<tr>
<td>25</td>
<td>Insertion of parentheses in expressions</td>
<td>CI</td>
<td>10 expr’s</td>
</tr>
<tr>
<td>26</td>
<td>Negate conditions in if statements</td>
<td>CR</td>
<td>6 ifs</td>
</tr>
<tr>
<td>27</td>
<td>Replace ifs by switches and vice/versa</td>
<td>CR</td>
<td>6 if/switches</td>
</tr>
<tr>
<td>28</td>
<td>Replace loops by equivalents</td>
<td>CR</td>
<td>6 loops</td>
</tr>
<tr>
<td>29</td>
<td>Inlining functions</td>
<td>OM</td>
<td>3 functions</td>
</tr>
<tr>
<td>30</td>
<td>Remove unnecessary parentheses</td>
<td>OM</td>
<td>3 expr’s</td>
</tr>
<tr>
<td>31</td>
<td>Exchanging += for equiv. assignment</td>
<td>OM</td>
<td>4 expr’s</td>
</tr>
</tbody>
</table>
Chapter 8

Analysis Results

8.1 Moss compared to the AST Diff

The results from running Detect’s AST diff and Moss on the benchmark files with plagiarism scenarios are shown in Table 8.1. The reader should focus on the columns labeled Detect and Moss (so does the rest of this section). These figures represent how many of the cases were found in matches detected by the systems.

The ratios in the tables warrant special consideration. Keep in mind that the similarity ratios given by Moss and Detect represent different things. For Moss it represents how much of one file (number of lines) was matched in the other file in relation to the total number of lines in the (first) file. For Detect it is the average of the similarity ratios (as explained in section 6.3) of pairs of procedures of greatest similarity. Since the functions in the benchmark files are rather small, the similarity ratios suffer from the drawbacks mentioned in section 6.4.3. Still, they are included in the table as some indication of how similar the files were. Here are some scenario-specific comments:

**Scenario 14 and 15** Both Detect and Moss detected cases of reordered formal parameters but were confused by the reordering of actual parameters in a function call.

**Scenario 24** This unexpected result was due to constant folding performed by the GCC C/C++ front end before the tree was dumped to file by the plugin.

**Scenario 25** A closer look at the matches found by Detect showed that those matches were the ones in which the order of evaluation of the expressions had not been changed.

**Scenario 31** The GCC C/C++ front end does not treat operators like += any different from a usual assignment. Therefore it is not surprising that such plagiarism scenarios are detected without fail.
Table 8.1: Results from running detect’s AST diff and Moss on the benchmark files. The columns labeled detect and Moss describe how many cases were found for each plagiarism scenario by detect and Moss respectively. The leftmost columns Sim1 and Sim2 are the similarity ratios of the benchmark files and calculated as explained in section 7.3. The rightmost columns Sim1 and Sim2 are the similarity ratios calculated by Moss.

For the systematic changes, (i.e. excluding scenarios 8, 11, 15, 17, 21) the total number of detected cases and the number of cases detected per plagiarism category has been summarized in table 8.2. The chart in figure 8.1 shows the ratios of how many cases detect’s AST diff and Moss detected as plagiarisms in relation to the total number of test cases for each plagiarism category. For the unsystematic and reckless scenarios, the results are summarized in table 8.3. The ratios of number of detected cases to number of total cases is shown in figure 8.2. For these scenarios, the notable result is the difference in detected cases for identifier renaming, this is
8.2 Comparison of the Front Ends

discussed in section 9.2.

<table>
<thead>
<tr>
<th>DETECT</th>
<th>Moss</th>
<th>Total Number of Cases</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cases of FA</td>
<td>68</td>
<td>48</td>
</tr>
<tr>
<td>Cases of IR</td>
<td>30</td>
<td>30</td>
</tr>
<tr>
<td>Cases of DR</td>
<td>11</td>
<td>7</td>
</tr>
<tr>
<td>Cases of SR</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>Cases of CI</td>
<td>12</td>
<td>10</td>
</tr>
<tr>
<td>Cases of CR</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Cases of OM</td>
<td>7</td>
<td>3</td>
</tr>
<tr>
<td>Total Cases Detected</td>
<td>128</td>
<td>100</td>
</tr>
</tbody>
</table>

Table 8.2: Number of cases detected by detect’s AST diff and Moss in scenarios where the semantics of the code was preserved. In total and by plagiarism category.

<table>
<thead>
<tr>
<th>DETECT</th>
<th>Moss</th>
<th>Total Number of Cases</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cases of IR</td>
<td>8</td>
<td>25</td>
</tr>
<tr>
<td>Cases of DR</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>Cases of SR</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>Total Cases Detected</td>
<td>11</td>
<td>30</td>
</tr>
</tbody>
</table>

Table 8.3: Number of cases detected by detect’s AST diff and Moss. In total and by plagiarism category for scenarios where the semantics of the code was changed (scenarios 8, 11, 15, 17 and 21).

8.2 Comparison of the Front Ends

The two equivalent sets of benchmark files for C and Ada were used to evaluate any differences between detect’s front ends. The results are shown in table 8.4. It is not surprising that there are slight differences in the results for some scenarios, since the trees used in the tree matching algorithm come from different sources. For instance, because the C/C++ trees had some information which the Ada tree was lacking, a false positive was detected for Ada in scenario 16 and 17. Moreover, the Ada tree comes straight from the parser and (in contrast to the C/C++ trees) has not undergone any slight optimizations. Because of this, cases of scenario 24, 25, and 30 were not detected when running on the Ada files.

The number of cases detected for the three benchmark file sets are summarized in table 8.5, in total and by plagiarism category. In figure 8.3, the ratios of detected cases divided by the number of total cases for that plagiarism category are presented.
Figure 8.1: Ratios of detected cases / total cases for DETECT’s AST diff and Moss in scenarios where the semantics of the code was preserved. In total and by plagiarism category.

Figure 8.2: Ratios of detected cases / total cases for DETECT’s AST diff and Moss in scenarios where semantics of the code was changed (scenarios 8, 11, 15, 17 and 21).
8.2 Comparison of the Front Ends

<table>
<thead>
<tr>
<th></th>
<th>C++</th>
<th></th>
<th></th>
<th>C</th>
<th></th>
<th></th>
<th>Ada</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>CD</td>
<td>Sim1</td>
<td>Sim2</td>
<td>CD</td>
<td>Sim1</td>
<td>Sim2</td>
<td>CD</td>
<td>Sim1</td>
<td>Sim2</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>100.00%</td>
<td>100.00%</td>
<td>2</td>
<td>100.00%</td>
<td>100.00%</td>
<td>2</td>
<td>100.00%</td>
<td>100.00%</td>
</tr>
<tr>
<td>2</td>
<td>16</td>
<td>100.00%</td>
<td>100.00%</td>
<td>16</td>
<td>100.00%</td>
<td>100.00%</td>
<td>16</td>
<td>100.00%</td>
<td>100.00%</td>
</tr>
<tr>
<td>3</td>
<td>14</td>
<td>100.00%</td>
<td>100.00%</td>
<td>14</td>
<td>100.00%</td>
<td>n/a</td>
<td>16</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>4</td>
<td>12</td>
<td>100.00%</td>
<td>100.00%</td>
<td>12</td>
<td>100.00%</td>
<td>100.00%</td>
<td>12</td>
<td>100.00%</td>
<td>100.00%</td>
</tr>
<tr>
<td>5</td>
<td>12</td>
<td>100.00%</td>
<td>100.00%</td>
<td>12</td>
<td>100.00%</td>
<td>100.00%</td>
<td>12</td>
<td>100.00%</td>
<td>100.00%</td>
</tr>
<tr>
<td>6</td>
<td>12</td>
<td>100.00%</td>
<td>100.00%</td>
<td>12</td>
<td>100.00%</td>
<td>100.00%</td>
<td>12</td>
<td>100.00%</td>
<td>100.00%</td>
</tr>
<tr>
<td>7</td>
<td>17</td>
<td>100.00%</td>
<td>100.00%</td>
<td>17</td>
<td>100.00%</td>
<td>100.00%</td>
<td>17</td>
<td>100.00%</td>
<td>100.00%</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>80.70%</td>
<td>81.90%</td>
<td>0</td>
<td>87.80%</td>
<td>88.70%</td>
<td>0</td>
<td>88.50%</td>
<td>88.50%</td>
</tr>
<tr>
<td>9</td>
<td>5</td>
<td>100.00%</td>
<td>100.00%</td>
<td>5</td>
<td>100.00%</td>
<td>100.00%</td>
<td>5</td>
<td>100.00%</td>
<td>100.00%</td>
</tr>
<tr>
<td>10</td>
<td>8</td>
<td>100.00%</td>
<td>100.00%</td>
<td>8</td>
<td>100.00%</td>
<td>100.00%</td>
<td>8</td>
<td>100.00%</td>
<td>100.00%</td>
</tr>
<tr>
<td>11</td>
<td>8</td>
<td>98.89%</td>
<td>98.83%</td>
<td>7</td>
<td>95.56%</td>
<td>91.43%</td>
<td>8</td>
<td>100.00%</td>
<td>100.00%</td>
</tr>
<tr>
<td>12</td>
<td>0</td>
<td>84.53%</td>
<td>87.90%</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>13</td>
<td>8</td>
<td>100.00%</td>
<td>100.00%</td>
<td>8</td>
<td>100.00%</td>
<td>100.00%</td>
<td>8</td>
<td>100.00%</td>
<td>100.00%</td>
</tr>
<tr>
<td>14</td>
<td>3</td>
<td>99.06%</td>
<td>99.06%</td>
<td>3</td>
<td>95.30%</td>
<td>95.30%</td>
<td>3</td>
<td>95.46%</td>
<td>95.43%</td>
</tr>
<tr>
<td>15</td>
<td>3</td>
<td>99.06%</td>
<td>99.06%</td>
<td>3</td>
<td>95.30%</td>
<td>95.30%</td>
<td>3</td>
<td>95.46%</td>
<td>95.43%</td>
</tr>
<tr>
<td>16</td>
<td>0</td>
<td>92.62%</td>
<td>93.12%</td>
<td>1</td>
<td>94.40%</td>
<td>94.40%</td>
<td>0</td>
<td>89.80%</td>
<td>89.80%</td>
</tr>
<tr>
<td>17</td>
<td>0</td>
<td>93.12%</td>
<td>93.12%</td>
<td>0</td>
<td>95.12%</td>
<td>95.12%</td>
<td>0</td>
<td>89.80%</td>
<td>89.80%</td>
</tr>
<tr>
<td>18</td>
<td>0</td>
<td>97.10%</td>
<td>97.10%</td>
<td>0</td>
<td>98.50%</td>
<td>98.50%</td>
<td>0</td>
<td>96.20%</td>
<td>96.20%</td>
</tr>
<tr>
<td>19</td>
<td>0</td>
<td>76.10%</td>
<td>70.62%</td>
<td>0</td>
<td>78.12%</td>
<td>64.30%</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>20</td>
<td>0</td>
<td>86.05%</td>
<td>72.65%</td>
<td>0</td>
<td>90.75%</td>
<td>77.15%</td>
<td>0</td>
<td>86.55%</td>
<td>78.33%</td>
</tr>
<tr>
<td>21</td>
<td>0</td>
<td>85.92%</td>
<td>77.73%</td>
<td>0</td>
<td>90.70%</td>
<td>83.25%</td>
<td>0</td>
<td>82.03%</td>
<td>75.23%</td>
</tr>
<tr>
<td>22</td>
<td>0</td>
<td>87.48%</td>
<td>78.45%</td>
<td>0</td>
<td>91.72%</td>
<td>87.07%</td>
<td>0</td>
<td>87.85%</td>
<td>68.58%</td>
</tr>
<tr>
<td>23</td>
<td>0</td>
<td>88.70%</td>
<td>73.92%</td>
<td>0</td>
<td>91.60%</td>
<td>80.35%</td>
<td>0</td>
<td>85.47%</td>
<td>58.77%</td>
</tr>
<tr>
<td>24</td>
<td>0</td>
<td>100.00%</td>
<td>100.00%</td>
<td>10</td>
<td>100.00%</td>
<td>100.00%</td>
<td>0</td>
<td>75.00%</td>
<td>68.60%</td>
</tr>
<tr>
<td>25</td>
<td>2</td>
<td>87.20%</td>
<td>87.20%</td>
<td>2</td>
<td>64.10%</td>
<td>64.10%</td>
<td>0</td>
<td>88.20%</td>
<td>90.80%</td>
</tr>
<tr>
<td>26</td>
<td>0</td>
<td>85.50%</td>
<td>86.40%</td>
<td>0</td>
<td>87.30%</td>
<td>87.70%</td>
<td>0</td>
<td>69.80%</td>
<td>70.30%</td>
</tr>
<tr>
<td>27</td>
<td>0</td>
<td>92.70%</td>
<td>87.60%</td>
<td>0</td>
<td>91.80%</td>
<td>86.70%</td>
<td>0</td>
<td>79.00%</td>
<td>79.50%</td>
</tr>
<tr>
<td>28</td>
<td>0</td>
<td>88.00%</td>
<td>84.30%</td>
<td>0</td>
<td>91.60%</td>
<td>88.50%</td>
<td>0</td>
<td>81.60%</td>
<td>81.60%</td>
</tr>
<tr>
<td>29</td>
<td>0</td>
<td>57.62%</td>
<td>74.00%</td>
<td>0</td>
<td>61.50%</td>
<td>75.20%</td>
<td>0</td>
<td>53.95%</td>
<td>70.80%</td>
</tr>
<tr>
<td>30</td>
<td>3</td>
<td>100.00%</td>
<td>100.00%</td>
<td>3</td>
<td>100.00%</td>
<td>100.00%</td>
<td>0</td>
<td>85.70%</td>
<td>85.70%</td>
</tr>
<tr>
<td>31</td>
<td>4</td>
<td>100.00%</td>
<td>100.00%</td>
<td>4</td>
<td>100.00%</td>
<td>100.00%</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
</tr>
</tbody>
</table>

Table 8.4: The result from running detect’s AST diff on the three sets of benchmark files. CD = Cases Detected. The similarity ratios Sim1 and Sim2 are calculated as explained in section 7.3. The scenarios which were not applicable for C or Ada have been marked as n/a.
Table 8.5: Number of cases detected for the three benchmark file sets, by plagiarism category and in total. The columns labeled C++, C and Ada show the number of detected cases for the C++, C and Ada front ends respectively. The columns labeled Total show the total number of test cases for each plagiarism category for each front end.

<table>
<thead>
<tr>
<th></th>
<th>C++</th>
<th>Total</th>
<th>C</th>
<th>Total</th>
<th>Ada</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cases of FA</td>
<td>68</td>
<td>68</td>
<td>68</td>
<td>68</td>
<td>54</td>
<td>54</td>
</tr>
<tr>
<td>Cases of IR</td>
<td>38</td>
<td>69</td>
<td>38</td>
<td>55</td>
<td>38</td>
<td>55</td>
</tr>
<tr>
<td>Cases of DR</td>
<td>14</td>
<td>16</td>
<td>14</td>
<td>16</td>
<td>14</td>
<td>16</td>
</tr>
<tr>
<td>Cases of SR</td>
<td>0</td>
<td>22</td>
<td>1</td>
<td>22</td>
<td>0</td>
<td>14</td>
</tr>
<tr>
<td>Cases of CI</td>
<td>12</td>
<td>63</td>
<td>12</td>
<td>63</td>
<td>0</td>
<td>63</td>
</tr>
<tr>
<td>Cases of CR</td>
<td>0</td>
<td>18</td>
<td>0</td>
<td>18</td>
<td>0</td>
<td>18</td>
</tr>
<tr>
<td>Cases of OM</td>
<td>7</td>
<td>10</td>
<td>7</td>
<td>10</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>Total Cases</td>
<td>139</td>
<td>266</td>
<td>140</td>
<td>252</td>
<td>106</td>
<td>226</td>
</tr>
</tbody>
</table>

Figure 8.3: Ratios of detected cases / total number of cases for each plagiarism category.
8.3 Execution Time

An experiment was performed to gauge the execution time of the AST analysis in DETECT. A set of source files of increasing size were analyzed. The files contain one procedure with a sequence of common language constructs, such as ifs and loops. Each file was run against itself in DETECT. This is the worst case scenario since similar code units have more subtree matches. The files can be found at http://www.ida.liu.se/~ernil/detect/time/. The result is shown in table 8.6 and in figure 8.4.

<table>
<thead>
<tr>
<th>Lines</th>
<th>Tree Weight</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td>135</td>
<td>1.060</td>
</tr>
<tr>
<td>101</td>
<td>262</td>
<td>1.479</td>
</tr>
<tr>
<td>151</td>
<td>380</td>
<td>2.856</td>
</tr>
<tr>
<td>202</td>
<td>514</td>
<td>6.819</td>
</tr>
<tr>
<td>253</td>
<td>642</td>
<td>15.262</td>
</tr>
<tr>
<td>299</td>
<td>748</td>
<td>26.815</td>
</tr>
<tr>
<td>352</td>
<td>882</td>
<td>52.970</td>
</tr>
<tr>
<td>401</td>
<td>1009</td>
<td>88.536</td>
</tr>
</tbody>
</table>

Table 8.6: Execution times for the AST analysis in DETECT. Note that the source files used are constructed to represent the worst possible case. Since the two code units come from the same source code, subtree matches are frequently detected. In such cases, there is very little benefit from fingerprinting subtrees.

Figure 8.4: Execution time for the AST analysis in DETECT. The execution time is dependent upon the number of procedures in the code units and the weight of the ASTs of those procedures. As we can see in table 8.6, the number of nodes in the trees of the tested source files is proportional to the number of lines of code in those files.
Chapter 9

Discussion

In this chapter the results in chapter 8 are discussed.

9.1 Format Alteration Scenarios

The observation that the AST analysis module in detect handled the scenarios of format alteration without fail was expected. Differing format features, such as whitespaces, comments, position of braces etc. are discarded when the source code is parsed. Being a text based tool, Moss is not as robust as detect against such plagiarism techniques. This was seen in particular in the scenarios of repositioning of braces and removal of unnecessary braces. These quick and easy plagiarism techniques confused Moss.

9.2 Identifier Renaming Scenarios

Since identifier strings are not considered in the tree matching algorithm, it was expected that the AST diff would do well in these scenarios. After a quick glance at figure 8.1 it would seem that Moss and detect are equally capable of detecting plagiarisms of this type. However, when one considers the number of cases detected where semantics were changed (see figure 8.2) this implies that Moss would be more prone to detecting “reckless plagiarism” or even false positives in this regard. The difference occurred in scenario 8 where all cases of (incorrect) renaming of variables were detected by Moss. None of the incorrect renamings were detected by detect. This is because, if one renames a variable in such a way that it in fact references a different variable, the resulting parse tree of the procedure will be structurally different.

Unfortunately, in scenario 11 both detect and moss found all cases where function names had been unsystematically replaced. When considering this, one must keep in mind that it is difficult to unsystematically replace function names. This is because, for two functions to be mixed up, the number of parameters and the types of those parameters for those functions must be the same.
9.3 Declaration Reordering Scenarios

As expected, the AST diff could deal with reordering of function bodies (scenario 13). Moss however, was confused by this technique and was only able to match about 50% of the file. Since the order of parameters is significant in the AST, it was not expected that detect would be able to find reordering of parameters (scenario 14). However, the detection of the cases of unsystematic reorderings in scenario 15 suggests that the AST analysis (and Moss as well) really does not correctly detect cases of parameter reordering.

9.4 Statement Reordering Scenarios

It was expected that the AST diff would not do well in these scenarios. Moss detected two out of four functions where variable definitions had been reordered (scenario 16) but also reported the same functions where variable definitions were reordered even though those changes had broken data dependencies among the statements (scenario 17).

9.5 Code Insertion Scenarios

For these plagiarism scenarios the AST diff and Moss performed equally poorly. The exception for detect was the result for scenario 24, which was due to some optimizations done by the parser (see section 8.1). The fact that Moss did well in scenario 25 was most likely due to the fact that Moss is tolerant against noise.

9.6 Control Replacement Scenarios

As expected, the AST analysis module could not find cases of plagiarism where control replacement had been applied. For this type of plagiarisms, Moss was equally ineffective at finding plagiarisms.

9.7 Other Scenarios

The fact that the AST diff could deal with scenario 31 (see section 8.1) was of course expected. Moss did well in scenario 30 where unnecessary parentheses were removed. This was interesting since it performed badly in scenario 3, where unnecessary braces were removed.

9.8 Overall

In chapter 2 we learned that AST based techniques were good for detecting plagiarisms of format alteration and identifier renaming. This was confirmed in our
9.9 Execution Time

experiments by the successful detection of plagiarism scenarios of these kinds. It
appears Moss is also a tool for primarily detecting plagiarisms of format alter-
ation and identifier renaming. The AST diff in detect is however more robust to
plagiarism cases of format alteration and less prone to detecting false positives in
cases of identifier renaming. Our ambition was that detect would be able to deal
with some cases of declaration reordering. The results show that it can effectively
detect plagiarisms where function bodies have been reordered, and that Moss can
not do this as effectively. Finally, we have seen that for C/C++ the AST diff can
detect some odd cases of plagiarism, but this is a side effect of using gcc as a
parser and is not supported for other languages (in our case, just Ada).

Furthermore, the AST diff is not the only part of detect. The PDG diff is
meant to complement the AST diff, filling in the gaps where AST analysis does
not perform well. The results from the testing of the PDG analysis module should
also be considered for the overall evaluation of detect. Those results can be
found in Niklas Holma’s master’s thesis [12].

9.9 Execution Time

The reader should keep in mind that apart from their aptitude in detecting pla-
giarisms of different kinds, there are other important aspects of these systems, for
instance their execution time. The results from the execution time experiment
in section 8.3 indicate that the AST diff in detect runs in polynomial time, of
degree three or possibly higher. Based on what we know of Moss’s algorithm (see
chapter 2), which could possibly be linear, it is conceivable that Moss can process
files much faster than the AST diff in detect. This aspect is of great importance
since these systems are intended to run on large batches of files.
Chapter 10

Conclusions

In this master’s thesis we have presented a method for plagiarism detection in code using abstract syntax tree matching. We have also explained the structure and implementation of a tool which uses this method. The syntax tree matching is one part in the tool’s arsenal for detecting plagiarism.

The tool supports several source languages by using different front ends for parsing the source code and obtaining parse trees. We used gcc and a custom plugin for parsing C and C++. For Ada we constructed a custom parser and symbol table. We also presented some transformations of the trees which were required by the construction of the approximate program dependence graphs.

Our method for finding subtree matches in the syntax trees is based on comparing fingerprints of subtrees. We looked at how subtrees were represented using a cryptographic hash which reflects the types of nodes and the structure of the subtree. The matching algorithm was outlined and we presented the various output formats of the tool.

Finally we constructed a set of scenarios of plagiarisms to evaluate our system. The scenarios ranged from simple techniques, like manipulating the comments of a file, to more invasive actions, like replacing control constructs by equivalents. We ran these scenarios both on our tool’s tree matching algorithm and another system for plagiarism detection in code. From the results we conclude that our method is at least as effective as the other system for finding plagiarisms in the scenarios. The tool is effective for detecting plagiarism of simple techniques such as renaming identifiers and manipulating the format of the code. The syntax tree matching is ineffective for finding plagiarisms of more advanced techniques such as reordering independent statements and inserting benign code.
Chapter 11

Future Work

11.1 Improvements to DETECT

The development of DETECT will not halt with the completion of this master’s thesis project. This is fortunate, since there is much room for improvements.

11.1.1 Extended Language Support

As mentioned in chapter 3, the system becomes more useful as it is extended to support more programming languages. Initially Java and MatLab were mentioned as candidates languages for new front ends. However, it is likely a Python module will be implemented in the near future, as it is a language of growing popularity at the Department of Information and Computer Science and elsewhere.

11.1.2 Serialization of Parsed Code Units

With the current implementation, each code unit is parsed every time it is analyzed by DETECT. To cut down on execution time for COJAC, which uses DETECT repeatedly, AST forests could be serialized and stored once they have been parsed. For a batch of $N$ input units, where we want to compare each unit to all others, each one would then only have to be parsed once instead of $N$ times. In fact, the entire AST forests of two input units do not have to be loaded for comparison. Since the fingerprints of those trees are compared, it would suffice to load those fingerprint sets. Whenever there is need to follow the references to actual trees (e.g. when two equal hashes in the matching algorithm are found) those specific trees can be loaded on demand. Indeed, this is one of the greater benefits of comparing fingerprints of subtrees rather than the subtrees themselves. This is an indispensable improvement. Without it, much time will be wasted on parsing the same input unit over and over.
11.1.3 Handling Commutative Nodes

An improvement to the current matching algorithm in the AST analysis module can be done while hashing the subtrees. In nodes where the order of the children is unimportant, such as nodes of commutative operators, the children could be sorted (by type) before hashing [7, p. 9]. Such an improvement would make the current algorithm stronger against certain reorderings.

11.1.4 Other Forms of Output

As explained in section 6.4.3, the similarity ratio is perhaps not the ideal metric for judging which pairs of code units should be flagged for possible plagiarisms. It may be more interesting to flag pairs of code units which contain subtree matches above a certain weight threshold. One possible additional output from the AST diff would thus be how many subtree matches were above that threshold.

11.1.5 Matching Algorithm for Detecting Near Matches

The matching algorithm finds exact matches of subtrees. By exact we mean with regard to the structure of the tree and to the types of the nodes. While this algorithm is effective for finding different kinds of plagiarisms, it does have its limits. Figure 11.1 and 11.2 show a plagiarism case which the algorithm will not find.

```plaintext
for (int i = 0; i < 100; ++i) {
    for (int j = 0; j < i; ++j) {
        for (int k = 0; k < j; ++k) {
            sum = sum + i*j*k;
        }
    }
}
```

Figure 11.1: Original code.

```plaintext
for (int i = 0; i < 100; ++i) {
    for (int j = 0; j < i; ++j) {
        for (int k = 0; k < j; ++k) {
            sum = sum + 1*j*k;
        }
    }
}
```

Figure 11.2: (Incorrect) plagiarism of 11.1. Notice the “1” instead of “i” on the fourth line.

The figures portray a quite conceivable case of plagiarism. Is it a plagiarism in the strict sense? No, the semantics of the entire code blocks are not the same. However, this could just be a simple case of a student copying another student’s code, and writing a “1” instead of an “i” by mistake. Such near matches are still of interest to find. The trees of the code excerpts above will be equal except for one leaf. A matching algorithm could be developed which permits “noise” differences in the trees up to a certain threshold. If such matching algorithms are implemented, an output of what parts of the trees did not match could be of interest, at least for code units which has an overall high similarity ratio.
11.2 Improvements to COJAC

At the time, COJAC is but a simple script which runs detect. The database mentioned in chapter 4 is merely a directory structure which stores graph images (of ASTs and APDGs) generated during past runs.

11.2.1 Graphical User Interface for Setting up Batch Runs

In order for COJAC to become a system which can easily be used by teachers, a graphical user interface (GUI) must be developed. This was a lower prioritized requirement of the system which was not implemented. The GUI should let the user configure batch runs and specify configuration parameters used in detect. It should also display the results of those runs and keep track of the history of the submitters.

11.2.2 Web Interface for Code Submissions

Ideally COJAC would be part of the back end of a web interface for code submissions. The web interface would let students upload code when demonstrating lab work. Preferably this interface would also display students’ progress and prior submissions. The benefit from this would be the automatic storage of code in a directory structure which COJAC accesses, relieving teachers from the cumbersome task of moving a large collection of submissions around.
Bibliography


Appendix A

Implementation Details

In this section code and implementation details of various parts of DETECT are presented.

A.1 Tree Representation Interface

```cpp
/*
 * Definition of AST_interface
 * Author Erik A. Nilsson <erik.nilsson@liu.se>
 * This class is the common tree representation in DETECT. It serves as an
 * interface, specifying which methods must be implemented by the front ends own
 * trees (e.g. AST_Cpp).
 */

// System headers
#include <map>
#include <set>
#include <sstream>
#include <vector>

// Boost headers
#include "boost/shared_ptr.hpp"

// Custom headers
#include "NodeControlType.h"

class missing_attribute_exception : public std::exception {};  
class missing_child_exception : public std::exception {};  
class cycle_in_graph : public std::exception {};  

class AST_interface {
  public:
    // These are the classes which should be used by the implementing classes
    // to keep track of that node's children and attributes.
    typedef std::map<std::string, boost::shared_ptr<AST_interface>> child_map;
    typedef std::map<std::string, std::string> attribute_map;
    typedef std::set<std::string> var_list;

    // Get the node type. Note that this is front end specific and cannot really
    // be used for anything useful, except for comparing to the type of other
    // nodes of the same type of tree. For that purpose though, the function
    // getHashValue is more appropriate.
    /*
     * @return The node type.
     */
```
virtual std::string getType() const = 0;
/**
 * Set the node type.
 * @param string the new node type.
 */
virtual void setType(const std::string&) = 0;
/**
 * Gives a value based on the node's type. This is used in
 * hashing trees and comparing nodes in AST analysis.
 * @return the type value.
 */
virtual int getHashValue() const = 0;
/**
 * Gets a unique ID (within the tree) for that node.
 * @return the ID.
 */
virtual int getUID() const = 0;
/**
 * Gets the function name of the tree, should be the same for any node
 * in the tree. 
 * @return the function name.
 */
virtual std::string getFunctionName() const = 0;
/**
 * Gets the source file name for this tree, should be the same for any node in
 * the tree.
 * @return the source file name.
 */
virtual std::string getSourceFileName() const = 0;
/**
 * Gets the NodeControlType for the node. The control types are specified in
 * NodeControlType.h. The front ends must set these control types accordingly
 * in order for the APDG factory to be able to discern the control
 * dependencies.
 * @return the control type.
 */
virtual NodeControl::NodeControlType getControlType() const = 0;
/**
 * Gets the child of a node. Gets the child labeled with the parameter name.
 * @throws missing_child_exception if there was no child labeled with the parameter
 * name.
 * @param name the label of the child to get.
 * @return the Child labeled by the parameter name.
 */
virtual boost::shared_ptr<AST_interface> getChild(const std::string& name) const = 0;
/**
 * Gets the begin iterator to the map of children.
 * @return the begin iterator.
 */
virtual child_iterator childBegin() = 0;
/**
 * Gets the end iterator to the map of children.
 * @return the end iterator.
 */
virtual child_iterator childEnd() = 0;
/**
 * Gets an attribute of a node. Gets the attribute labeled by the parameter attribute
 * name. 
 * @throws missing_attribute_exception if there was no such attribute.
 * @param attribute the label of the attribute to get.
 * @return the attribute labeled by the parameter attribute.
 */
virtual std::string getAttribute(const std::string& attribute) const = 0;
/**
 * Gets the begin iterator to the map of attributes.
 * @return the begin iterator.
 */
virtual attribute_iterator attributeBegin() = 0;
/**
 * Gets the end iterator to the map of attributes.
 * @return the end iterator.
 */
virtual attribute_iterator attributeEnd() = 0;
virtual attribute_iterator attributeEnd() = 0;

/**
 * @return the list of the node. The list for the node must be set by the
 * front end in order for the PDG factory to be able to discern the data
 * dependancies. A node which uses a variable/parameter named <VARIABLE>
 * should have the string <UID><VARIABLE> in its use set. We only need to
 * calculate the use set for nodes which have a control type other than NONE.
 */

virtual var_list& getUseList() = 0;

/**
 * @return the list of the node. The list for the node must be set by the
 * front end in order for the PDG factory to be able to discern the data
 * dependancies. A node which defines a variable/parameter named <VARIABLE>
 * should have the string <UID><VARIABLE> in its def set. We only need to
 * calculate the def set for nodes which have a control type other than NONE.
 */

virtual var_list& getDefinitionList() = 0;

virtual void fixForPDG() = 0;

virtual boost::shared_ptr<AST_interface> getLoopBefore() = 0;

virtual boost::shared_ptr<AST_interface> getLoopAfter() = 0;

virtual boost::shared_ptr<AST_interface> getLoopBody() = 0;

virtual boost::shared_ptr<AST_interface> getLoopPreCond() = 0;

virtual boost::shared_ptr<AST_interface> getLoopPostCond() = 0;

virtual std::string getGotoLabel() = 0;

virtual std::string getLabel() = 0;
virtual std::string getLabelName() = 0;

virtual boost::shared_ptr<AST_interface> getStmtNr(int) = 0;

virtual int getNoOfStmts() = 0;

virtual unsigned numChildren() const = 0;

bool equals(const AST_interface& other);

friend void dump_as_dot(AST_interface& tree, std::ostream& ostr, bool detnod, bool highlight_source_locations);

friend void assertTree(AST_interface& tree);

friend int getWeight(AST_interface& tree);

virtual void addChild(const std::string&, boost::shared_ptr<AST_interface>) = 0;

virtual void printTree(int depth);

virtual int maxUID();

friend void assertTree(AST_interface& tree);

virtual void setWeight(int);

virtual void calculateWeight();
virtual void removeChild(const std::string&);  // Removes a child of the node.

virtual void renameChild(const std::string&, std::string newname);  // Renames a child node.

virtual void addAttribute(const std::string&, const std::string&);  // Adds an attribute to the node.

void insertStmtAt(boost::shared_ptr<AST_interface> node, int index);  // Insert a child node to a node with the control type STATEMENT_LIST.

void removeStmtAt(int index);  // Remove a child node of a particular index for a node with the control type STATEMENT_LIST.

private:

// Throws exception cycle_in_graph if a cycle is detected
void assertTree(std::set<int>&, std::string);  // Ensures the tree is acyclic.

void assertTree(AST_interface&);  // Asserts the tree is acyclic.

int getWeight(AST_interface&);  // Gets the weight of the tree.

std::string get_name_if_node_has_one(AST_interface&);  // Returns the attribute labeled "name". If the node does not have such a node, the string "artificial" is returned instead.

static int node_count;  // Stores the node count.

protected:

// For keeping track of the maximum UID in the tree.

void dump_as_dot(AST_interface& tree, std::ostream& ostr, bool detnod, bool highlight_source_locations);  // Prints the tree in a dot format.

void asserTree(AST_interface&);  // Asserts the tree is acyclic.

int getWeight(AST_interface&);  // Gets the weight of the tree.

std::string get_name_if_node_has_one(AST_interface&);  // Returns the attribute labeled "name". If the node does not have such a node, the string "artificial" is returned instead.

std::string nodeToString(AST_interface&);  // Returns a string representation of the node.

std::string nodeToString(AST_interface&);  // Returns a string in the format \%&UID<\%NODENAME>.

std::vector<int> getLinesAsList(AST_interface& tree);  // Finds all line numbers which appear in the parameter tree. These numbers are stored in a list and sorted ascendingly.

std::vector<int> getLinesAsList(AST_interface& tree);  // Gets the line number of a node if it has one, otherwise return 0. Line numbers are either found in nodes with the "line" attribute, or the "srcp" attribute. However, "srcp" attributes also have filename information. If the filename in the srcp matches the source file name of the parameter tree,
then we count the line number from the srcp attribute. Otherwise we ignore it.

@param tree the node to get the line number for

@return the line number.

Listing A.1: AST_interface.h

Notice that there is no class for the whole tree. A tree is simply a reference to an AST_interface node, the root of the tree.

As can be seen in line 26 (among others), objects of the AST_interface class are often referenced through smart pointers from the boost C++ library smart_ptr [6]. This is mainly to avoid memory leaks when the tree is constructed and when transformations are performed. The Type Value is given by the member function getHashValue() (line 45).

A.2 GCC Plugin

```cpp
#include <stdlib.h>
#include <string>
#include <iostream>
#include <my-tree-dump.h> // added by ernil
#include "int1.h"
#include "tm.h"
#include "cp/cp-tree.h"
#include "c-family/c-common.h"
#include "diagnostic.h"
#include "langhooks.h" // added by ernil
#include "c-family/c-pretty-print.h" // added by ernil
}
#include <set>
#include <iostream>
#include <iostream>
#include "my-tree-dump.h" // added by ernil
using namespace std;

int plugin_is_GPL_compatible;

// This plugin was written with help from the following blog:
// http://www.codesynthesis.com/~boris/blog/2010/05/03/parsing-cxx-with-gcc-plugin-part-1/

// The plugin basically does the same thing as the gcc flag -fdump-tree-original-raw
// albeit with a few modifications:
// – Clearly prints the name of the current file and function
// as well as an end-function marker, this makes scanning of
// the dump file easier, and the information about file and function
// name is interesting later on in the analysis
// – Only dumps the function if it is not from a system header file.
```
void my_traverse (tree fndecl)
{
  expanded_location xloc = expand_location (DECL_SOURCE_LOCATION (fndecl));
  if (!xloc.sysp) { // is not a system header
    cout << "FUNCTION: " << lang_hooks.decl_printable_name (fndecl, 2)
         << endl;
    cout << "FILE : " << xloc.file << endl;
  }
  if (!xloc.sysp) { // is not a system header
    int flags = (get_dump_file_info (4) - flags);
    my_dump_node (DECL_SAVED_TREE (fndecl), TDF_SLIM | flags, stdout);
    cout << "END_OF_FUNCTION" << endl;
  }
}

extern "C" void
gate_callback (void *gcc_data, void *user_data)
{
  // If there were errors during compilation,
  // let GCC handle the exit.
  // set GCC to handle the exit.
  if (errorcount | sorrycount)
    cerr << "Found errors while compiling, plugin will not run." << endl;
  return;
}

extern "C" int
plugin_init (plugin_name_args *info, plugin_gcc_version *ver)
{
  cerr << "starting" << info->base_name << endl;
  cerr << "processing" << main_input_filename << endl;
  asm_file_name = HOST_BIT_BUCKET;
  // Unfortunately PLUGIN_PRE_GENERICIZE just works for c/c++
  register_callback (info->base_name,
                    PLUGIN_PRE_GENERICIZE,
                    &gate_callback,
                    0);
  return 0;
}

Listing A.2: cpplugin.cc

The comments on lines 50 to 80 describe changes to the header file my-tree-dump.h, which is an alternate version of gcc's tree-dump.h. We have not included my-tree-dump.h here since it is very large and only has those slight modifications compared to the original. It can however be found at http://www.ida.liu.se/~ernil/detect/code/.
### Appendix B  List of Requirements

<table>
<thead>
<tr>
<th>Id</th>
<th>Changed</th>
<th>Type</th>
<th>Description</th>
<th>Prio</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2012-01-27</td>
<td>Documents</td>
<td>Documentation on how to use COJAC.</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2012-01-27</td>
<td>System</td>
<td>COJAC can be run in a POSIX system.</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>2012-01-26</td>
<td>System</td>
<td>COJAC can define a group of input units in a configuration file and detect similarities within this group.</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>2012-01-26</td>
<td>System</td>
<td>COJAC can define in which ways the comparisons should be done in the configuration file.</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>2012-01-26</td>
<td>System</td>
<td>There is a GUI for modifying the configuration file for COJAC.</td>
<td>2</td>
</tr>
<tr>
<td>6</td>
<td>2012-01-26</td>
<td>System</td>
<td>There is a GUI for COJAC.</td>
<td>3</td>
</tr>
<tr>
<td>7</td>
<td>2012-01-26</td>
<td>System</td>
<td>COJAC uses a program DETECT, which uses one or several comparison algorithms to compare two units of source code at a time.</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>2012-01-26</td>
<td>System</td>
<td>DETECT is a command-line based program.</td>
<td>1</td>
</tr>
<tr>
<td>9</td>
<td>2012-01-26</td>
<td>System</td>
<td>DETECT takes two input units of code (of the same programming language) and compares them.</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>2012-01-27</td>
<td>System</td>
<td>Parameters passed to DETECT can be given on the command-line.</td>
<td>2</td>
</tr>
<tr>
<td>11</td>
<td>2012-01-27</td>
<td>System</td>
<td>Parameters passed to DETECT can be extracted from a configuration file given by COJAC.</td>
<td>1</td>
</tr>
<tr>
<td>12</td>
<td>2012-01-26</td>
<td>System</td>
<td>DETECT supports input units written in the language C.</td>
<td>1</td>
</tr>
<tr>
<td>13</td>
<td>2012-01-26</td>
<td>System</td>
<td>DETECT supports input units written in the language C++.</td>
<td>1</td>
</tr>
<tr>
<td>Id</td>
<td>Changed</td>
<td>Type</td>
<td>Description</td>
<td>Prio</td>
</tr>
<tr>
<td>----</td>
<td>-----------</td>
<td>----------</td>
<td>-----------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>14</td>
<td>2012-01-26</td>
<td>System</td>
<td>DETECT supports input units written in the language Ada.</td>
<td>1</td>
</tr>
<tr>
<td>15</td>
<td>2012-01-26</td>
<td>System</td>
<td>DETECT supports input units written in the language Java.</td>
<td>2</td>
</tr>
<tr>
<td>16</td>
<td>2012-01-26</td>
<td>System</td>
<td>DETECT supports input units written in the language MatLab.</td>
<td>3</td>
</tr>
<tr>
<td>17</td>
<td>2012-01-26</td>
<td>System</td>
<td>DETECT integrates different front-ends for the supported languages.</td>
<td>1</td>
</tr>
<tr>
<td>18</td>
<td>2012-01-26</td>
<td>System</td>
<td>It should be possible to integrate a new front-end with DETECT so that a new programming language can be supported.</td>
<td>1</td>
</tr>
<tr>
<td>19</td>
<td>2012-01-27</td>
<td>System</td>
<td>It should be possible to remove a front-end in DETECT.</td>
<td>1</td>
</tr>
<tr>
<td>20</td>
<td>2012-01-26</td>
<td>System</td>
<td>It should be possible to replace a front-end in DETECT so that a new standard of a language can be supported.</td>
<td>1</td>
</tr>
<tr>
<td>21</td>
<td>2012-01-27</td>
<td>System</td>
<td>It should be possible to integrate a new module with comparing algorithm(s) in DETECT</td>
<td>1</td>
</tr>
<tr>
<td>22</td>
<td>2012-02-02</td>
<td>Documents</td>
<td>Documentation exists on how to manage (adding and removing) the front-ends within DETECT.</td>
<td>1</td>
</tr>
<tr>
<td>23</td>
<td>2012-01-27</td>
<td>System</td>
<td>DETECT determines how many lines in the input files are textually identical.</td>
<td>1</td>
</tr>
<tr>
<td>24</td>
<td>2012-01-26</td>
<td>System</td>
<td>DETECT determines how many lines in the input files are identical without regarding whitespaces.</td>
<td>1</td>
</tr>
<tr>
<td>25</td>
<td>2012-01-31</td>
<td>System</td>
<td>DETECT reports which lines were textually identical.</td>
<td>1</td>
</tr>
<tr>
<td>26</td>
<td>2012-01-31</td>
<td>System</td>
<td>DETECT reports which lines were textually identical not regarding whitespaces.</td>
<td>1</td>
</tr>
<tr>
<td>27</td>
<td>2012-02-02</td>
<td>System</td>
<td>DETECT reports a metric representing the amount of similarity in code where Format Alteration, Identifier Renaming and Declaration Reordering has been applied.</td>
<td>1</td>
</tr>
<tr>
<td>28</td>
<td>2012-02-02</td>
<td>System</td>
<td>DETECT reports a metric that represents the amount of similarity in code where Statement Reordering, Code Insertion and Control Replacement has been applied and also fulfills requirement 27.</td>
<td>2</td>
</tr>
<tr>
<td>Id</td>
<td>Changed</td>
<td>Type</td>
<td>Description</td>
<td>Prio</td>
</tr>
<tr>
<td>----</td>
<td>-----------</td>
<td>--------</td>
<td>-----------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>29</td>
<td>2012-01-26</td>
<td>System</td>
<td>DETECT finds similarities in code which has been copied and modified by micro optimization.</td>
<td>3</td>
</tr>
<tr>
<td>30</td>
<td>2012-02-22</td>
<td>System</td>
<td>COJAC records data regarding submitters history of submissions and plagiarism metrics.</td>
<td>3</td>
</tr>
<tr>
<td>31</td>
<td>2012-02-22</td>
<td>System</td>
<td>COJAC flags submitters whose submissions have frequently high plagiarism metrics.</td>
<td>3</td>
</tr>
</tbody>
</table>
Appendix C  Default Configuration File

AST_ALL_MATCHES_OUTPUT=TRUE
AST_DETACHED_OUTPUT=FALSE
AST_DOT_GRAPH_DETAIL=FALSE
AST_DUMP_DOT_GRAPHS=FALSE
AST_DUMP_FINGERPRINTS=FALSE
AST_DUMP_GENERICIZED=FALSE
AST_MATRIX_OUTPUT=FALSE
AST_GREATEST_OUTPUT=TRUE
AST_WEIGHT_THRESHOLD=0
COLOR_RED_THRESHOLD=98
COLOR_YELLOW_THRESHOLD=90
CRYPT_HASH_METHOD=MD5_METHOD
ENABLE_AST_DIFF=TRUE
ENABLE_COLORFUL_OUTPUT=TRUE
ENABLE_CONFIG_DUMP=FALSE
ENABLE_DEBUG_MODE=FALSE
ENABLE_PDG_DIFF=TRUE
ENABLE_TEXT_DIFF=TRUE
PDG_DOT_GRAPH_DETAIL=TRUE
PDG_DOT_SHOW_CDEP=TRUE
PDG_DOT_SHOW_CFLOW=TRUE
PDG_DOT_SHOW_DDEP=TRUE
PDG_DUMP_DOT_GRAPHS=FALSE
PDG_DUMP_MAPPINGS=TRUE
PDG_DUMP_TEXT_GRAPHS=FALSE
PDG_GRAPH_MATCH_TYPE=SUB-MONO
PDG_MATRIX_OUTPUT=TRUE
PDG_SHOW_LEGEND=TRUE
PDG_VERBOSE_OUTPUT=TRUE
TEXT_DIFF_DETAILED_OUTPUT=FALSE
TEXT_DIFF_PREPROCESS=TRUE
TEXT_DIFF_SECTION_DECREMENT=5
TEXT_DIFF_SECTION_MIN=1
TEXT_DIFF_WARNING_THRESHOLD=95
Appendix D  Configuration File for Benchmarking

# Main parts of detect
ENABLE_DEBUG_MODE=FALSE
ENABLE_CONFIG_DUMP=FALSE
ENABLE_TEXT_DIFF=FALSE
ENABLE_AST_DIFF=TRUE
ENABLE_PDG_DIFF=FALSE

# Graph output
AST_DUMP_DOT_GRAPHS=FALSE
AST_DOT_GRAPH_DETAIL=FALSE
AST_DUMP_GENERICIZED=FALSE
PDG_DUMP_DOT_GRAPHS=FALSE
PDG_DUMP_TEXT_GRAPHS=FALSE
PDG_DOT_GRAPH_DETAIL=FALSE
PDG_DOT_SHOW_CFLOW=FALSE
PDG_DOT_SHOW_DDEP=FALSE
PDG_DOT_SHOW_CDEP=FALSE

# Flags for AST Analysis
CRYPT_HASH_METHOD=SHA1_METHOD
AST_WEIGHT_THRESHOLD=5
AST_DUMP_FINGERPRINTS=FALSE
AST_MATRIX_OUTPUT=FALSE
AST_GREATEST_OUTPUT=TRUE
ENABLE_COLORFUL_OUTPUT=TRUE
AST_DETAILED_OUTPUT=FALSE
AST_ALL_MATCHES_OUTPUT=TRUE

# Flags for PDG Analysis
## SUB-MONO | SUB-ISO | EXACT-ISO
PDG_GRAPH_MATCH_TYPE=SUB-MONO
PDG_DUMP_MAPPINGS=TRUE
PDG_VERBOSE_OUTPUT=TRUE
**Upphovsrätt**

Detta dokument hålls tillgängligt på Internet — eller dess framtida ersättare — under 25 år från publiceringsdatum under förutsättning att inga extraordinära omständigheter uppstår.

Tillgång till dokumentet innebär tillstånd för var och en att läsa, ladda ner, skriva ut enstaka kopior för enskilt bruk och att använda det oförändrat för icke-kommersiell forskning och för undervisning. Överföring av upphovsrätten vid en senare tidpunkt kan inte upphäva detta tillstånd. All annan användning av dokumentet kräver upphovsmannens medgivande. För att garantera äktheten, säkerheten och tillgängligheten finns det lösningar av teknisk och administrativ art.

Upphovsmannens ideella rätt innefattar rätt att bli nämnd som upphovsman i den omfattning som god sed kräver vid användning av dokumentet på ovan beskrivna sätt samt skydd mot att dokumentet ändras eller presenteras i sådan form eller i sådant sammanhang som är kränkande för upphovsmannens litterära eller konstnärliga anseende eller egenart.

För ytterligare information om Linköping University Electronic Press se förlagets hemsida [http://www.ep.liu.se/](http://www.ep.liu.se/)

---

**Copyright**

The publishers will keep this document online on the Internet — or its possible replacement — for a period of 25 years from the date of publication barring exceptional circumstances.

The online availability of the document implies a permanent permission for anyone to read, to download, to print out single copies for his/her own use and to use it unchanged for any non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional on the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility.

According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement.

For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its www home page: [http://www.ep.liu.se/](http://www.ep.liu.se/)

© Erik A. Nilsson