

Institutionen för datavetenskap
Department of Computer and Information Science

Final thesis

**Program Dependence Graph Generation and
Analysis for Source Code Plagiarism Detection**

by

Niklas Holma

LIU-IDA/LITH-EX-A--12/065-SE

2012-12-19



Linköpings universitet

Final thesis

Program Dependence Graph Generation and Analysis for Source Code Plagiarism Detection

by


Niklas Holma

LIU-IDA/LITH-EX-A--12/065-SE

2012-12-19

Supervisor: Jonas Wallgren

Examiner: Christoph Kessler

	Avdelning, Institution Division, Department Division of Software and Systems Department of Computer and Information Science Linköpings universitet SE-581 83 Linköping, Sweden		Datum Date 2012-12-19
	Språk Language <input type="checkbox"/> Svenska/Swedish <input type="checkbox"/> Engelska/English <input type="checkbox"/> _____	Rapporttyp Report category <input type="checkbox"/> Licentiatavhandling <input checked="" type="checkbox"/> Examensarbete <input type="checkbox"/> C-uppsats <input type="checkbox"/> D-uppsats <input type="checkbox"/> Övrig rapport <input type="checkbox"/> _____	ISBN _____ ISRN LIU-IDA/LITH-EX-A--12/065-SE Serietitel och serienummer ISSN Title of series, numbering _____
URL för elektronisk version http://www.ida.liu.se http://www.ep.liu.se			
Titel Title	Generering och analys av programberoendegrafer för detektering av plagiat i källkod Program Dependence Graph Generation and Analysis for Source Code Plagiarism Detection		
Författare Author	Niklas Holma		
Sammanfattning Abstract <p>Systems and tools that finds similarities among essays and reports are widely used by todays universities and schools to detect plagiarism. Such tools are however insufficient when used for source code comparisons since they are fragile to the most simplest forms of diguises. Other methods that analyses intermediate forms such as token strings, syntax trees and graph representations have shown to be more effective than using simple textual matching methods. In this master thesis report we discuss how program dependence graphs, an abstract representation of a programs semantics, can be used to find similar procedures. We also present an implementation of a system that constructs approximated program dependence graphs from the abstract syntax tree representation of a program. Matching procedures are found by testing graph pairs for either sub-graph isomorphism or graph monomorphism depending on whether structured transfer of control has been used. Under a scenario based evaluation our system is compared to Moss, a popular plagiarism detection tool. The result shows that our system is more or least as effective than Moss in finding plagiarized procedured independently on the type of modifications used.</p>			
Nyckelord Keywords plagiarism, program dependence graph			

Abstract

Systems and tools that find similarities among essays and reports are widely used by today's universities and schools to detect plagiarism. Such tools are however insufficient when used for source code comparisons since they are fragile to the most simplest forms of disguises. Other methods that analyse intermediate forms such as token strings, syntax trees and graph representations have shown to be more effective than using simple textual matching methods. In this master thesis report we discuss how program dependence graphs, an abstract representation of a program's semantics, can be used to find similar procedures. We also present an implementation of a system that constructs approximated program dependence graphs from the abstract syntax tree representation of a program. Matching procedures are found by testing graph pairs for either sub-graph isomorphism or graph monomorphism depending on whether structured transfer of control has been used. Under a scenario based evaluation our system is compared to Moss, a popular plagiarism detection tool. The result shows that our system is more or least as effective than Moss in finding plagiarized procedures independently on the type of modifications used.

Sammanfattning

System och verktyg som hittar likheter mellan uppsatser och rapporter används i stor omfattning av dagens universitet och skolor för att hitta plagiat bland studenters inlämningar. Sådana verktyg är dock otillräckliga när de används för att jämföra programkod eftersom de är svaga mot de enklaste formerna av modifierationer. Andra metoder som analyserar mellanstegsformer såsom tokensträngar, syntaxträd och grafrepresentationer har visat sig vara mer effektiva än att använda sig av enkla textuella metoder. I denna examensuppsats diskuterar vi hur programberoendegrafer, en abstrakt representation av en programs semantik, kan användas för att hitta jämförelsevis liknande procedurer. Vi presenterar också ett system som konstruerar approximerade programberoendegrafer från det abstrakta syntaxträdet av ett program. Matchande procedurer hittas genom att testa grafpar för antingen sub-graf isomorfism eller monomorfism beroende på om strukturerad byte av kontrollflöde har använts. I en scenariobaserad utvärdering jämför vi vårt system mot Moss, ett populärt verktyg för att detektera plagiat. Resultaten visar att vårt system är lika eller mer effektivt som Moss att detektera plagierade procedurer oberoende av de typer av modifierationer som använts.

Acknowledgments

I would like to thank my supervisor, Jonas Wallgren for having great patience in proofreading my report and giving ideas on how to present my work in the best way. I would like to thank Erik Nilsson for his commitment to our goals and the amount of work he put into the abstract syntax tree transformation. I would also like to thank Torbjörn Jonsson and Klas Arvidsson for helping us start this project and for all the ideas and positive spirit they gave us.

Contents

1	Introduction	1
1.1	Background	1
1.2	Clients	1
1.3	Plagiarization	1
1.4	Plagiarism by means of code clones	2
1.5	Looking through code plagiarism	2
1.6	Plagiarism detection system	3
1.7	Acronyms	3
2	Related Work	5
2.1	Plagiarism disguises	5
2.2	Plagiarism detection techniques and tools	7
2.2.1	Textual comparison based	7
2.2.2	Metrics based	7
2.2.3	Token comparison	7
2.2.4	Syntactical analysis	7
2.2.5	Program Dependence Graph analysis	8
3	Program Dependence Graph	9
3.1	Terminology	9
3.1.1	Control-flow Graph	10
3.1.2	Control Dependence	11
3.1.3	Data Dependence	13
3.1.4	Reaching Definitions	14
3.1.5	Program Dependence Graph	17
3.2	Plagiarized Program Dependence Graphs	18
3.2.1	Graph Morphisms	18
3.3	Algorithms for subgraph isomorphism testing	19
3.3.1	Time Complexities	20
3.4	Construction of Approximated Program Dependence Graphs	21
4	Implementation	23
4.1	Requirement summary	23
4.2	System Overview	24
4.2.1	Modules	26

4.3	External libraries and dependencies	26
4.3.1	VFLib	26
4.3.2	Boost	26
4.4	Configuration of modules	26
4.5	TextDiff Module	27
4.5.1	LCSSet	27
4.6	Abstract syntax tree module	31
4.7	APDG Generator Module	31
4.7.1	Data structures	31
4.7.2	AST Interface	32
4.7.3	ACDS Generation	33
4.7.4	ADDS Generation	38
4.7.5	APDG Examples	38
4.8	APDG Analysis Module	46
4.8.1	Pruning the search space	46
4.8.2	Optimal Configuration	47
4.8.3	Threading	47
4.8.4	Output	48
4.8.5	Sub-graph isomorphism example	50
5	Analysis	53
5.1	Method of analysis	53
5.1.1	Scenarios	54
5.2	Interpreting Results	56
5.3	Test Results	56
5.3.1	All results	56
5.3.2	Similarity ratio by type of modification	56
5.3.3	Detected procedures by type of modification	56
6	Discussion	63
6.1	Format Alteration Scenarios	63
6.2	Identifier Renaming Scenarios	63
6.3	Declaration Reordering Scenarios	63
6.4	Statement Reordering Scenarios	64
6.5	Code Insertion Scenarios	64
6.6	Control Replacement Scenarios	65
6.7	Other Modification Scenarios	65
6.8	Overall	65
7	Conclusions	67
8	Future Work	69
8.1	APDGs for Ada	69
8.2	Serialization of APDGs	69
8.3	Improving the preciseness of APDGs	70
8.3.1	Pointers and aliases	70
8.3.2	Exception analysis	70

Contents	xi
8.3.3 Call graphs	70
8.4 Other types of APDG analysis	71
A Requirements of DETECT	75
B Interface for PDGFactory	77
C Configuration file for benchmarking	84
D Default configuration file	86

Chapter 1

Introduction

In this master thesis project we will examine the possibility of analysing program dependence graphs to detect software plagiarism. We also present an implementation of a system that generates and analyses approximate program dependence graphs for plagiarism detection. At last we compare the results to conventional methods of plagiarism detection by performing a quantitative analysis with our system and another popular detection tool.

1.1 Background

In 2011, the teaching group at the Department of Computer and Information Science (IDA) at Linköping University discussed the idea of constructing a system aiding in the detection of software plagiarism. This discussion was later solidified into a master thesis project to be performed by Niklas Holma (author) and Erik Nilsson, both working as teaching assistants at IDA at that point of time.

1.2 Clients

The clients of this project are Torbjörn Jonsson and Klas Arvidsson representing the teaching group (UPP) at the department of Computer and Information Science at Linköping University.

1.3 Plagiarization

In programming courses, there are always students who hand in code that does not belong to them. Looking at someone else's code can be a good starting point in learning how to program, but directly copying someone's code and handing in as one's own work is an act of plagiarism. Using code from the web is a popular choice, but there are also cases when a fellow student has been victimized.

Detecting software plagiarism is not a simple task. First and foremost is the job of finding the actual similarity between submissions. Code can easily be disguised,

and plagiarised code is easily overlooked in large courses. Second, if course-staff would find identical submissions, proof of plagiarisation occurrence would have to be found as well.

During literature research we found several commercial and noncommercial systems for aiding in the detection of software plagiarisms. One alternative would be to use one of these existing systems, but constructing a tool that is tailored for the courses taught at IDA would give better support in finding plagiarised code. It would give the teachers the possibility to adapt the system to the specific programming assignments and languages used.

1.4 Plagiarism by means of code clones

The act of finding plagiarism consists of finding similar code, and has its roots in finding code clones. We define a *code clone* to be a piece of code that is identical to another piece of code, and a *near miss clone* as a piece of code that is nearly identical to another piece of code.

There are many reasons why code clones occur in software projects. Baxter et al. [4] mentions several,

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

Plagiarized code can also be seen as code clones or near miss clones that have separate authors. Plagiarism detection is not much different from regular code clone detection, the only difference is the reason behind the occurrence of such clones. In academia, plagiarized code arises from a programmer's unwillingness to learn or solve a problem and many cheaters will put in efforts to hide their actions.

When clones or near miss code clones have been found, problematically there exist no method or standard that constitutes what is to be regarded as plagiarism and not. This emphasizes that software plagiarism can not be *asserted* by a system, the use of plagiarism detection tools is merely an aid in finding code clones. The actual assertion of code plagiarism must ultimately be made by a teacher or a judge.

1.5 Looking through code plagiarism

Finding plagiarized code is not an easy task, a programmer can easily overhaul code in such a way that a textual comparison would yield a zero percent match. To look

through such disguises, plagiarism detection tools need therefore incorporate more effective ways of analysing the code, such as intermediate code analysis. Interpreting and comparing intermediate forms of program code makes it possible to analyse the syntax and the meaning of the code rather than just the text. This is necessary to capture the core functionality of the program as well as the intent of the programmer.

Many popular code plagiarism detection tools that exist today analyse the intermediate code forms that modern popular compilers work with, such as tokens or syntax trees. Other more advanced tools generate and analyse their own abstract representations of the code, such as program dependence graphs, which is an abstract representation of a program's semantics.

1.6 Plagiarism detection system

The result of the project proposed at IDA includes the implementation of a system finding matches and correlations between program code which we call DETECT. DETECT is a subsystem of COJAC, the entire system that manages student hand-ins and that provides a user interface. The details of the entire system are given in chapter 4.

Our literature research shows many different more or less successful techniques that can be applied to find plagiarism in source code. We have chosen to create a hybrid system incorporating several approaches. DETECT employs textual matching, abstract syntax tree analysis and program dependence graph analysis to compare and find correlations between two units of source code. It finds plagiarized code that is written in C, C++ and Ada.

We divided the work into three parts: textual matching, the generation and analysis of abstract syntax trees and the generation and analysis of program dependence graphs. The exposition of the textual matching and program dependence graph analysis is given in this report, while Nilsson [17] presents the work of abstract syntax tree matching.

1.7 Acronyms

Table 1.1 explains common acronyms used in this report.

Name	Description
APDG	Approximate Program Dependence Graph
ACDS	Approximate Control Dependence Subgraph
ADDS	Approximate Data Dependence Subgraph
AST	Abstract Syntax Tree
VF2	A graph matching algorithm used for finding graph isomorphisms and monomorphisms

Table 1.1. Acronyms

Chapter 2

Related Work

2.1 Plagiarism disguises

Much work has been done in the area of program plagiarization detection. Although there actually exists no standard on how to classify source code plagiarism, the result of previous work can help to analyse the effectiveness of existing and future detection tools. One especially important part of work in plagiarism detection is in the classification and categorization of plagiarization techniques. Liu et al. [16] characterize 5 main plagiarism disguises, and rank these in order from trivial to more complex obfuscations. Among other research, we have also found other valid forms of plagiarism disguises. Here follows a summary.

Format Alteration (FA)

The plagiarist systematically reformats the code by adding or removing newlines, whitespaces and comments. Since the student need no understanding of the original code or the programming language used to perform such alterations, this is deemed to be the most simplest form of alteration.

Identifier Renaming (IR)

The plagiarist systematically renames all or some of the identifiers of the code. Renaming all the variables in a program is an easy task and can even trick the most experienced programming teacher or assistant.

Statement Reordering (SR)

The plagiarist systematically reorder statements in the program, while still preserving the semantics of the code.

Declaration Reordering (DR)

The plagiarist systematically reorders statements in the program that declare new variables or introduces new identifiers, while still preserving the semantics of the code. This is included in Liu et al. [16] definition of statement reordering. We have chosen to distinguish between these two.

Control Replacement (CR)

The plagiarist replaces language control-constructs with equivalent constructs, such as replacing a `for` loop statement with an equivalent `while` loop (C++). Another more intricate replacement would be to switch a `while`-statement for a `do-while` loop. Figure 2.1 shows an example of the first mentioned type.

```

1 /* Original */
2 int k = 0;
3 for(int i = 0; i < 10; ++i)
4 {
5     k += i;
6 }
1 /* Plagiarism */
2 int k = 0;
3 int i = 0;
4 while(i < 10)
5 {
6     k += i;
7     ++i;
8 }

```

Figure 2.1. Example of C++ code where a `for` loop has been replaced by a `while` loop.

Code Insertion (CI)

The plagiarist adds code that does not change the functionality of the program, such as *dead code*. Dead code is a computation that calculates a result which is not used in later parts of the procedure [8], such as a `for` loop with an empty body, or assigning a variable that never will be used.

Other Modifications (OM)

All other forms of code disguises fall in this category. For instance making small optimizations to the code such as reusing or removing variables, eliminating dead code or inlining functions. The keyword *small* is important since optimizing code demands knowledge of how the code works and can be seen as unique work. For instance, we do not classify loop unrolling as a form of disguise since the programmer needs to understand all the conditions and invariants involved to perform such a task.

2.2 Plagiarism detection techniques and tools

Techniques based on the detection of plagiarism has its roots in the detection of code clone detection and near miss clone detection. This section summarizes the different types of techniques that exists and some examples of systems employing them.

2.2.1 Textual comparison based

Textual comparison based techniques find similarities between sections of text and code. These kind of tools are often language independent since they have little or no regard to the syntax of the language used. Such systems usually implement some of the popular string matching algorithms such as Longest Common Substring matching, Levenshtein distance or Greedy String Tiling [22]. Schleimer [20] introduces MOSS, an example of a popular plagiarization detection service that considers the text of a program to find similar code. MOSS hashes and compares string k-grams to find partial copies of documents, where a k-gram is a contiguous substring of length k.

2.2.2 Metrics based

A metric based plagiarization detection tool collects data about the programs to analyse, such as number of loops and declarations, and creates metric vectors from these. The metric vectors are then interpreted as points in a cartesian coordinate system, and near points are considered to respond to similar code.

2.2.3 Token comparison

A token is a unit that represents a string recognized by the language that the code is written in. Tokenising code is the process of retrieving tokens from code by means of running it through a lexical analyser [1].

JPlag[18] is an example of a system using tokens to find plagiarisms among code, it analyses strings of tokens by using the Greedy String Tiling algorithm.

2.2.4 Syntactical analysis

When analysing the syntax of the language, the grammar of the programming language has to be taken into account. This can be done by parsing the program code and transforming it into some intermediate representation. The most widely used intermediate form produced by compiler-front ends is the abstract syntax tree, or AST for short. An AST is a tree data structure which represents the hierarchical syntactic structure of the source program. The nodes of the abstract syntax tree are operators or language constructs, and the children are the components of that construct.

Chilowicz et al. [6] introduce a method for plagiarism detection comparing abstract syntax trees by means of sub-tree fingerprinting. Baxter et al. [4] show

how abstract syntax trees can be used to find code clones and near miss clones by using an artificially bad hash function to find subtrees.

Nilsson [17] presents how syntax trees can be used to find plagiarized code, and further presents how DETECT implements AST analysis.

2.2.5 Program Dependence Graph analysis

Previously, data and control dependence graphs have been used by compilers e.g for instruction scheduling and dead code elimination. They can be seen as an abstract representation of the control and data dependencies within a program. Ferrante et al. [9] introduce the program dependence graph, or PDG for short, a multigraph that unifies these two types of dependencies.

The PDG has many areas of use in software engineering. It has mainly been used for program slicing, but it has also been shown to be a valid representation in finding code duplications and plagiarizations. Komondoor and Horwitz [14] first introduced the concept of finding code clones by checking for isomorphic PDGs with program slicing. Later on Krinke [15] presented an approach in detecting code clones by comparing length limited paths in the program dependence graph.

In focus, the most related work in the field of plagiarization detection and to our approach has been conducted by Liu et al. [16]. Here they present the tool GPLAG that finds plagiarism among PDGs using relaxed subgraph isomorphism testing. Program dependence graphs have the remarkable property that they are invariant during nearly all forms of plagiarism. This makes PDG analysis a very robust technique against almost all forms of code disguises, even the most complex types such as control replacement and code insertion.

Chapter 3

Program Dependence Graph

In this chapter, we will give the definition of program dependence graphs, as well as the type of PDG used in our application. We also present how to analyse PDGs to find code plagiarization.

3.1 Terminology

The PDG is a labelled directed multigraph representing the unification of control and data dependencies *within* a program, hence it can be seen as an abstract representation of a program's *semantics*.

The vertices of the PDG represents program statements and predicate expressions. The directed edges represent the control and data dependencies given by the control and data flow of the program.

To clarify what we actually mean with a program dependence graph, we will here explain the terminology used. The initial definition of a program dependence graph was given by Ferrante et al. [9], and the definition expressed the control dependence by the means of post-dominance nodes in a program's *control-flow* graph.

3.1.1 Control-flow Graph

The control-flow graph of a program is a directed graph, where the vertices represent the statements of the program, and the edges represent the transfer of control between these. Our definition differs slightly from the classic definition where the nodes in a control-flow graph represent basic blocks. In our control-flow graph, the nodes represent each separate statement in the program.

We divide the nodes of a control-flow graph into two categories, *predicate* nodes and *regular statement* nodes. A predicate node represent a boolean expression that can be found in constructs that alter the flow of control. In C++ they would represent the boolean expression inside an `if`, `while`, `do while` or `for` statement. In Ada they can, apart from those already mentioned and supported, represent the boolean expression inside a `loop` or `exit when` statement. Since the C++ `switch` statement and Ada's equivalent `case` statement can be seen as a series of `if`, `else if/elsif` and `else` statements, it is not necessary to treat them separately. Regular statements are other types of statements in the language that do not alter the flow of control, e.g. variable assignments or declarations. We also consider function call sites as regular statements since we do not consider interprocedural dependencies.

To be able to determine the post-dominators, we first need to assume some fundamental properties of the control-flow graph. Here follows a formal definition of the control-flow graph.

Definition 1

A *control-flow graph* for a program P is a directed graph G augmented with the unique nodes `START` and `STOP` such that every node in the graph has at most two successors. We assume that predicate nodes have two successors with attributes "T" (true) and "F" (false) associated with the outgoing edges. We assume that for any node N in G there exists a path from `START` to N and a path from N to `STOP`.

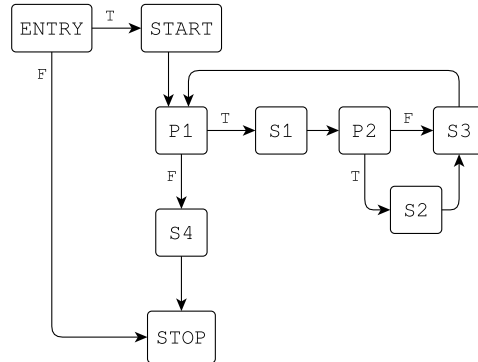
Apart from the definition, we also add the node `ENTRY` to our control-flow graph with one edge to `START` with attribute "T" and an edge to `STOP` with attribute "F". The `ENTRY`-node will represent any external reason the program is executed. The nodes `START` and `STOP` are unique nodes that represents the start and termination of control-flow. Figure 3.1 shows an example of a control-flow graph which illustrates the definition.


```

1  int main ()
2  {
3    while (P1)
4    {
5      S1;
6      if (P2)
7        S2;
8      S3;
9    }
10  S4;
11 }

```

Pseudo-code



Control-flow graph

Figure 3.1. Illustrative example of a program’s control-flow graph. P1 and P2 are predicate nodes and represent boolean expressions. S1 to S4 represent regular statements.

3.1.2 Control Dependence

The definition of control dependence is expressed in terms of post-dominators in the control-flow graph [9].

Definition 2

A node V is *post-dominated* by a node W in G if every directed path from V to STOP (not including V) contains W .

Definition 3

Let G be a control-flow graph. Let X and Y be nodes in G . Y is *control dependent* on X iff

- (1) there exists a directed path P from X to Y with *any* Z in P (excluding X and Y) post-dominated by Y and
- (2) X is not post-dominated by Y

Condition 1 can even be satisfied when P consists of only one edge since $Z \notin \emptyset$. Condition 2 can always be satisfied when X and Y are the same node.

The graphical representation of control dependencies can be shown in a control dependence graph, but also as a subgraph of a program dependence graph which we call the control dependence subgraph (CDS). Apart from the regular statement and predicate nodes that are already mentioned, we insert *region* nodes to the CDS to summarize the control dependencies from a node. The explanations of the different types of nodes that a CDS can consist of are given in Table 3.1.

Since the post-dominator relation is transitive, we can express it in a hierarchical graph called the *post-dominator tree*. Figure 3.2 shows an example of a control dependence subgraph calculated from a control-flow graph along with the corresponding post-dominator tree. In the graphs of this report we will use the conventional way of representing control dependencies used by Ferrante et al. [9], which is by using the reverse direction of the control dependence relations. This is the way DETECT represents control dependencies and helps for data-flow analysis

Statement Node	The statement node represent a regular program statement. This corresponds to a vertex in the program's control-flow graph that does only have one exit.
Predicate Node	A predicate node which represents a boolean expression. It corresponds to a node in the program control-flow graph that have two exits.
Region Node	The region node does not correspond to a program statement. It is inserted to summarize dependencies from a node and it can be seen as an entry point to a sequence of statements or a program block.

Table 3.1. Types of nodes in the control dependence subgraph.

later on in the generation process.

There is not always an intuitive way of determining control dependencies between the nodes, especially if the control-flow graph is complex. There is one algorithm that determines control dependencies which uses annotations on the post-dominator tree [9], but a more straightforward way is to look at every node pair and see if both conditions for control dependency are met, which will suffice for showing that the control dependencies in Figure 3.2 holds. E.g. by looking at node P_1 and P_2 .

- (1) There is a path in the control-flow graph from P_1 to P_2 with the path $P_1 \rightarrow S_1 \rightarrow P_2$. S_1 is post-dominated by P_2 , therefore condition 1 is met.
- (2) By looking at the post-dominator tree we can see that P_1 is not post-dominated by P_2 , therefore condition 2 is met and we have shown that P_2 is control dependent on P_1 .

Another more intuitive and non-computational way of grasping control dependencies is to look at the structure of the program code. A node representing a statement inside the body of an `if` statement or a `for` loop always depends on the predicate node stating the condition for the execution. Another notion is that there exists no control dependencies between statement nodes in the same block unless some form of control statement has been used in that block, such as `break`, `continue` or `goto`.

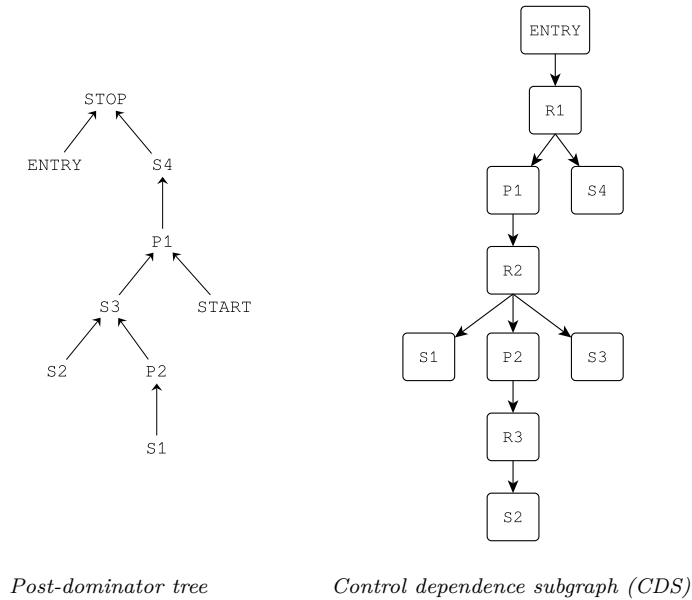


Figure 3.2. Post-dominator tree and control dependence subgraph from the example in Figure 3.1. R_1 to R_3 are inserted region nodes. The START and STOP nodes are not shown in the control dependence subgraph, they are only used for analysis purposes.

3.1.3 Data Dependence

Our formulation of data dependence is similar to the one given by Liu et al. [16],

Definition 4

There is a *data (flow) dependence edge* from a node v_1 to v_2 if there is some variable **var** such that:

- v_1 may do an assignment to **var**.
- v_2 may use the value of **var**.
- There is an execution path in the program from the code corresponding to v_1 to the code corresponding to v_2 along which there is no assignment to **var**.

Data dependencies in program code arise from statements or expressions trying to access or modify the same resource. The type of data dependencies used by the PDG is called *flow-dependence*, which exists between a statement defining or modifying a resource (variable) and another statement along the execution path using that resource, without any intervening modifications of the variable in the path. To find such paths, we must perform data-flow analysis on the control-flow graph which allows us to tell which execution paths that contains no assignments of a given variable. One such form, or *schema*, of data-flow analysis is called *reaching definitions*.

3.1.4 Reaching Definitions

We determine the data dependence in the graph by using the *reaching definitions* data-flow schema¹ as described by Aho et al. [1]. By using reaching definitions we can determine which variables are *live* at a given point in a program's flow of control, which means that they hold values that will be used later on. We can also tell in which node or nodes those variables were defined from a given program point.

The schema states the dataflow in terms of *definitions* that flow in and out of nodes in the control-flow graph. A *definition* is a 2-tuple : (v, var) where the control-flow graph node v may do an assignment to var . Each node is assigned the set of variable definitions that might reach it and the set of definitions that comes "out" of it, which we call the IN set and the OUT set.

Definition 5

The IN set of a control-flow graph node N is the set of all definitions that might reach node N , denoted $\text{IN}[N]$. The OUT set of a node N is the set of definitions that comes from node N , denoted $\text{OUT}[N]$.

To be able to calculate the IN and OUT sets for a node properly, it is necessary to look at all the definitions that the node generates. This is done by assigning each node GEN and KILL sets.

Definition 6

The GEN set of a node N is the set of definitions generated by the statement(s) the node represent, which we denote GEN_N . The KILL set of N is the set of all other definitions in the program of the variables defined in N , denoted KILL_N .

We make a difference between the notation of these sets to emphasize that the GEN and KILL sets are regarded to be constant during the analysis. The IN and OUT sets are variables and can be incrementally calculated using an algorithm that employs this data-flow scheme.

The OUT set for a node N can be calculated using the transfer function of N ,

$$\text{OUT}[N] = \text{GEN}_N \cup (\text{IN}[N] - \text{KILL}_N)$$

and by assigning a relationship between a node's IN set and the OUT set from its predecessors

$$\text{IN}[N] = \bigcup_{P \text{ is a predecessor of } N} \text{OUT}[P].$$

we must also assume that

$$\text{OUT}[\text{ENTRY}] = \emptyset$$

A definition flows through N from any predecessor P unless it is killed. We *kill* a definition of a variable var if there is any other definition of var anywhere along the path of the data-flow. In practice, the KILL set of a node is not something

¹Reaching definitions can only spot data flow dependencies for scalars and entire arrays, and is not sufficient for analysing individual array elements.

that has to be calculated for the analysis to work. The expression $(\text{IN}[N] - \text{KILL}_N)$ can be simplified by iteratively removing definitions of the same variable from the $\text{IN}[N]$ set looking at the definitions in the GEN_N set.

Algorithm

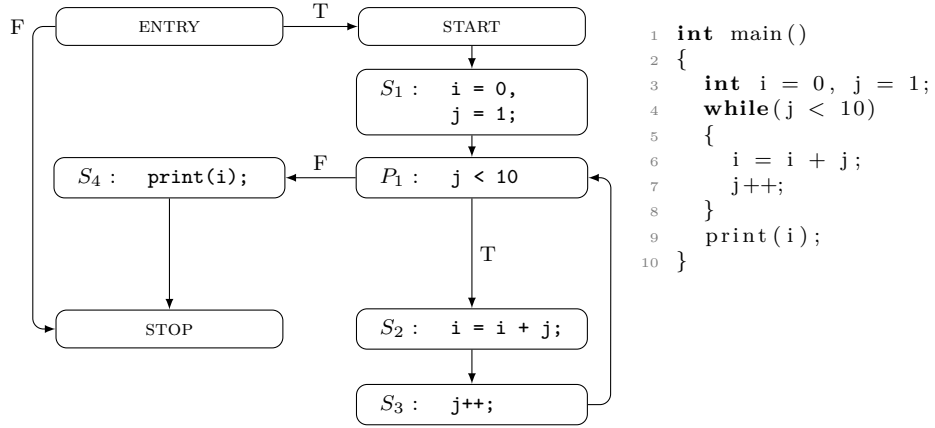
Since the transfer function of the reaching definitions scheme is monotonic,

$$N_1 \sqsubseteq N_2 \Rightarrow \text{OUT}[N_1] \sqsubseteq \text{OUT}[N_2]$$

a fixed-point iteration algorithm can be used to generate the data dependencies [13]. Explanation on how DETECT calculates the IN and OUT-sets to generate data dependencies is given in Section 4.7.4.

Example

Figure 3.3 illustrates the data-flow analysis scheme with a control-flow graph. In the figure, the IN and OUT-sets have converged to the point described by the equations.



$$\begin{aligned} \text{GEN}_{S_1} &= \{(S_1, i), (S_1, j)\} \\ \text{KILL}_{S_1} &= \{(S_2, i), (S_3, j)\} \\ \text{IN}[S_1] &= \emptyset \\ \text{OUT}[S_1] &= \{(S_1, i), (S_1, j)\} \end{aligned}$$

$$\begin{aligned} \text{GEN}_{S_3} &= \{(S_3, j)\} \\ \text{KILL}_{S_3} &= \{(S_1, j)\} \\ \text{IN}[S_3] &= \{(S_1, j), (S_3, j), (S_2, i)\} \\ \text{OUT}[S_3] &= \{(S_3, j), (S_2, j)\} \end{aligned}$$

$$\begin{aligned} \text{GEN}_{P_1} &= \emptyset \\ \text{KILL}_{P_1} &= \emptyset \\ \text{IN}[P_1] &= \{(S_1, i), (S_1, j), (S_2, i), (S_3, j)\} \\ \text{OUT}[P_1] &= \text{IN}[P_1] \end{aligned}$$

$$\begin{aligned} \text{GEN}_{S_4} &= \emptyset \\ \text{KILL}_{S_4} &= \emptyset \\ \text{IN}[S_4] &= \{(S_1, i), (S_1, j), (S_2, i), (S_3, j)\} \\ \text{OUT}[S_4] &= \text{IN}[S_4] \end{aligned}$$

$$\begin{aligned} \text{GEN}_{S_2} &= \{(S_2, i)\} \\ \text{KILL}_{S_2} &= \{(S_1, i)\} \\ \text{IN}[S_2] &= \{(S_1, i), (S_1, j), (S_2, i), (S_3, j)\} \\ \text{OUT}[S_2] &= \{(S_1, j), (S_3, j), (S_2, i)\} \end{aligned}$$

Figure 3.3. Illustrative example of the reaching definitions scheme. S_1 to S_4 are statement nodes and P_1 is a predicate node.

3.1.5 Program Dependence Graph

We formulate the definition of a program dependence graph in the same way as by Liu et al. [16].

Definition 7

The *program dependence graph* G for a procedure P is a 4-tuple $G = (V, E, \mu, \delta)$ where

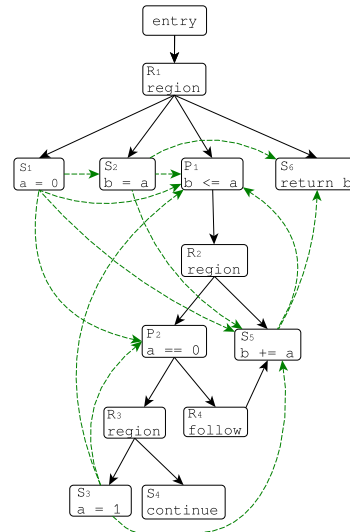
- V is the set of program nodes in P .
- $E \subseteq V \times V$ is the set of data and control dependency edges
- $\mu : V \rightarrow S$ is a function assigning types to program nodes from P ,
- $\delta : E \rightarrow T$ is a function assigning dependency types to edges from P .

```

1  int main()
2  {
3    int a = 0;
4    int b = a;
5
6    while(b <= a)
7    {
8      if(a == 0)
9      {
10       a = 1;
11       continue;
12     }
13
14     b += a;
15   }
16
17   return b;
18 }

```

Program code



Program dependence graph

Figure 3.4. Illustrative example of a program dependence graph. R_1 to R_3 represent region nodes, S_1 to S_6 represent regular statements and P_1 , P_2 represent predicate nodes. Regular lines represent control dependencies. Dashed lines represent data-dependencies.

The dependency from S_3 to P_1 in Figure 3.4 is an example of *loop-carried* data dependency, the dependency exists since data flows from the `continue` node S_4 back to P_1 . Similar reasoning holds for the dependency between S_5 and P_1 .

The dependency from S_3 to S_5 is less intuitive. If `a == 0` is true, the definition of `a` in S_3 will flow to the beginning of the while loop via the `continue` statement. But in the second iteration, the statement might be false, so there is still a path where that definition can flow from S_3 to S_5 without being killed. In fact there are an infinite number of such paths, it depends on how many times we follow the while loop without entering the if statement.

The figure also contains a case where a `continue` statement has been used. The `continue` statement on line 11 results in control dependency edges between the nodes P_2 and S_5 via the added region node R_4 . The region node R_4 is called a *follow* region and was first proposed by Ballance and Maccabe [3] to add missing control dependencies due to statements generating structured transfer of control.

3.2 Plagiarized Program Dependence Graphs

Plagiarized PDGs are nearly invariant during most forms of plagiarism. Control replacements will not modify the control dependencies as long as the replacements are defined. Adding statements that do not modify the semantics of the program will in the worst case just add control and/or data dependence edges.

Reordering statements at the same nesting depth will generate equal PDGs if the relative order of data dependencies between the statements are maintained.

Therefore, an original PDG can be seen as graph isomorphic or subgraph isomorphic to the plagiarized one. The method of finding plagiarized PDGs contains the problem of *subgraph isomorphism* testing. DETECT also allows to find plagiarizations by looking for *graph monomorphisms*.

Subgraph-isomorphism testing is in general a NP-complete problem [7]. However, under the application of program dependence graphs this becomes manageable due to the fact that program dependence graphs are not general graphs. First of all, PDGs are limited in size since they represent procedures that often are written with certain design principles in mind that make them small and manageable. Secondly, PDGs contain directed edges and particular types of nodes which allows backtracking algorithms to become more efficient.

3.2.1 Graph Morphisms

We say that a program dependence graph G is a plagiarization of another program dependence graph G' if G is either subgraph isomorphic or graph monomorphic to G' . The terminology we use for graph isomorphism, graph monomorphism and sub-graph isomorphism are given here, these are similar to those given by Liu et al. [16].

Definition 8

A bijective function $f_{iso} : V \rightarrow V'$ is a *graph isomorphism* from a PDG $G = (V, E, \mu, \delta)$ to an PDG $G' = (V', E', \mu', \delta')$ if

- (1) $\mu(v) = \mu'(f_{iso}(v))$
- (2) $\forall e = (v_1, v_2) \in E, \exists e' = (f_{iso}(v_1), f_{iso}(v_2)) \in E'$ such that $\delta(e) = \delta(e')$
- (3) $\forall e' = (v'_1, v'_2) \in E', \exists e = (f_{iso}^{-1}(v'_1), f_{iso}^{-1}(v'_2)) \in E$ such that $\delta(e') = \delta(e)$

Condition (1) specifies that there must be a mapping from all the nodes in G to G' and that they must have the same node type. Condition (2) specifies that there must also be a mapping between the edges and that their type must be the same. Condition (3) completes the definition by saying that the edge isomorphism must be bijective.

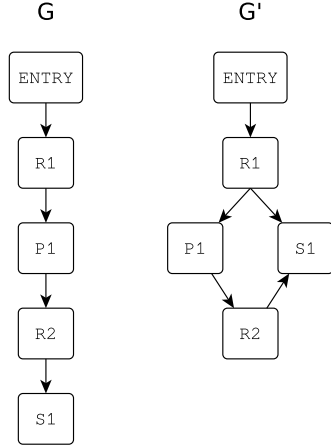


Figure 3.5. Illustrative example of two program dependence graphs that are monomorphic but not sub-graph isomorphic.

Definition 9

An injective function $f_{sub} : V \rightarrow V'$ is a *sub-graph isomorphism* from G to G' if there exists a node-induced subgraph $S \subset G'$ such that f_{sub} is a graph isomorphism from G to S .

Definition 10

An injective function $f_{mono} : V \rightarrow V'$ is a *graph monomorphism* from a program dependence graph $G = (V, E, \mu, \delta)$ to a program dependence graph $G' = (V', E', \mu', \delta')$ if

- (1) $\mu(v) = \mu'(f_{mono}(v))$
- (2) $\forall e = (v_1, v_2) \in E, \exists e' = (f_{mono}(v_1), f_{mono}(v_2)) \in E'$ such that $\delta(e) = \delta(e')$

Graph monomorphisms are a slightly weaker form of sub-graph isomorphisms. It is only required that all nodes and edges can be mapped from G to G' . As long as the graph is contained it is sufficient, the second graph can have both extra edges and nodes.

Figure 3.5 shows an example of two program dependence graphs that are only (of those mentioned) graph monomorphic. They are not sub-graph isomorphic since the only node-induced sub-graph in G' that is large enough (G' itself) contains an extra edge (R_1, S_1) that can not be mapped from G . This situation often arises whenever follow regions are added and shows that graph monomorphism is necessary to test whenever unstructured control statements have been used.

3.3 Algorithms for subgraph isomorphism testing

There are several applications of subgraph isomorphism testing. Some of the examples include pattern analysis, pattern recognition and computer vision.

The first algorithms dealing with graph-isomorphism testing proposed brute-force enumeration solutions. However these kinds of algorithms often become very

inefficient, yielding high time and memory complexities when testing large graphs. Many low-complexity algorithms exist. Some apply topology constraints on the graph, such as trees, planar graphs and bounded valence graphs, but relatively efficient algorithms applying no restriction exists as well. Two commonly used algorithms that do not impose any constraints on graph topology are Ullman's algorithm [21] and the VF2 algorithm by Cordella et al. [7].

Ullmans algorithm performs a tree-search enumeration in adjacency matrices and uses refinement procedures to backtrack and prune the tree-search space.

The VF2 algorithm is a backtracking algorithm that employs a state space representation with feasibility rules to prune the search tree. Each state corresponds to a partial solution of the graph matching, transfer of states represents the addition of a matching pair of nodes.

3.3.1 Time Complexities

A complexity comparison between Ullmans and the VF2 algorithm has been performed by Cordella et al. [7]. The summary of this study is shown in Table 3.2.

Algorithm	Time (Worst)	Time (Best)	Space (Worst)	Space (Best)
Ullman's algorithm	$\Theta(N!N^2)$	$\Theta(N^3)$	$\Theta(N^3)$	$\Theta(N)$
VF2	$\Theta(N!N)$	$\Theta(N^2)$	$\Theta(N!N)$	$\Theta(N)$

Table 3.2. Complexity comparison between Ullmans algorithm and the VF2 algorithm.

3.4 Construction of Approximated Program Dependence Graphs

The conventional way of computing a program dependence graph is done by constructing the control-flow graph, post-dominator tree and then by generating the control and data dependencies from these.

Another approach proposed by Harrold et al. [11] introduces a new algorithm to efficiently construct a program dependence graph directly from abstract syntax trees. Since this algorithm generates PDGs without post-dominator information, time and memory usage can be saved in the generation process. The algorithm proposed handles both structured and non-structured programs.

This method constructs the PDG in several passes:

- 1) The first pass generates the CDS by iterating through the abstract syntax tree. The control-flow information becomes implicit since the order of nodes is preserved.
- 2) In the case where non-structured statements such as `goto` are encountered the method adds explicit control-flow edges, and performs an additional pass of computation to remedy approximations.
- 3) The final pass(es) generates the DDS by performing data-flow analysis on the CDS to compute edges representing either flow-, anti- or output-dependence, depending on the application.

DETECT implements an adapted version of this algorithm to generate program dependence graphs, however our system generates approximations. We will therefore constrain ourselves to call the generated graphs used by DETECT Approximated Program Dependence Graphs (APDGs), Approximated Control Dependence Subgraphs (ACDS) and Approximated Data Dependence Subgraphs (ADDS). These factors contribute to the approximations:

Pointers, Aliases

The Abstract Syntax Tree representation used in our application does not give exact memory locations of the data used by the program. For instance, variables indirectly accessed via pointers or aliases will not be handled. The fact that pointer-addresses can be computed during run-time further complicates this problem.

Goto statements

DETECT generate the control-flow and control dependencies in only one pass, which makes the control dependencies unpredictable in unstructured programs. Although DETECT handles statements such as the `continue` or `break` correctly when constructing control and data dependencies, the `goto` statement is only partially handled by our system.

Exceptions

High-level control-structures such as exception throwing and catching generate control transfers in the program. Although the transfers are structured, they resemble the type generated by *goto* statements and therefore not handled by DETECT. This is further discussed in Section 8.3.2.

Chapter 4

Implementation

Our work consisted of the design and implementation of a plagiarism detection system for C, C++ and Ada. This chapter presents the requirements, design and implementation details of a system that we call COJAC, including its subsystem DETECT.

4.1 Requirement summary

From the customer elicitation a list of requirements were collected. Most of them specify functional requirements, but there are also requirements on documentation. This section summarizes the most important functionality of the system. The entire list can be found in Appendix A.

POSIX Conformance

The system would have to be able to run on a POSIX conformant platform.

Language Support

The system would have to support analysis of code written in at least C, C++ and Ada since these languages are used in the most popular courses taught by the UPP group. DETECT's APDG-Analysis currently only supports C and C++. Full Ada support is left over for future work, which is further discussed in 8.1.

Java and MatLab support was also of interest but this was left as requirements with lower priority.

Versatility

For the system to be effective against plagiarism and to be able to detect a wider spectrum of code disguises, the tool would have to examine the code on multiple levels of abstraction.

It was requested that the tool should be robust against format alteration, identifier renaming, declaration reordering, statement reordering, code insertion

and control replacement. This matched what would be expected from using AST and PDG based approaches.

In addition to this, it was also of interest to find purely textual matches and an output from the system that could specify which lines of code in one file matched another. To meet this functionality we also decided to analyse the code on a textual level.

Modularity

It was necessary that the system was designed with modularity in mind so that support for new programming languages and detection techniques could be added later on.

Documentation

It was requested that there existed documentation on how to integrate new functionality for the system, such as other types of detection techniques and language support.

4.2 System Overview

Figure 4.1 shows an overview of COJAC. COJAC was intended to be a simple graphical user interface with functionality for managing assignments and for the presentation of correlation metrics. COJAC does not perform any comparison of code. This is set aside for a subsystem that we call DETECT. DETECT is the actual tool that correlates and analyses two separate units of code and it is the system which our work focuses on. The implementation of COJAC is currently just a simple script and extending it can be the subject of a future project proposal.

Since it is necessary to integrate different front-ends so that new languages can be supported we designed DETECT to consist of easily integratable modules, as shown in Figure 4.2. There are two separate parts of DETECT that are modular: the front-end used for the language and the analysis to be performed. The APDGs are generated directly from the AST, which decouples the parser from the generation process.

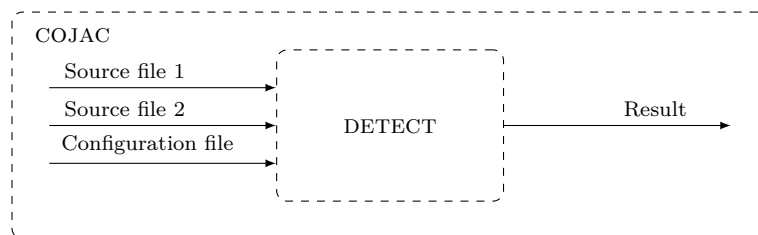


Figure 4.1. System overview of COJAC.

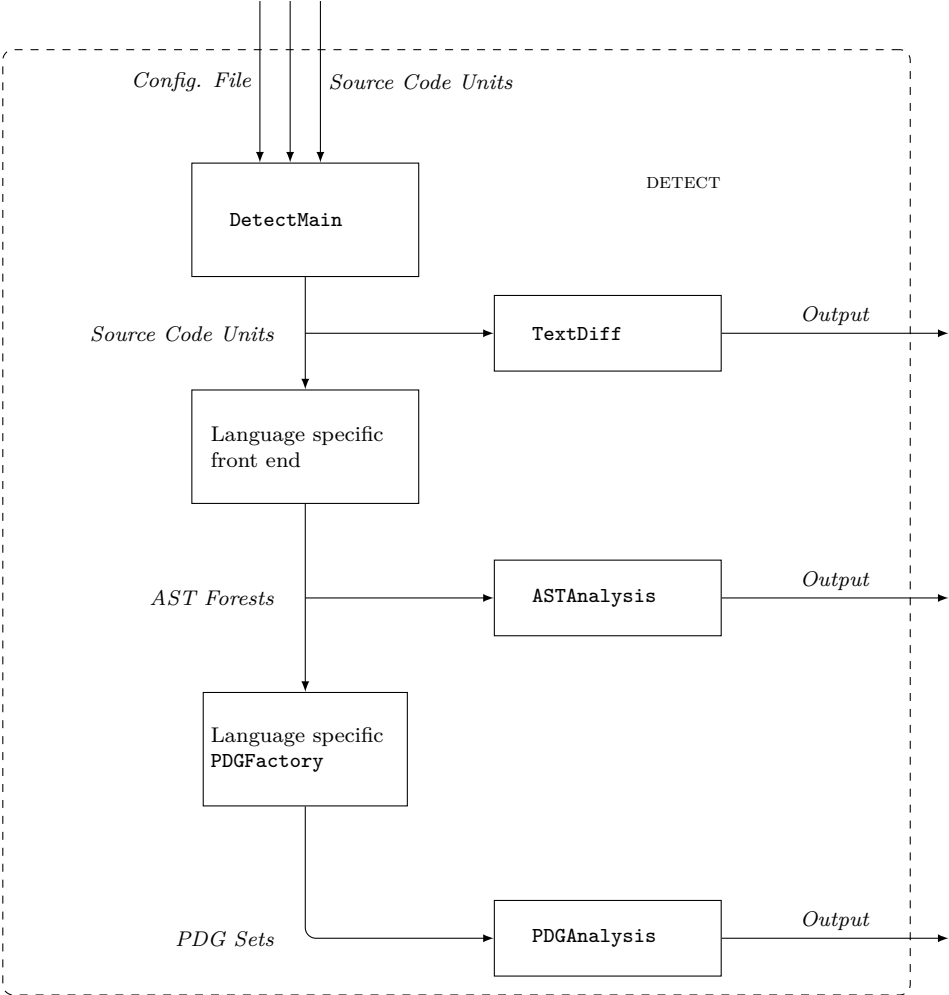


Figure 4.2. System overview of DETECT.

4.2.1 Modules

DETECT consists of six main modules to be able to analyse and correlate code.

- `DetectMain` is the module that is first run when a user invokes DETECT. Depending on the settings in the configuration file, it in turn invokes other modules for plagiarism detection.
- `TextDiff` performs textual matching.
- The language specific front end is a parser that recognizes the target language.
- The `ASTAnalysis` performs analysis of abstract syntax trees.
- The `PDGFactory` module generates Approximate Program Dependence Graphs from the transformed syntax tree given by the `ASTAnalysis` module.
- The `PDGAnalysis` module analyses the Approximate Program Dependence Graphs to find plagiarizations.

4.3 External libraries and dependencies

4.3.1 VFLib

To find graph morphisms among APDGs, we have used the VFLib [10] C++ library. VFLib was written by Pasquale Foggia, one of the authors of the VF2 algorithm. It implements the VF2 algorithm and is designed to work with simple directed graphs with attributes on both edges and vertices, which are the prerequisites for APDG matching.

4.3.2 Boost

DETECT is written in C++ and relies heavily on the Boost C++ libraries [5]. For DETECT to compile, several boost libraries must be available on the host system: `boost_system`, `boost_filesystem`, `boost_regex` and `boost_thread`.

4.4 Configuration of modules

Each step of the analysis is individually configurable by using the configuration file. Among other, it is possible to specify exactnesses, thresholds and type of output. Whenever a configuration file excludes a value, DETECT will automatically fallback to the default configuration file which holds default values for all configurable options. The default configurations can be found in Appendix D.

4.5 TextDiff Module

The DETECT TextDiff module finds disjoint sections of similar code inside two code units by using the `LCSSet` matching algorithm. Under exhaustive search the algorithm will find the largest sections of mappings between similar code.

Before matching pieces of code, we need to preprocess the files using a pre-processor. The (simple) preprocessor we have written, the *detect preprocessor*, perform inclusions for both Ada, C and C++ files. It does so by recursively following each inclusion directive that specifies a local file that has not already been included. From this, a preprocessed code unit is generated in a separate file. After preprocessing, the system generates an index-file that indexes the beginning of each non-empty line in the preprocessed code unit.

4.5.1 LCSSet

The algorithm `LCSSet` is a naive approach in finding the mappings of common distinct sections between two text files. It does so by looking at all possible line-sections in descending order of size and compares them line by line. It can be seen as a specialization of a string matching algorithm that finds all possible sub-string “patterns” inside another “text” string, where the symbols of these strings represent lines of code in the corresponding code units.

Input : Two code units with number of lines U_1 and U_2 , where $U_1 \leq U_2$.

Output: Disjunct mappings between common sections of lines.

Mappings, M_1 , $M_2 \leftarrow \emptyset$;

Initially call `LCSSet(U_1)`;

LCSSet(N):

for $i \leftarrow 0$ **to** $U_1 - N$ **do**

 /* S_1 is a line-section of size N in file 1 */

$S_1 \leftarrow (i, i + N)$;

if $S_1 \notin M_1$ **then**

for $j \leftarrow 0$ **to** $U_2 - N$ **do**

 /* S_2 is a line-section of size N in file 2 */

$S_2 \leftarrow (j, j + N)$;

if $S_2 \notin M_2$ **then**

if *MatchingSection*(S_1, S_2) **then**

Mappings \leftarrow *Mappings* \cup (S_1, S_2);

$M_1 \leftarrow M_1 \cup S_1$;

$M_2 \leftarrow M_2 \cup S_2$;

end

end

end

end

end

if $N > 1$ **then**

 | `LCSSet(N - 1)`;

end

Algorithm 1: LCSSet

MatchingSection(S_1, S_2) returns true iff the lines in the sections S_1 and S_2 of the code units are equal, after all whitespaces have been stripped off.

At every iteration step the algorithm goes through all the possible line sections

of size N in code unit 1 towards all possible line sections of same size in code unit 2.

Theorem

The algorithm finds *maximal* sections of distinct similar sections.

Proof outline

This is easily seen. If two sections are mapped and they are not maximal, they belong to some larger section that is maximal. Since the algorithm matches sections in descending order of size, this can not happen.

Execution time

According to the structure of the algorithm, the worst case time complexity is at least $\Theta(U_1^2 U_2)$. By performing a number of textual comparisons with the TextDiff module on randomly generated files, the average case time complexity was calculated to exist between U_1^2 and U_1^3 . Figure 4.3 shows the execution times for these comparison in relation to other exponential functions.

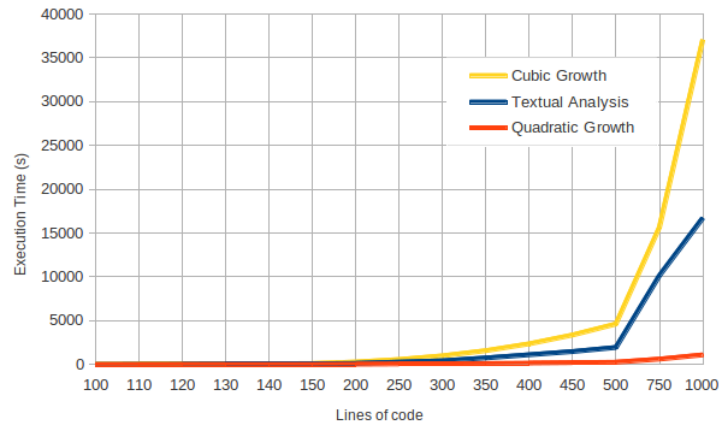


Figure 4.3. Execution time for the TextDiff depending on the number of lines in the code units.

Enhancing performance

In practice, the LCSSet becomes inefficient for code units with more than hundred lines of code. A test matching code units with 1000 lines took several hours to perform. To manage this, we let the recursive call to LCSSet step down with a configurable constant q instead of 1. We also make sure that $\text{LCSSet}(N_{min})$ always runs, to ensure that sections of a minimum size always get matched. In

the implementation of the TextDiff module we have chosen q as

$$q = \left\lceil k * \frac{U_1}{100} \right\rceil$$

where $k \in]0, 100]$ is a percentage of U_1 to step down with. The drawback of using this solution is that the sections of mapped code might not be maximal, where the advantage is that the algorithm can process files with an arbitrary number of lines within a reasonable amount of time. k and N_{min} can be specified by `TEXT_DIFF_SECTION_DECREMENT` and `TEXT_DIFF_SECTION_MIN` in the configuration file.

Other algorithms

Other string search algorithms such as Karp-Rabin are often used for plagiarism detection since it can handle multiple pattern strings. Wise [22] introduces a string similarity algorithm that uses greedy string tiling and Karp-Rabin matching to find the maximal coverage of distinct common substrings. It is shown that the *worst* case time complexity of this algorithm is $\Theta(n^3)$, but it is estimated that the average case exist between $\Theta(n)$ and $\Theta(n^2)$.

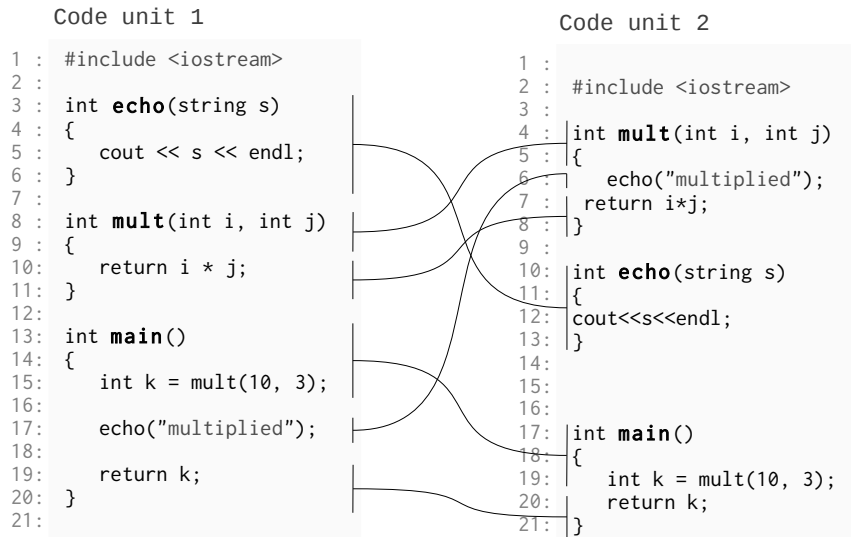


Figure 4.4. Illustrative example of the LCSSet matching algorithm.

```

Similarity report, matching
file1.dpp.cc towards file2.dpp.cc
-----
lines 3-6 were found equal to lines 10-13
lines 8-9 were found equal to lines 4-5
lines 10-11 were found equal to lines 7-8
lines 13-15 were found equal to lines 17-19
line 17 were found equal to line 6
lines 19-20 were found equal to lines 20-21
-----
Total of 14 out of 14 nonempty lines matched.

```

Figure 4.5. Similarity report from the TextDiff module.

Figure 4.4 shows an example of LCSSet running on code units where format alteration and statement reordering has been applied. Figure 4.5 shows a similarity report from DETECT's textual analysis performed on the code units. Empty lines and preprocessor directives are not matched since they are discarded during the preprocessing and line-indexing.

4.6 Abstract syntax tree module

To generate an abstract syntax tree from source code, a parser must be used. For the C and C++ AST generation, we have written a plugin for the GNU C++ compiler. For the Ada generation we have written our own Ada-parser using yacc.

When the code has been parsed into an abstract syntax tree, the language specific front-ends transforms the AST representation into another syntax tree data structure that is more suitable for APDG generation, which we call the *genericized* abstract syntax tree.

The implementation details of the `ASTAnalysis` module and the different front-ends is given in Nilsson [17].

4.7 APDG Generator Module

The APDG generator module is the module that generates Approximate Program Dependence Graphs from the syntax trees given by the AST module. The `PDGFactory` is the class that performs the generation of an APDG and it does so in two steps. It first generates the ACDS directly from the abstract syntax tree. Then it generates the ADDS from the ACDS and other information collected from the AST during the ACDS generation. It does so by using the reaching definitions data-flow analysis scheme previously presented in this report.

Since data dependencies in the ADDS depends on the control-flow of a program, control-flow edges are added during the ACDS generation.

To summarize:

- The module generates ACDS with control-flow edges.
- The module generates ADDS using the reaching definitions data-flow analysis scheme.

4.7.1 Data structures

To separate the analysis and construction of an APDG, two data-structures are used for its representation.

PDG

The PDG class is the data structure that contains all the nodes and edges of an APDG and is created by the `PDGFactory` when the APDG is built.

AnalysisGraph

Before the analysis of an APDG, a PDG is transferred into an `AnalysisGraph`. This data-structure provides an interface that is better suited for APDG analysis and captures the numbers, types and frequencies of nodes and edges that the graph consists of. This information is later on used in the matching process for pruning and output.

4.7.2 AST Interface

To be able to generate control and data dependencies from separate kinds of abstract syntax trees, we impose an interface on the data structure. The interface specifies different control types for the nodes in the genericized AST. The control type of each node depends on the statement(s) it represents. To be able to handle each node appropriately, the interface also specifies extra methods depending on the type of the node. These methods will allow for the retrieval of the separate components which each language construct can consist of. The different control types are presented in Table 4.1.

Node control type	Representation
ASSIGNMENT	An assignment statement.
DECLARATION	Declaration of a variable.
FUNCALL	Call site of a function.
STATEMENT_LIST	Block containing several statements. The statements are the child nodes.
IF_ELSE_BEGIN	Beginning of any if-else control structure, as well as possible following <code>else if</code> and <code>else</code> combinations.
DETECT_LOOP	Beginning of <i>any</i> possible loop control structure the language can provide.
CONTINUE	Redirection of control flow to next iteration of a <code>DETECT_LOOP</code> node.
BREAK	Redirection of control flow out of a <code>DETECT_LOOP</code> node.
GOTO	Redirection of control flow to a <code>LABEL</code> node.
LABEL	Entry point of control flow from a <code>GOTO</code> statement.
PROGRAM_EXIT	Represents the exit point of a procedure.

Table 4.1. Control types of AST nodes.

A `DETECT_LOOP` node is a generic representation for any loop construct that the language provides. In C++ it can either be a `while`, `do while` or `for` loop. To handle these loops in a generic way, the `DETECT_LOOP` node provides methods to determine and retrieve the parts that the loop consists of. The four parts that a `DETECT_LOOP` node can have are:

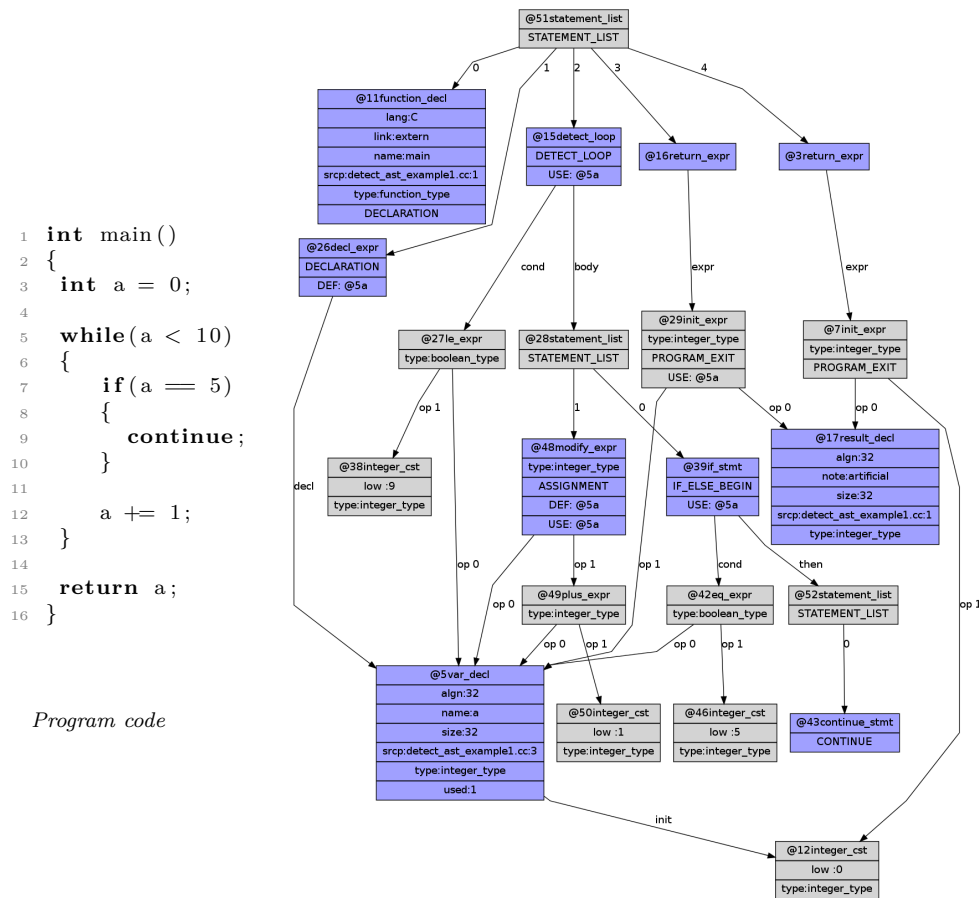
- Pre-condition
- Loop body
- Post-condition
- Post-statement

E.g. a `for` loop would (could) have a pre-condition, loop body and could have a post-statement, but a `while` loop would only have a pre-condition and a loop body. The loop body is necessary for all loop constructs and it must consist a `STATEMENT_LIST` AST-node containing all the statements inside the loop. If the loop body is empty, the `STATEMENT_LIST` will contain no children.

If a loop, such as a `for` loop, contains any initializing statements (declaration of variables etc.), such statements are assumed to appear just before the `DETECT_LOOP` node in the AST.

An IF_ELSE_BEGIN node has similar methods to retrieve the separate `else` if and `else` parts of the construct.

Figure 4.6 shows a genericized AST with assigned node control types. The tree has been automatically generated by DETECT from the Abstract Syntax Tree representation used in the GCC C++ compiler. In this tree there are two PROGRAM_EXIT nodes. The first exit point comes from the return statement on line 15, the second comes from an automatically added return statement by GCC. GCC adds this extra `return_expr` for all main functions.



Genericized abstract syntax tree

Figure 4.6. Example of a genericized abstract syntax tree generated by DETECT.

4.7.3 ACDS Generation

The approximate control dependence generation is done in the method `CDSGeneration`. It recursively descends the abstract syntax tree in left-to right preorder

iteration. At every step it generates ACDS nodes and control-flow edges depending on the control type of the current AST-node.

Context

To be able to correctly determine where control flow edges should be added in the separate invocations of `CDSGeneration`, we specify a *context* before every recursive call. The context is a specified set of APDG nodes that can be relevant whenever transfer of control is found, such as when the control reaches the end of a statement list or a node representing structured transfer of control. The nodes that a context consists of is described in Table 4.2.

Context node	Description
<code>currentPdg</code>	Specifies a parent node which any new APDG node should relate to.
<code>breakNode</code>	Specifies where control should flow when a break statement or the end of a statement list is reached.
<code>continueNode</code>	Specifies where control should flow when a continue statement is found.
<code>nextStmt</code>	Specifies where control flow goes if no explicit transfer of control flow is found.

Table 4.2. Context nodes used for determining control-flow in `CDSGeneration`.

Backpatching

Before the recursive descent of each subtree in the AST, we need to decide which nodes are to be forwarded as the context. If its necessary to forward a node that has not yet been created, we perform backpatching [1] and create *placeholder* nodes that will yield a valid destination for added control-flow and dependence edges. Placeholder nodes get replaced by properly generated nodes in subsequent calls to `CDSGeneration`.

Figure 4.7 shows an example of how generated placeholder nodes are used to build an ACDS for a program fragment.

- 1) In the first step, a region node is created for the statements of the program. Placeholder 1 is added as child and a control-flow edge is added from R1 to it.
- 2) In the second step, a new predicate node is added and replaces placeholder 1. A new placeholder node is created and control-flow edges are added from the predicate node to it. This edge represents the false-branch of control-flow.
- 3) A region node and a new placeholder node is added. `nextStmt` is set to the placeholder node. `CDSGeneration` descends down into the sub-tree of the `IF_ELSE_BEGIN` AST-node. A new region and placeholder node as well as control-flow edges are added to the APDG.
- 4) The statement node S1 takes over the position of the placeholder. Since there are no more statements inside the body of the if-statement, no more placeholder nodes are added to R2.

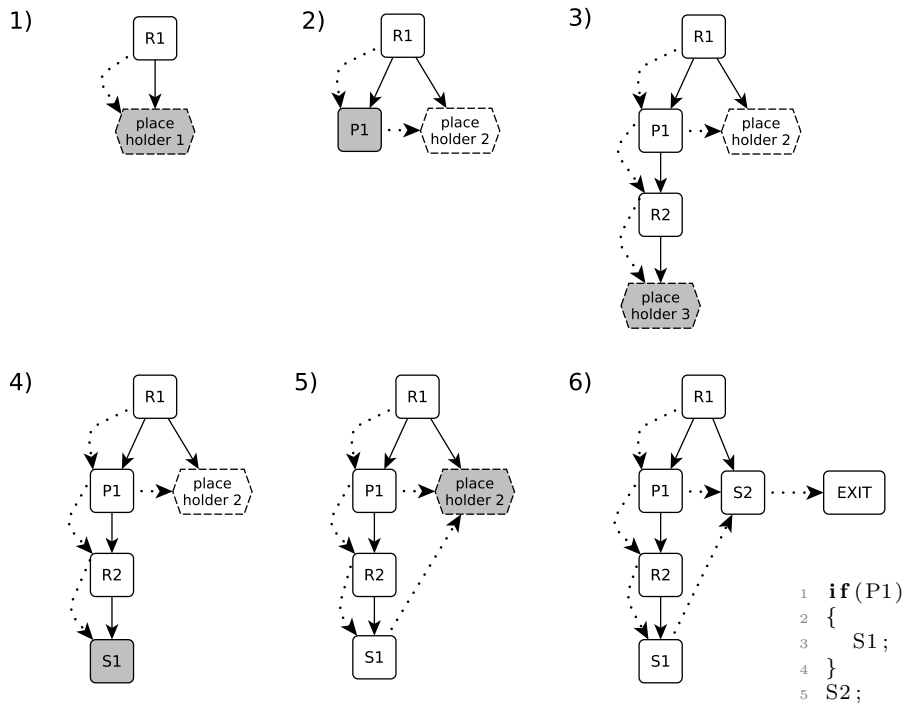


Figure 4.7. Illustrative example of the creation of an ACDS using placeholder nodes. Colored nodes specifies the `currentPdg` context node. Regular edges represent control dependence, dotted edges represent control-flow.

- 5) All statement-nodes inside the `if`-statement have been visited and the control flows back to the `nextStmt` node which was specified in step 3). `CDSGeneration` has finished the descent of the `if`-statement and the current node is now placeholder 2.
- 6) Since there are no more statements in the AST to traverse, the control flows to the exit-node.

For other control-structures the construction of the graph would be done in a similar way. If the program fragment contained a `while` loop instead of an `if`-statement, in step 3) the `nextStmt` would have been set to P1, the `breakStmt` would have been set to placeholder 2 and `continueStmt` would have been set to P1. If it had been a `do while` loop, P1 would have been added to R2 instead and R2 would have replaced the placeholder node in step 1). The control flow graph for `for` loops are generated in a similar way as `while` loops. In this case the `DETECT_LOOP` might have a post-statement and an ending statement-node will

be added to the region node of the loop.

If a node without a node control type is encountered, the procedure ignores the node and descends into its sub-tree without modifying the context.

Use and Def

The ASSIGNMENT, DECLARATION and FUNCALL nodes are AST nodes that might contain *use* and *def* sets. The *use* set contains all variables that are used in the node, and the *def* set contains all variables that are defined. If such information exists, it is added to the generated APDG node which is later used in the ADDS generation.

Handling structured transfer of control

Structured transfer of control is essential to handle if we are interested in detecting control-replacement disguises such as the one given in Figure 4.8.

<pre> 1 int main() 2 { 3 int sum = 0; 4 for(int i = 0; i < 10; ++i) 5 { 6 if(i % 2 == 0) 7 sum += i; 8 } 9 return sum; 10 } </pre>	<pre> 1 int main() 2 { 3 int sum = 0; 4 for(int i = 0; i < 10; ++i) 5 { 6 if(i % 2 != 0) 7 continue; 8 9 sum += i; 10 } 11 12 return sum; 13 } </pre>
--	--

Original Code

Plagiarism Example

Figure 4.8. Example of code where a more clever form of Control Replacement has been applied.

If we reach structured control statements during the descent of the AST, it might be the case that extra control-dependencies have to be added in the APDG. For an example, if the flow of control reaches a `continue` statement inside an `if`-statement which in turn resides in a loop, the remaining statements in the loop will be control dependent on the predicate of the `if`-statement. In this case, extra control-dependency edges have to be added between the remaining statements and the predicate node via follow regions. To solve this, in addition to the context we also forward a reference to a *dependency stack* in every recursive descent. Whenever we reach an `IF_ELSE_BEGIN` that has a `continue` or `break` statements in the body, we push the predicate node onto the dependency stack. If there are any statements following the `IF_ELSE_BEGIN` we then add a follow region and additional control dependencies from every node in the dependency stack to these statements (explained in Section 3.1.5). For this to work in nested loops

we create new dependency stacks. In the separate `else if` and `else` parts of an `if` statement we send a copy of the dependency stack and merge these after each recursive descent.

#	Pseudo code	Dependency stack	Added dependencies
1	while (P1)	$dep_1 = \{\}$	
2	{		
3	S1;		
4	if (P2)	$dep_2 = \{\}$	
5	{		
6	continue	$dep_2 = \{P2\}$	
7	}	$dep_1 = dep_1 \cup dep_2 = \{P2\}$	
8	while (P3)	$dep_3 = \{\}$	P3 to $dep_1 = \{P2\}$.
9	{		
10	if (P4)	$dep_4 = \{\}$	
11	{		
12	break ;	$dep_4 = \{P4\}$	
13	}		
14	else if (P5)	$dep_5 = \{\}$	
15	{		
16	continue ;	$dep_5 = \{P5\}$	
17	}	$dep_3 = dep_3 \cup dep_4 \cup dep_5 = \{P4, P5\}$	
18	S2;		S2 to $dep_3 = \{P4, P5\}$
19	}		
20	S3;		S3 to $dep_1 = \{P2\}$
21	}		

Figure 4.9. Illustrative example of the creation and merging of dependency stacks.

Figure 4.9 shows how the dependency stacks work for nested loops. The beginning of each block represents the descent of each language construct sub-tree, and the end of the block represents coming back from the descent. In step 1) a new empty dependency stack, dep_1 is created for the `while` loop descent. This will be used for all extra dependencies for the statements that exists in the block. In step 4) a new dependency stack, dep_2 is created for the if-statement, and when the `continue` statement in step 6) is reached, the predicate P2 is pushed onto dep_2 . When the descent comes out of the if-statement, dep_1 is merged with dep_2 . When we reach the while loop in step 8), all the nodes within dep_1 are added to the predicate node for the while loop (P3) via a follow region. In the descent of the `while` loop a new dependency stack is created. For the `if` statement and its `else if` part, new stacks are created. In handling these constructs, the predicate nodes P4 and P5 are pushed onto the corresponding dependency stacks. In step 17) dep_3 is merged with the stacks created for the `if` statement. In 18), dependencies from S2 are added to the nodes in dep_3 . When the procedure comes back to the overlying while loop in step 20) the dependency stack for the descent is dep_1 and extra dependencies are added from the newly created node S3 to P2. Examples of fully constructed APDG's for various types of programs are given in Section 4.7.5.

Handling unstructured transfer of control

Whenever `goto` and `label` statements are encountered, appropriate nodes are mapped to the given label name and control-flow is resolved in the end of the ACDS generation.

4.7.4 ADDS Generation

To transform the control dependence subgraph into a program dependence graph we must add data dependence edges. To calculate data dependencies we have implemented an algorithm which employs the reaching definitions data-flow scheme.

Calculation of data dependencies is done by iteratively calling the method `descentDDS()` on the root node which performs data flow-analysis on each node of the graph by recursively descending control flow edges to generate the IN and OUT sets that are necessary for scheme to work.

`DDSGeneration()` does not calculate the KILL set explicitly. The IN and OUT sets are instead calculated as explained in Section 3.1.4, by iteratively removing definitions by looking at the variables in the GEN set. The GEN set is made up of the variables in the *def* set. Whenever a variable occurs both in the *use* set and IN set, data flow dependency edges are added between the current node and the node that generated the definition.

At the end of each invocation of `descentDDS()`, we check to see if any OUT set has been changed. If any has, there might be unresolved data-flow and the algorithm is required to perform all calculations again and until all OUT sets have stabilized. This is implemented by using a boolean flag that tells whether any OUT set has been changed during the calculation step.

4.7.5 APDG Examples

Figures 4.10 to 4.15 shows examples of APDG's that have been generated by DETECT. Regular edges represent control-dependence, dashed edges represent data-dependence and dotted edges represent control-flow edges. The control-flow edges from predicate nodes has not been labelled 'T' or 'F' since control-flow information is of no interest for the APDG analysis.

Figure 4.10 shows an APDG representing a program with simple statements, illustrating the calculated data dependencies between them.

Figure 4.11 shows an APDG representing a program containing an `if`-statement with `else if` and `else` structures. In this figure we can see that the function `operator>>` used in line 8 both uses and defines the variable 'a', although it actually only defines the value and never reads it. Since DETECT does not handle reference variables or pointers it is not possible to know whether any external function actually uses or defines any actual parameter. In all function call sites we take a conservative stance and assume that both use and definition of the actual parameter will be done.

Figure 4.12 and 4.13 shows examples of a `while` and `do while` loop. In the `while` loop, the loop-condition predicate comes before statement 1 and in the `do while` loop the predicate comes afterwards.

Figure 4.14 shows an APDG representing a program containing a `for`-statement. In this graph, the statement representing the declaration of `'i'` is only dependent on the program region. This is due to the fact that `'i'` is not a part of the actual control-flow loop. The statement node 6 representing incrementation of `'i'` is set as the last statement of the control-flow loop and is control-dependent on the predicate node.

Figure 4.15 shows an APDG representing the AST in Figure 4.6. Dependencies from statement 2 to predicate 1 via follow region 4 is added as a result of the `continue` statement on line 9.

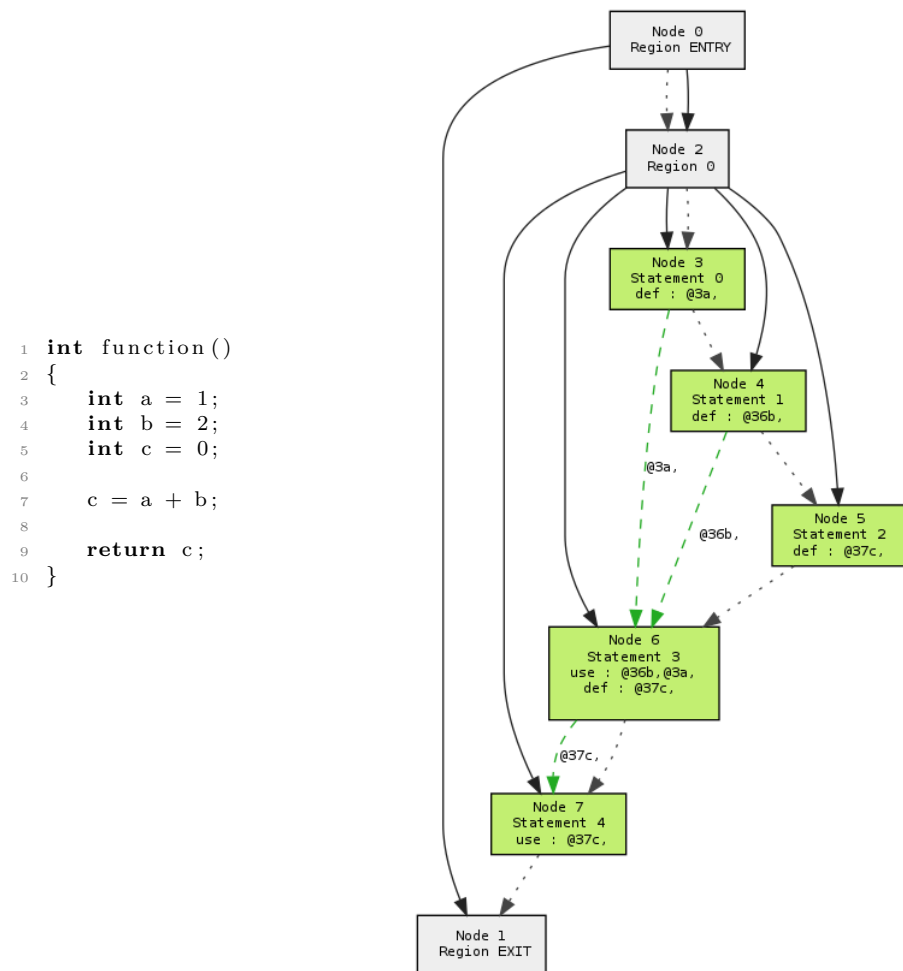


Figure 4.10. Example of an APDG for a C++ function generated by DETECT.

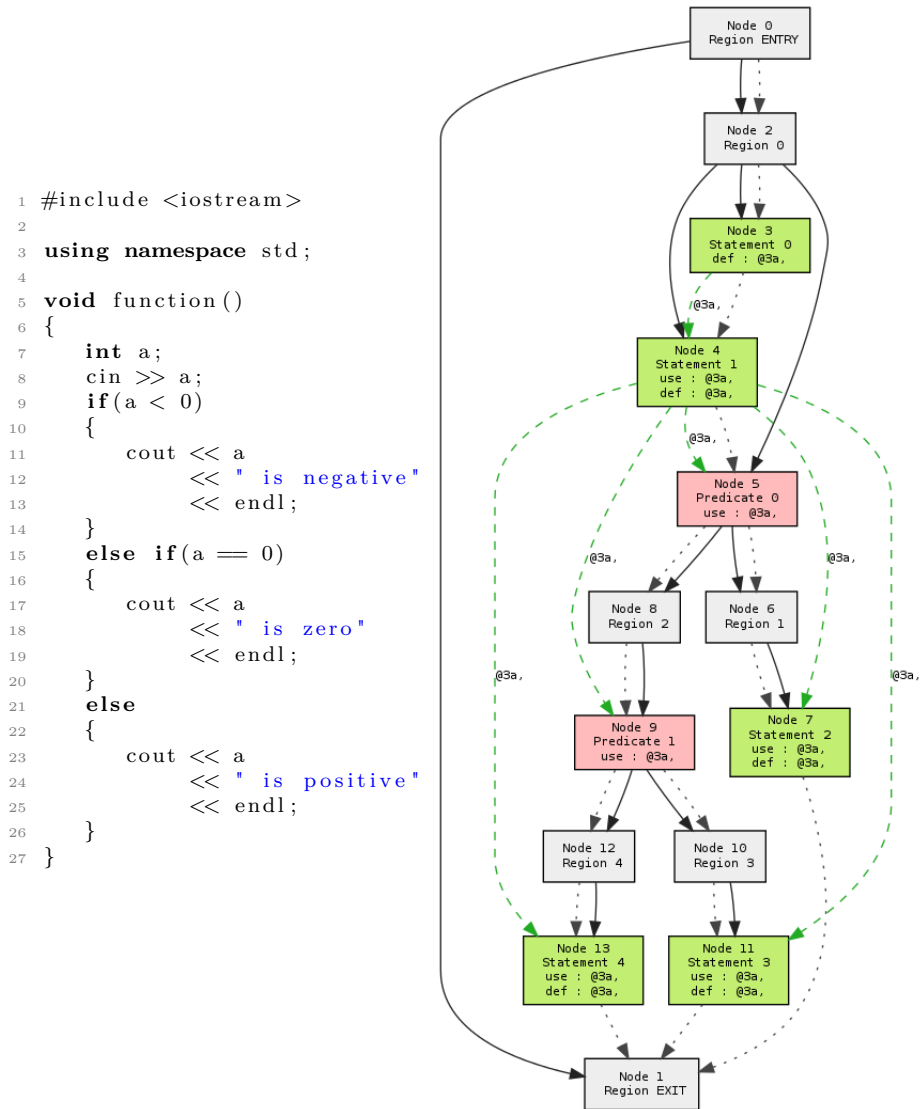


Figure 4.11. Example of an APDG for a C++ if-statement generated by DETECT.

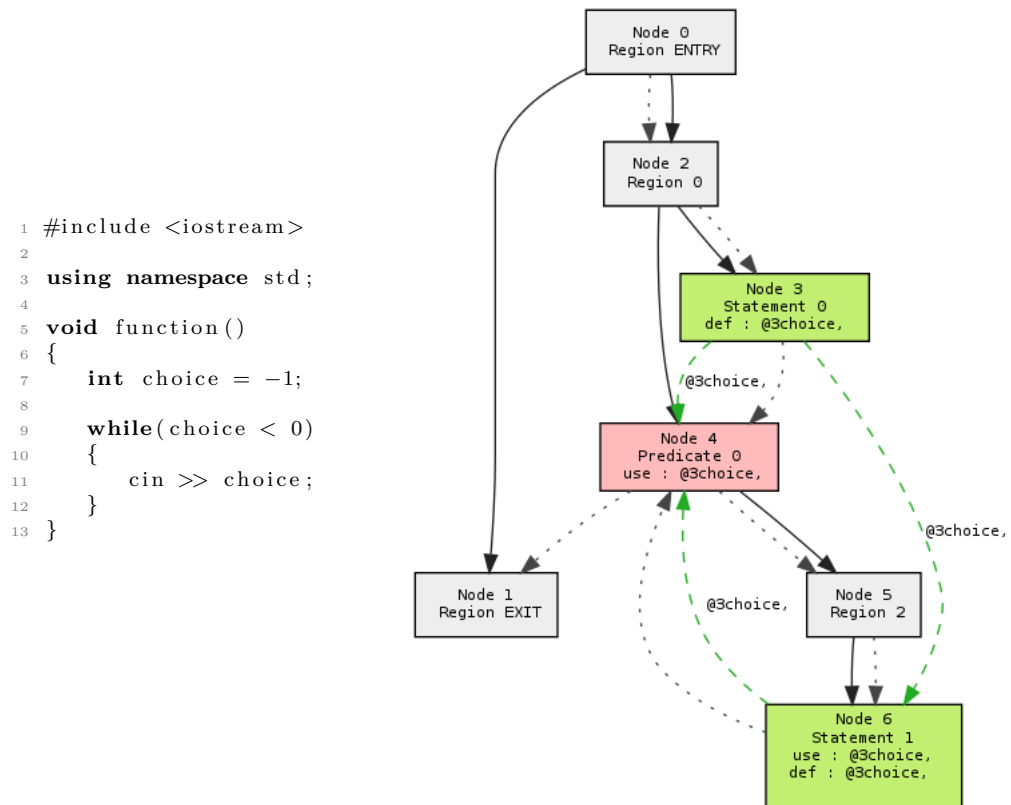


Figure 4.12. Example of an APDG for a C++ while-statement.

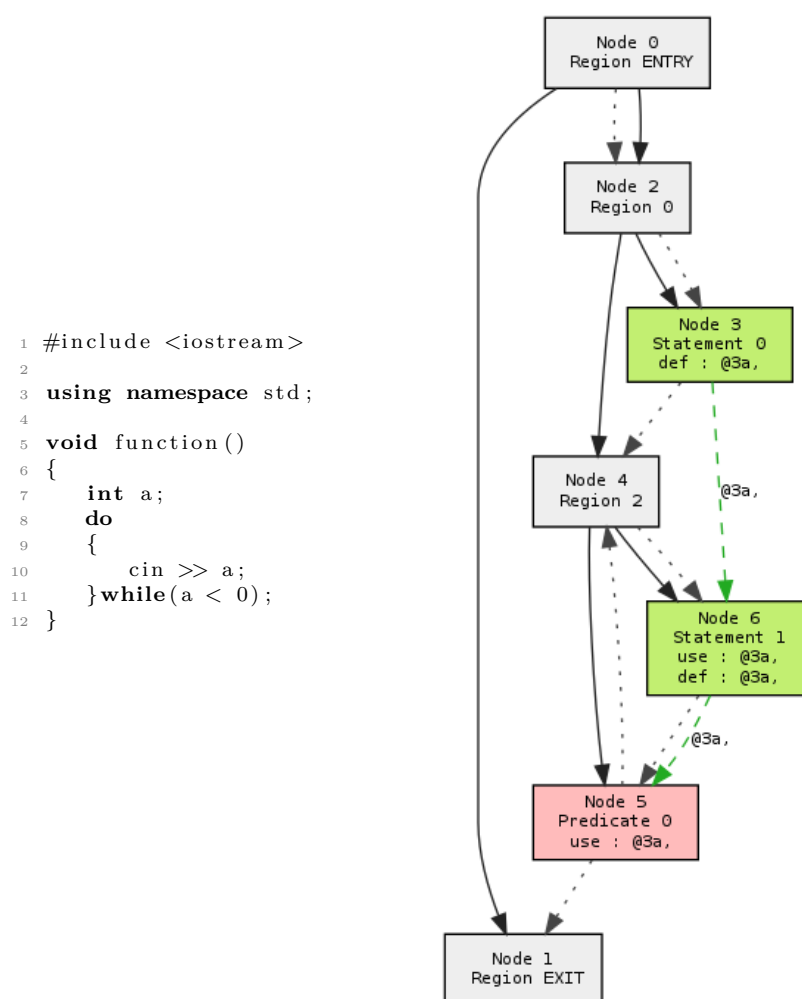


Figure 4.13. Example of an APDG for a C++ do-while-statement.

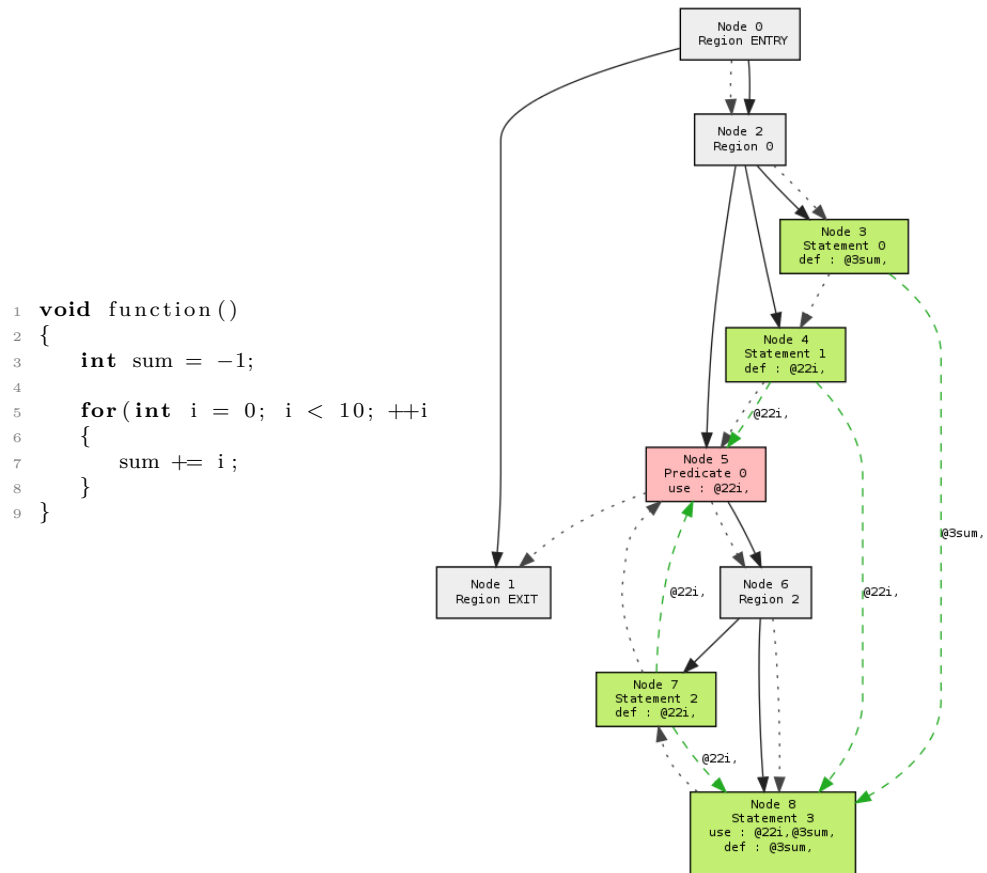


Figure 4.14. Example of an APDG for a C++ for-statement.

```

1  int main()
2  {
3      int a = 0;
4
5      while(a < 10)
6      {
7          if(a == 5)
8          {
9              continue;
10         }
11
12         a += 1;
13     }
14
15     return a;
16 }

```

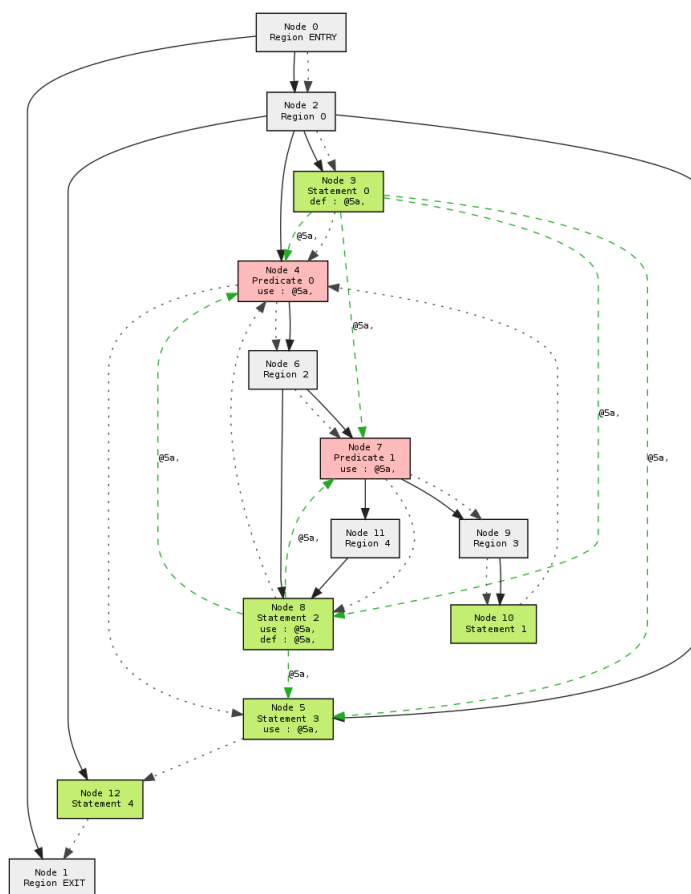


Figure 4.15. Example of an APDG generated from the genericized AST in Figure 4.6.

4.8 APDG Analysis Module

The `PDGAnalysis` is the module that analyses two sets of APDGs and tries to find plagiarisations between these. It initially takes two sets of abstract syntax trees and transforms them into sets of APDGs using the `PDGFactory`-module. After this the analysis module prunes the search space and applies the VF2 algorithm to test for graph morphism between the graphs in the first code-unit towards the graphs in the second. The type of graph morphism testing depends on the value `PDG_GRAPH_MATCH_TYPE` in the configuration file, and can be specified to be either exact isomorphism testing (`EXACT-ISO`), subgraph isomorphism testing (`SUB-ISO`) or graph monomorphism testing (`MONO`). The module finishes by printing similarity information that shows in each case which procedure matched which. Figure 4.16 illustrates the flow of construction and analysis of an APDG.

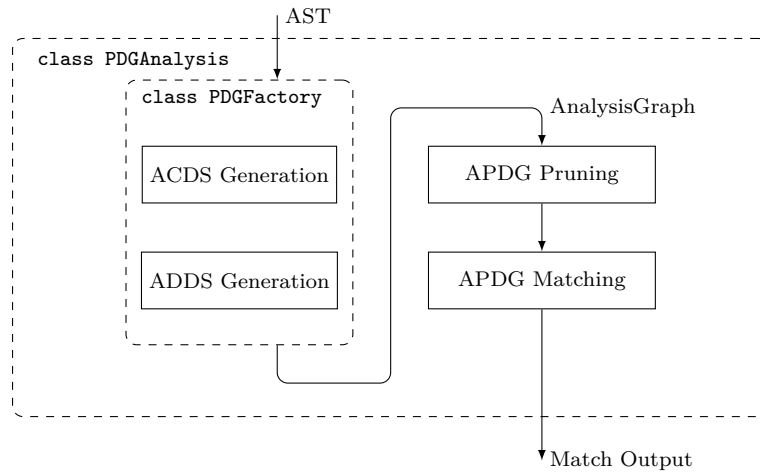


Figure 4.16. Overview of the APDG construction and analysis

4.8.1 Pruning the search space

A Program Dependence Graph is a very loose and stripped down representation of the program code. Small APDG's will represent very general programs and spurious matches can be found between large and small graphs if the match is not pruned beforehand. Although a spurious match still is a match, it is necessary to be able to discard distinct graph pairs from the matching process since they can be considered unique work in relation to each other.

Before directly comparing an APDG with any other APDG, the analysis module prunes out match pairs by looking at the number of nodes and frequencies of the node-types in the graph. The node type frequencies are captured during the generation of an `AnalysisGraph` from an APDG. The following criteria describe when a match pair (G_1, G_2) can be pruned from the matching process:

- G_1 can't be a subgraph isomorphism of G_2 due to the fact that it is smaller in regards to the number of nodes.
- The node ratio between graph G_1 and G_2 is beyond a constant threshold, specified by `PDG_NODE_RATIO_THRESHOLD`.
- The edge ratio between graph G_1 and G_2 is beyond a constant threshold, specified by `PDG_EDGE_RATIO_THRESHOLD`.
- The Euclidean distance between the node-type frequencies in G_1 and G_2 is beyond a constant threshold, specified by `PDG_NODE_FREQ_THRESHOLD`. This is given by the equation

$$\sqrt{(s_{G1} - s_{G2})^2 + (p_{G1} - p_{G2})^2 + (r_{G1} - r_{G2})^2} > \text{PDG_NODE_FREQ_THRESHOLD}$$

Where s , p and r are the frequencies of the number of statement, predicate and region nodes in the graphs.

- The euclidean distance between the edge-type frequencies in G_1 and G_2 is beyond a constant threshold, specified by `PDG_EDGE_FREQ_THRESHOLD`. This is given by the equation

$$\sqrt{(c_{G1} - c_{G2})^2 + (d_{G1} - d_{G2})^2} > \text{PDG_EDGE_FREQ_THRESHOLD}$$

Where c and d are the frequencies of the number of control and data dependency edges in the graphs.

- The number of nodes or edges in G_1 are below a minimum threshold, they can be specified by `PDG_EDGE_NUM_THRESHOLD` and `PDG_NODE_NUM_THRESHOLD`.

When a match pair has survived the pruning process, the `AnalysisGraphs` are transferred into `ArgGraphs` that are used by the VFlib matching library

4.8.2 Optimal Configuration

To be able to decide which configurable values are optimal when performing an analysis of the system, a simple machine learning procedure was used. The training set consisted of 27 procedures which had been modified using various forms of code insertion, a type of disguise which adds nodes and edges to the graph. The procedure consisted of lowering each of the distance and ratio based thresholds separately while keeping the other constant to minimize the number of spurious matches while maintaining the number of detected procedures. The results are shown in Table 4.3. The training set is available at <http://www.ida.liu.se/~nikho42/detect/learning/>.

4.8.3 Threading

Some isomorphism testing can take a lot of time to perform. In most cases the VFlib library will find these in microseconds while in some cases it can take several minutes or even hours to perform. This is an effect of the way the graph

Threshold Parameter	Optimal range
Node frequency distance	15-25%
Edge frequency distance	20-30%
Node ratio	35-45%
Edge ratio	65-75%

Table 4.3. Optimal values for the pruning thresholds.

matching algorithm works. To avoid such comparisons we have set time-outs for each matching process. We implemented this by using the *boost threads* library, creating a new thread for each match. If a thread does not finish within a given time it gets forcefully killed. The time-out for a thread can be specified in the configuration file. All critical sections are synchronized using boosts `scoped_lock` and `interprocess_mutexes`.

4.8.4 Output

The full output by the PDGAnalysis module is of textual form and contains several parts. First comes the APDG-legend, then the match-matrix, similarity ratio and time information.

APDG Legend

The APDG-legend lists each function signature and general information about the APDG's generated from each function. This include number of nodes and edges of the different types. Figure 4.17 shows an example of an APDG-legend.

```

A00 : "void bar int int "
      nodes: 16(3R / 0P / 13S )      edges: 38(11DE / 15CE )

A01 : "void foo int "
      nodes: 14(4R / 1P / 9S )      edges: 33(10DE / 13CE )

A02 : "void print double double double double "
      nodes: 10(3R / 0P / 7S )      edges: 25(8DE / 9CE )

B00 : "void function1 int "
      nodes: 14(4R / 1P / 9S )      edges: 33(10DE / 13CE )

B01 : "void function3 double double double double "
      nodes: 10(3R / 0P / 7S )      edges: 25(8DE / 9CE )

B02 : "int function4 "
      nodes: 28(3R / 0P / 25S )      edges: 92(32DE / 27CE )

```

Figure 4.17. Example APDG-Legend.

Match Matrix

The match matrix shows the result of each match-process between the procedures of the code units. To interpret the results, each match outcome is assigned a symbol. An example of a match matrix is shown in Figure 4.18. The explanation of the different symbols that a match can result in is shown in Table 4.4.

Symbol	Description
[M]	Match has been found between the graphs
-	No match between the graphs were found
-t	Graph match did not finish within specified time-out
*	Pruned due to incompatible graph sizes
*nr	Pruned due to node ratio exceeding specified threshold
*er	Pruned due to edge ratio exceeding specified threshold
*nf	Pruned due to node frequency distance exceeding specified threshold
*ef	Pruned due to edge frequency distance exceeding specified threshold
*s	Pruned due to (smallest) graph size below threshold

Table 4.4. Match symbols of the match-matrix.

```

          B00 B01 B02 B03
    +---+---+---+---+
A00 | [M] | * | * | *nr |
    +---+---+---+---+
A01 | *nf | [M] | * | *er |
    +---+---+---+---+
A02 | * | -t | *s | [M] |
    +---+---+---+---+
A03 | - | - | [M] | *nf |
    +---+---+---+---+

```

Figure 4.18. Example of an APDG match-matrix.

Similarity ratio

The similarity ratio is always shown as output of the APDG analysis. The ratio shows the similarity between two code-units as the percentage of matched nodes in all APDGs:

$$\text{Similarity ratio} = \frac{\text{Number of nodes in matched graphs from code unit 1}}{\text{Total number of nodes from graphs in code unit 2}} \cdot 100$$

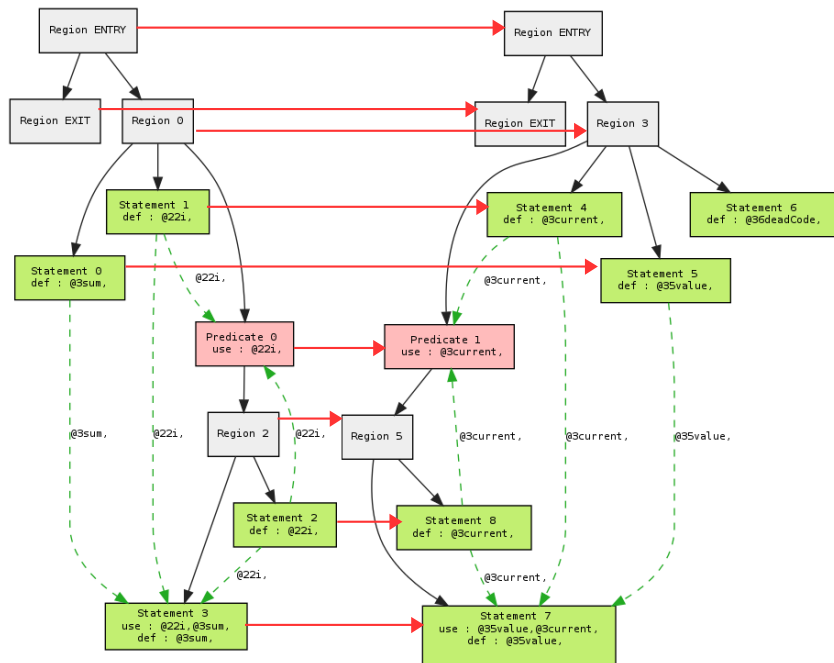
If all graphs in code unit 1 are isomorphic (exact) to any other graph in code unit 2, this equation will yield a 100% similarity ratio. If any graph in code unit 1 is not isomorphic, but e.g. sub-graph isomorphic to any other graph in code unit 2, the similarity ratio will be strictly less than 100%.

Time information

Whenever `PDG_SHOW_TIME_INFO` is specified in the configuration file, time information will be shown. The time information shows how long the construction and analysis took, both total and average per APDG. The time is automatically given in different magnitudes depending on the duration.

4.8.5 Sub-graph isomorphism example

Figure 4.19 shows an example of a sub-graph isomorphism matching between two APDGs where control replacement, identifier renaming, statement reordering and code insertion has been performed. In this example no exact isomorphism can be found since there is an extra node in the second APDG (Statement 6), generated by the declaration of `deadCode` (Line 5).



```

1 void function1 ()
2 {
3   int sum = 0;
4   for(int i = 0; i < 10; ++i)
5   {
6     sum += i;
7   }
8 }

```

```

1 void function2 ()
2 {
3   int current = 0;
4   int value = 0;
5   int deadCode = 0;
6   while(current < 10)
7   {
8     value += current;
9     current++;
10  }
11 }

```

Figure 4.19. Illustrative figure of a sub-graph isomorphism between two APDGs. Horizontal arrows shows the node-mappings between the graphs.

Chapter 5

Analysis

In this chapter we will describe methods used for the analysis of DETECT's APDG and AST based comparison. In the end we show the result of the analysis.

5.1 Method of analysis

There have been several studies on evaluation and comparison of clone detection tools. Such evaluations have however been very challenging to perform due to the diverse number of detection techniques, lack of standards in similarity definitions, absence of benchmarks, diversity of target languages and sensitivity to tuning parameters [2].

Roy and Cordy [19] attempt to compare clone-detection tools more uniformly, independent of tool availability and limitations of languages by using *scenario-based evaluation*. This evaluation technique is based upon a designed set of hypothetical editing scenarios which represent typical changes to copy/pasted code. We performed a quantitative analysis by means of comparing the results in such scenario-based evaluation between DETECT's APDG-analysis, DETECT's AST-analysis and another clone detection tool called *Moss* [20].

5.1.1 Scenarios

For each scenario we performed some cases of code modifications to a set of C or C++ procedures in a file. Table 5.1 lists the different scenarios used in the analysis. The table shows the short description, type and number of procedures affected by every scenario. Not all scenarios can be seen as a plagiarism scenario. Where *unsystematic* code modifications have been performed the semantics of the code has been changed as well. Description of the modifications performed in each scenario can be found in Table 5.2.

The benchmark files can be found at <http://www.ida.liu.se/~nikho42/detect/bench/>. The benchmark configuration file for DETECT’s APDG-analysis is also available in Appendix D.

Id	Short description	Type	# Procedures
1	Change indentation of code	FA	1
2	Change position of braces	FA	1
3	Remove unnecessary braces	FA	1
4	Add comments	FA	1
5	Remove comments	FA	1
6	Change comments	FA	1
7	Systematically replace variable names	IR	1
8	Unsystematically replace variable names	IR	1
9	Systematically replace function names	IR	4
10	Unsystematically replace function names	IR	4
11	Change types with typedefs	IR	9
12	Change order of function definitions	DR	4
13	Systematically change order of params	DR	5
14	Unsystematically change order of params	DR	5
15	Systematically change order of variable definition	SR	4
16	Unsystematically change order of variable definition	SR	4
17	Change the order of switch-cases	SR	1
18	Move variable declaration to outside of loop	SR	3
19	Insertion of statements	CI	4
20	Insertion of statements	CI	5
21	Insertion of control structure (execute once)	CI	4
22	Insertion of control structure (execute never)	CI	4
23	Insertion of constants and parentheses in expressions	CI	5
24	Negate predicate in if-statements	CR	1
25	Replacing for loops with equiv. while loops and vice versa	CR	4
26	Replacing do-while loops with equiv. and vice versa	CR	4
27	Unsystematically replace control-structures	CR	4
28	Control replacement with structured control statements (continue)	CR	1
29	Inlining functions	OM	4
30	Removal of unnecessary parenthesis, Exchanging '+=' for equivalent assignment	OM	4

Table 5.1. Scenarios used in analysis.

Id	Modifications
1	All whitespaces in the beginning of of each line were removed.
2	Indentation were changed from BSD to Allman style.
3	All unnecessary braces were removed.
4	2 lines of comments were added.
5	All comments were removed.
6	Multiline comments were switched with single line comments and vice versa.
7	A total of 10 Variables were renamed.
8	2 variables 'uses' were renamed.
9	Renamed functions to get more general names, such as 'function1', 'function2', ...
10	Functions with the same signature were renamed. Intraprocedural dependencies were not preserved.
11	Declared 9 typedefs to replace types used in the program.
12	All 4 function definitions were reordered.
13	A total of 9 parameters switched names. All functions preserved their semantics.
14	4 formal and 4 actual parameters switched names.
15	A total of 29 variable definitions were affected. The functions that were affected preserved their semantics.
16	A total of 29 variable definitions were affected. The functions that were affected did not preserve their semantics.
17	A total of 6 branches switched order.
18	A total of 7 variables were moved in and out of <code>for</code> loops, 4 functions were affected.
19	A total of 13 statements were added. All functions preserved their semantics.
20	A total of 10 statements were added. No functions preserved their semantics.
21, 22	7 and 10 control structures respectively were added in such a way that they did not modify the semantics of the code.
23	Constans were inserted into 10 expressions.
24	Predicate in <code>if</code> and <code>else if</code> statements were negated and the bodies of the structures were rearranged.
25	A total of 4 <code>for</code> and <code>while</code> loops were replaced. For this kind of replacement, variable declarations were moved into and out from the structure of the <code>for</code> loops.
26	A total of 4 <code>do while</code> , <code>while</code> and <code>for</code> loops were replaced.
27	A total of 4 loops were replaced. No functions preserved their semantics.
28	2 <code>continue</code> statements were inserted so that <code>else if</code> and <code>else</code> parts could be replaced with <code>if</code> statements.
29	Code from 4 functions were merged into one function.
30	+=,-=, *=, /= assignment operators were replaced with equivalent assignment statements. Parentheses were removed and added. All functions preserved their semantics.

Table 5.2. Description of scenario modifications.

5.2 Interpreting Results

To evaluate the output given by DETECT's APDG-analysis towards DETECT's AST-analysis and Moss for a given scenario, two types of metrics were evaluated.

- Similarity between the code units
- The number of detected procedures

The number of detected procedures and similarity ratio of DETECT's APDG analysis is given directly by the output of the system (as explained in Section 4.8.4). When interpreting the result from the AST-analysis and Moss, system-specific assessment methods had to be used.

AST-Analysis

The similarity ratio of DETECT's AST analysis was calculated by running it twice, once for how similar the first code-unit was to the second and then the opposite. For each run the average of each similarity ratio between each maximal matching pair was used to take the similarity ratio for that run. The average of the similarities between both runs was used to represent the similarity for the scenario.

If a modified procedure was found to have a similarity ratio of at least 95% towards its original procedure in any run, it was regarded to be detected.

Moss

The similarity ratio was given directly by the service. If a modified procedure was found to match its original with at least 95% of the lines of code it comprised of, it was regarded to be detected.

5.3 Test Results

5.3.1 All results

Table 5.3 shows both the number of detected procedures and average similarity ratio for the test scenarios.

5.3.2 Similarity ratio by type of modification

Tables 5.4 and 5.5 as well as the corresponding charts in Figures 5.1 and 5.2 show the average similarity ratio by type and nature of modification.

5.3.3 Detected procedures by type of modification

Tables 5.6 and 5.7 as well as the corresponding charts in Figures 5.3 and 5.4 show the total number of detected procedures by type and nature of modification. Figure 5.5 show a summary of the number of detected procedures.

Id	APDG Similarity	Detected procedures	AST Similarity	Detected procedures	Moss Similarity	Detected procedures
1	100.00%	1	100.00%	1	92.00%	1
2	100.00%	1	100.00%	1	92.00%	1
3	100.00%	1	100.00%	1	92.00%	1
4	100.00%	1	100.00%	1	92.00%	1
5	100.00%	1	100.00%	1	92.00%	1
6	100.00%	1	100.00%	1	92.00%	1
7	100.00%	1	100.00%	1	94.00%	1
8	0.00%	0	72.00%	0	99.00%	1
9	100.00%	4	100.00%	4	98.00%	4
10	75.36%	4	88.74%	2	87.50%	5
11	100.00%	9	99.79%	9	71.00%	5
12	100.00%	4	100.00%	4	95.00%	4
13	100.00%	5	98.14%	4	93.00%	5
14	58.49%	3	97.55%	3	98.00%	5
15	100.00%	4	93.12%	2	35.00%	0
16	20.29%	1	93.12%	2	48.00%	2
17	26.76%	1	98.57%	1	55.00%	1
18	69.49%	4	78.28%	0	20.50%	1
19	43.94%	3	84.13%	0	13.00%	0
20	9.46%	1	81.14%	0	40.00%	2
21	0.00%	0	84.00%	1	34.00%	0
22	67.53%	4	78.69%	1	59.50%	1
23	100.00%	5	98.75%	5	26.00%	2
24	100.00%	1	85.51%	0	0.00%	0
25	100.00%	4	88.65%	0	0.00%	0
26	22.50%	1	87.31%	0	0.00%	0
27	0.00%	0	87.83%	0	0.00%	0
28	90.91%	1	86.13%	0	55.00%	0
29	0.00%	0	67.87%	0	24.00%	0
30	100.00%	4	98.75%	3	47.50%	2

Table 5.3. Scenario results.

Type	APDG Analysis	AST Analysis	Moss
FA Procedures	100.00%	100.00%	92.00%
IR Procedures	100.00%	99.93%	92.00%
DR Procedures	100.00%	99.07%	94.00%
SR Procedures	65.42%	89.99%	36.83%
CI Procedures	52.87%	86.39%	33.13%
CR Procedures	78.35%	85.65%	13.75%
OM Procedures	50.00%	83.31%	35.75%

Table 5.4. Average similarities where no modification to procedure semantics had occurred.

Type	APDG Analysis	AST Analysis	Moss
IR Procedures	37.68%	80.37%	93.25%
DR Procedures	58.49%	97.55%	98.00%
SR Procedures	20.29%	93.12%	48.00%
CI Procedures	9.46%	81.14%	40.00%
CR Procedures	0.00%	87.83%	0.00%

Table 5.5. Average similarities where modification to procedure semantics had occurred.

Type	APDG Analysis	AST Analysis	Moss
FA Procedures	5	5	5
IR Procedures	13	13	9
DR Procedures	9	8	9
SR Procedures	9	3	2
CI Procedures	12	7	3
CR Procedures	7	0	0
OM Procedures	4	3	2
Total	59	39	30

Table 5.6. Detected procedures where no modification to procedure semantics had occurred.

Type	APDG Analysis	AST Analysis	Moss
IR Procedures	3	2	5
DR Procedures	3	3	5
SR Procedures	1	2	2
CI Procedures	1	0	2
CR Procedures	0	0	0
Total	8	7	14

Table 5.7. Detected procedures where modification to procedure semantics had occurred.

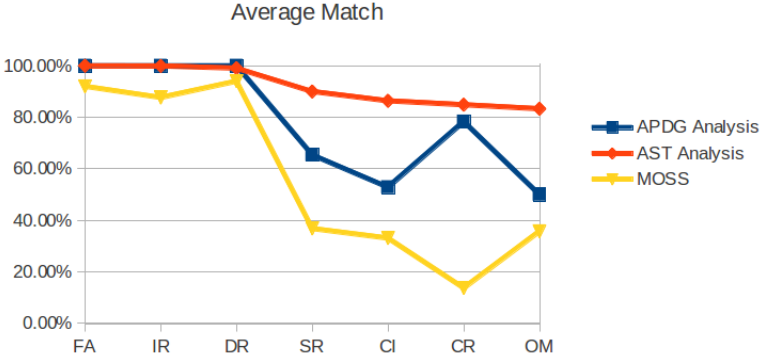


Figure 5.1. Chart over average similarities where no modification to procedure semantics had occurred, as given by Table 5.4.

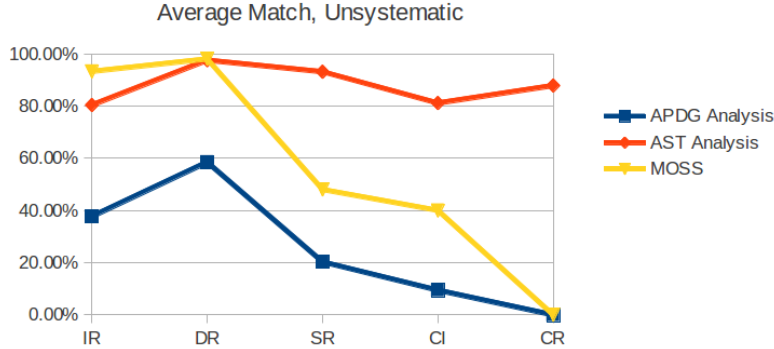


Figure 5.2. Chart over average similarities where modification to procedure semantics had occurred, as given by Table 5.5.

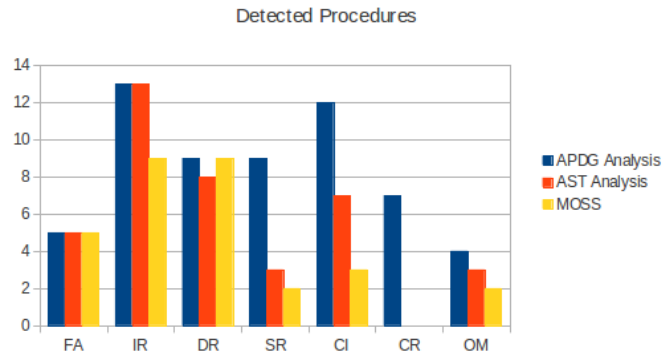


Figure 5.3. Chart over detected procedures where no modification to procedure semantics had occurred, as given by Table 5.6.

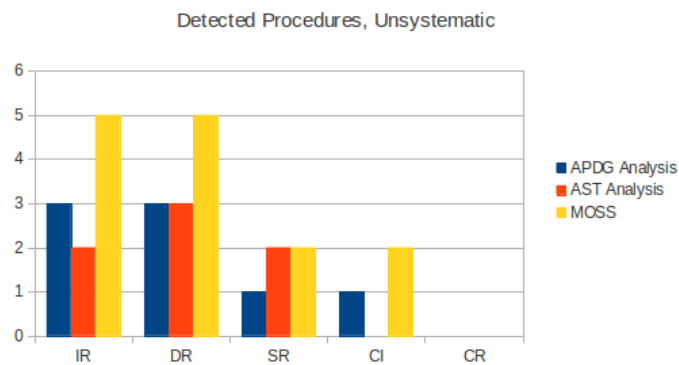


Figure 5.4. Chart over detected procedures where modification to procedure semantics had occurred, in Table 5.7.

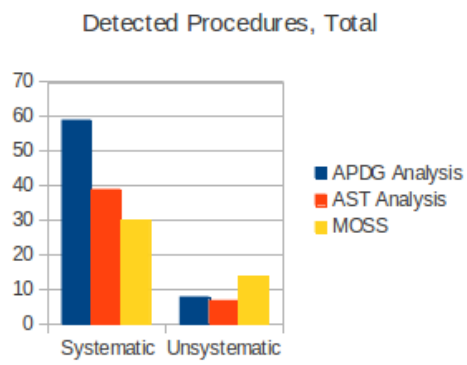


Figure 5.5. Summary of number of detected procedures by nature of modification.

Chapter 6

Discussion

In this chapter the test results are discussed.

6.1 Format Alteration Scenarios

All three systems detected all forms of format alteration. Since whitespaces or format information are discarded during parsing such alteration will not modify the ASTs or APDGs in any way. Moss had some problems discovering exact plagiarism, but still performed very well.

6.2 Identifier Renaming Scenarios

In these cases of systematic modifications, both the APDG and AST analysis performed well. Moss had some problems detecting plagiarism by means of using typedefs.

In the cases of unsystematic modification, the APDG and AST analysis performed equally well in terms of detected procedures, although the AST analysis had a higher average similarity ratio. For the APDG analysis, the detections of scenario 10 can be explained by the fact that the program dependencies stood the same while only interprocedural dependencies were changed. The APDG analysis does not calculate or make use of interprocedural dependencies. Moss performed poorly in both scenarios having a high average similarity and detecting all procedures.

6.3 Declaration Reordering Scenarios

In the cases of systematic modifications, both the APDG and AST analysis performed equally well both in terms of detected procedures and average similarity. Moss had a slightly lower average similarity but still detected all procedures.

In the cases of unsystematic code modifications, the APDG and AST analysis performed equally bad in terms of detected procedures. Moss performed poorly

and detected all procedures. Both the AST-analysis and Moss had high similarity ratio (97.5% and 98%).

6.4 Statement Reordering Scenarios

In the cases of systematic modifications, the APDG analysis performed well in cases where variable definitions were moved around, but poorly when branches of switch-statements were reordered. This can be expected from such forms of alteration since reordered branches will generate unique non-isomorphic APDG's. When variable declarations were moved into and out from loops, the APDG performed well in detecting some procedures. At closer look, the modifications that the APDG analysis has problems with is in cases when variable declarations were moved into a loop. This can be expected since it can change data-dependencies, sometimes in ways that can modify the semantics of the program. Although the APDG-analysis performed poorly in some separate cases, it overall detected more procedures than the AST-analysis and Moss. The AST-analysis had a high average similarity ratio but detected only 3 procedures. Moss performed poorly and had a similarity ratio below 60% in all scenarios, detecting only 2 procedures.

In the case of unsystematic modifications, the APDG-analysis performed well overall having a low average similarity ratio. It detected one procedure where the semantics had been changed. The AST-analysis performed well but had a higher average ratio than Moss and the APDG analysis. Moss had low average similarities in both unsystematic and systematic cases, which suggests that the system is more prone to miss these types of modifications.

6.5 Code Insertion Scenarios

In the cases of systematic modifications, the APDG analysis performed well overall. The APDG-analysis had a higher number of detected procedures than both the AST-analysis and Moss, but a slightly lower average similarity ratio than the AST-analysis. Moss performed least both in terms of average similarity and detected procedures.

In scenario 21, added control structures insert extra predicate and region nodes as well as control dependencies to the graph, which explains the poor result of APDG analysis. In the case of "careful" insertion of statements only one procedure was not matched. At closer examination this procedure was not matched due to the conservative stance on parameters use and definition at function call sites.

In the case of unsystematic modifications, the APDG-analysis performed well in most procedures, having the lowest similarity ratio of about 10%. One procedure was detected since DETECT's APDG-analysis does not separate between parameter and regular variable declarations. Moss detected two procedures but had still a low average similarity ratio. The AST-analysis had highest average similarity ratio but detected no procedures.

6.6 Control Replacement Scenarios

In the cases of systematic modifications, the APDG analysis performed well overall, yielding a high average similarity ratio and number of detected procedures. All modifications except in the cases where `do while` loops were replaced were detected. Replacing `do while` with e.g. `while` or `for` loops changes the order in which predicate nodes appear in the APDG, and isomorphism between the graphs will not be found. The AST-analysis had a fairly high similarity ratio but detected no cases. Moss performed very poorly, having none to low similarity ratio and detecting no cases.

In the case of unsystematic modifications, Moss and the APDG analysis performed best, yielding 0% similarity and no detected cases. The AST analysis had fairly high similarity ratio but detected no cases.

6.7 Other Modification Scenarios

In the case of inlining functions, all systems performed poorly. The APDG-analysis performed least, not managing to find any similarity. This type of optimization can not be detected properly by the APDG-analysis since macros and inlined functions will (at the compilers discretion) be inlined and not found in intermediate code forms.

In scenario 30, the APDG-analysis performed equally well or marginally better than the AST-analysis, Moss performed least having a low average similarity and detecting only half of the cases.

6.8 Overall

In the cases of systematic modifications, the APDG-analysis detected the most number of procedures but had a lower average similarity than the AST-analysis. Moss had both the least average similarity and number of detected cases.

In the cases of unsystematic modifications, the AST-analysis had the highest average ratio but equally many or slightly fewer detected procedures than the APDG-analysis. Moss had the highest number of detected procedures where unsystematic modifications had been performed.

Chapter 7

Conclusions

In this master thesis we have shown how to construct and analyse Approximate Program Dependence Graphs from Abstract Syntax Trees in a system called `DETECT` to find plagiarized procedures.

Our method in constructing the APDG's relies on using Abstract Syntax Trees with imposed interfaces to decouple the parser from the generation process. The construction of an APDG is done by first constructing the Approximate Control Dependence Subgraph with control flow information and then by calculating data dependencies to generate the Approximate Data Dependence Subgraph.

We have shown how to analyse the APDGs to detect plagiarism by testing for subgraph isomorphism or graph monomorphism between APDG pairs. To reduce the matchings to perform during analysis we have also used both distance and ratio-based pruning of the search space. Such pruning can avoid false positive matches and substantially minimize the time taken to find plagiarized APDG's in a large set of programs.

We performed an analysis of `DETECT` by comparing the results in a scenario-based evaluation against a popular and non-commercial system called `Moss`. The scenarios ranged from simple to more invasive forms of modifications to procedures in source code. The scenarios were categorized into both systematic and unsystematic forms of modifications to test detection quality both in terms of true and false positives.

The result of the analysis showed that the APDG-based system is equally effective as `Moss` and the AST-based system in detecting plagiarized procedures where more trivial disguises has been used, such as format alteration, identifier renaming and declaration reordering. It shows that the APDG analysis is more effective than both `Moss` and the AST based system in detecting plagiarized procedures where statement reordering, code insertion and control replacement has been used. All systems fall short where the code has been modified by other means.

The analysis also shows that the APDG-based method is not fully sufficient where more invasive forms of code modifications has been performed, such as some forms of reordering of statements, reordering of declarations and control replacements.

In terms of unsystematic modifications we have shown that Moss is more prone than DETECT to find similar code where the semantics of the code differ.

Chapter 8

Future Work

In this chapter we will discuss suggestions and ideas for future improvement or functionality of DETECT. These should be seen as an extension to those already given by Nilsson [17].

8.1 APDGs for Ada

More work is needed in parsing Ada code and in the step of transforming the Ada syntax trees for APDG analysis. At the current state, the generated ASTs do not fully conform to the interface required by the APDG-analysis. Without adequate conformance, APDG matching will not guarantee the detection of plagiarized procedures.

8.2 Serialization of APDGs

COJAC will be used for managing and batch processing of code hand-ins from programming students. Over time the amount of old hand-ins will accumulate and many comparisons between code-units will have to be performed. DETECT will at its current state become very inefficient in running large batches of program comparisons. APDG's are discarded after each invocation of DETECT and new APDGs will have to be generated for every new run. Serializing and saving APDG's for reuse would be of necessity for COJAC to become time effective in extensive APDG matching.

8.3 Improving the preciseness of APDGs

8.3.1 Pointers and aliases

The Abstract Syntax Tree used in DETECT does not give correct 'use' and 'define' information where pointers and references have been used. Investigating whether to perform some form of pointer analysis or to use other intermediate code forms for APDG generation can be the subject for further research.

8.3.2 Exception analysis

Currently, DETECT does not determine dependencies generated by exceptions. Jiang et al. [12] presents an approach to improve the preciseness of a PDG by using exception analysis. By promoting the traditional control-flow graph into an Improved Exception Control Flow Graph (IECFG) a more accurate program dependence graph can be computed. Implementing similar forms of analysis in DETECT is essential for calculating correct dependencies in programs using exceptions.

8.3.3 Call graphs

Since DETECT does not calculate or make use of dependencies between procedures, APDG's can match between programs that are semantically inequivalent and false-positive matches will arise. By also finding graph morphisms in graphs that represent static dependencies between functions, such as call graphs, might alleviate this. Figure 8.1 demonstrates this by showing an example of 2 programs that most likely are semantically inequivalent. In this case an APDG analysis would yield a similarity ratio of 100%.

<pre> 1 function_A: 2 call function_B; 3 4 function_B: 5 call function_C; 6 7 function_C: 8 call function_D; 9 call function_E;</pre>	<pre> 1 function_A: 2 call function_C; 3 4 function_B: 5 call function_C; 6 7 function_C: 8 call function_D; 9 call function_D;</pre>
--	--

Figure 8.1. Example of matching procedures with distinct call-graphs.

Another question to address is whether call-graphs can be used in conjunction with APDGs to detect e.g. subroutine extractions, or the opposite: inlining of functions.

8.4 Other types of APDG analysis

The APDG-module only finds exact sub-graph isomorphisms or monomorphisms. Using other forms of APDG analysis such as γ -isomorphism (relaxed isomorphism) as described by Liu et al. [16] can allow for more proper near matching of APDG's.

Bibliography

- [1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006. ISBN 0321486811.
- [2] Brenda S. Baker. Finding clones with dup: Analysis of an experiment. *IEEE Trans. Softw. Eng.*, 33(9):608–621, September 2007. ISSN 0098-5589.
- [3] Robert A. Ballance and Arthur B. Maccabe. *Program Dependence Graphs for the Rest of Us*. Issue 91, Part 10 of Technical Report, University of New Mexico, Dept. of Computer Science. URL <http://www.cs.unm.edu/>.
- [4] Ira D. Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant’Anna, and Lorraine Bier. Clone detection using abstract syntax trees. In *Proceedings of the International Conference on Software Maintenance, ICSM ’98*, pages 368–, 1998. ISBN 0-8186-8779-7.
- [5] Boost C++ Libraries. <http://www.boost.org/>.
- [6] Michel Chilowicz, Etienne Duris, and Gilles Roussel. Syntax tree fingerprinting for source code similarity detection. In *Program Comprehension, 2009. ICPC ’09. IEEE 17th International Conference on*, pages 243–247, may 2009. doi: 10.1109/ICPC.2009.5090050.
- [7] Luigi P. Cordella, Pasquale Foggia, Carlo Sansone, and Mario Vento. A (sub)graph isomorphism algorithm for matching large graphs. *IEEE Trans. Pattern Anal. Mach. Intell.*, 26(10):1367–1372, October 2004. ISSN 0162-8828.
- [8] Saumya K. Debray, William Evans, Robert Muth, and Bjorn De Sutter. Compiler techniques for code compaction. *ACM Trans. Program. Lang. Syst.*, 22(2):378–415, March 2000. ISSN 0164-0925.
- [9] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319–349, July 1987. ISSN 0164-0925.
- [10] Pasquale Foggia. *The VFLib Graph Matching Library, version 2.0*, March 2001. URL <http://www.cs.sunysb.edu/~algorithm/implement/vflib/implement.shtml>.

- [11] Mary Jean Harrold, Brian Malloy, and Gregg Rothermel. Efficient construction of program dependence graphs. *SIGSOFT Softw. Eng. Notes*, 18(3): 160–170, July 1993. ISSN 0163-5948.
- [12] Shujuan Jiang, Shengwu Zhou, Yuqin Shi, and Yuanpeng Jiang. Improving the preciseness of dependence analysis using exception analysis. In *Proceedings of the 15th International Conference on Computing*, CIC '06, pages 277–282, 2006. ISBN 0-7695-2708-6.
- [13] Gary A. Kildall. A unified approach to global program optimization. In *Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '73, pages 194–206, 1973.
- [14] Raghavan Komondoor and Susan Horwitz. Using slicing to identify duplication in source code. In *Proceedings of the 8th International Symposium on Static Analysis*, SAS '01, pages 40–56, 2001. ISBN 3-540-42314-1.
- [15] Jens Krinke. Identifying similar code with program dependence graphs. In *Proceedings of the Eighth Working Conference on Reverse Engineering (WCRE'01)*, WCRE '01, pages 301–, 2001. ISBN 0-7695-1303-4.
- [16] Chao Liu, Chen Chen, Jiawei Han, and Philip S. Yu. Gplag: detection of software plagiarism by program dependence graph analysis. In *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, KDD '06, pages 872–881, 2006. ISBN 1-59593-339-5.
- [17] Erik Nilsson. Abstract syntax tree analysis for plagiarism detection. Master's thesis, Linköping University, Department of Computer and Information Science, The Institute of Technology, 2012. ISRN: LIU-IDA/LITH-EX-A--12/043-SE.
- [18] Lutz Prechelt, Guido Malpohl, and Michael Philippsen. Finding plagiarisms among a set of programs with jplag. *J. UCS*, 8:1016–1038, 2000.
- [19] Chanchal K. Roy and James R. Cordy. Scenario-based comparison of clone detection techniques. In *Proceedings of the 2008 The 16th IEEE International Conference on Program Comprehension*, ICPC '08, pages 153–162, 2008. ISBN 978-0-7695-3176-2.
- [20] Saul Schleimer. Winnowing: Local algorithms for document fingerprinting. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data 2003*, pages 76–85. ACM Press, 2003.
- [21] Jeffrey R. Ullmann. An algorithm for subgraph isomorphism. *J. ACM*, 23(1): 31–42, January 1976. ISSN 0004-5411.
- [22] Michael Wise. String Similarity via Greedy String Tiling and Running Karp-Rabin Matching. Technical Report, University of Sydney, Department of Computer Science, 1993. URL http://luggage.bcs.uwa.edu.au/~michaelw/ftp/doc/RKR_GST.ps.

Appendix A

Requirements of DETECT

Each requirement has a priority. The system must implement requirements of priority 1. requirements of priority 2 and 3 may be implemented.

Id	Type	Description	Prio
1.	Documents	Documentation on how to use COJAC.	1
2.	System	COJAC can be run in a POSIX system.	1
3.	System	COJAC can define a group of input units in a configuration file and detect similarities within this group.	1
4.	System	COJAC can define in which ways the comparisons should be done in the configuration file.	2
5.	System	There is a GUI for modifying the configuration file for COJAC.	2
6.	System	There is a GUI for COJAC.	3
7.	System	COJAC uses a program DETECT, which uses one or several comparison algorithms to compare two units of source code at a time.	1
8.	System	DETECT is a command-line based program.	1
9.	System	DETECT takes two input units of code (of the same programming language) and compares them.	1
10.	System	Parameters passed to DETECT can be given on the command-line.	2
11.	System	Parameters passed to DETECT can be extracted from a configuration file given by COJAC.	1
12.	System	DETECT supports input units written in the language C.	1
13.	System	DETECT supports input units written in the language C++.	1
14.	System	DETECT supports input units written in the language Ada.	1
15.	System	DETECT supports input units written in the language Java.	2
16.	System	DETECT supports input units written in the language MatLab.	3
17.	System	DETECT integrates different front-ends for the supported languages.	1

Id	Type	Description	Prio
18.	System	It should be possible to integrate a new front-end with DETECT so that a new programming language can be supported.	1
19.	System	It should be possible to remove a front-end in DETECT.	1
20.	System	It should be possible to replace a front-end in DETECT so that a new standard of a language can be supported.	1
21.	System	It should be possible to integrate a new module with comparing algorithm(s) in DETECT	1
22.	Documents	There exists documentation on how to manage (adding and removing) the front-ends within DETECT.	1
23.	System	DETECT determines how many lines in the input files are textually identical.	1
24.	System	DETECT determines how many lines in the input files are identical without regarding whitespaces.	1
25.	System	DETECT reports which lines were textually identical.	1
26.	System	DETECT reports which lines were textually identical not regarding whitespaces.	1
27.	System	DETECT reports a metric representing the amount of similarity in code where Format Alteration, Identifier Renaming and Declaration Reordering has been applied.	1
28.	System	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.	2
29.	System	DETECT finds similarities in code which has been copied and modified by micro optimization.	3
30.	System	COJAC records data regarding submitters history of submissions and plagiarism metrics.	3
31.	System	COJAC flags submitters whose submissions have frequently high plagiarism metrics.	3

Appendix B

Interface for PDGFactory

```
1#ifndef PDG_FACTORY_H
2#define PDG_FACTORY_H
3
4/**
5 * @file
6 * Definition of the class PDGFactory and NestingStack
7 * @author Niklas J. Holma
8 *
9 * The PDGFactory class constructs a program dependence graph from a
10 * given AST interface specified in "AST_interface.h".
11 *
12 */
13
14/* Standard Includes */
15#include <string>
16#include <vector>
17#include <map>
18
19/* Local Includes */
20#include "AST_interface.h"
21#include "PDGExceptions.h"
22#include "PDG.h"
23#include "Config.h"
24#include "VarSet.h"
25
26/* External Includes */
27#include "boost/shared_ptr.hpp"
28
29namespace detect
30{
31
32 // The AST_interface operate with shared_ptr
33 typedef boost::shared_ptr<AST_interface> ASTptr;
34 // Predeclaration of NestingStack, comes after PDGFactory.
35 class NestingStack;
36
37 class PDGFactory
38 {
39 // A LabelMap maps label string to nodes in the graph
40 typedef std::map<std::string, PDG::Node> LabelMap;
41
42 // A GotoMap maps nodes in the graph to label strings
43 typedef std::map<PDG::Node, std::string> GotoMap;
44
45 public:
46 PDGFactory();
47
48 /**
49 * This method constructs the program dependence graph from an AST.
50 * It generates a new PDG and AnalysisGraph, and constructs them in
51 * three steps:
52 * 1) generate CDS by calling constructCDS()
53 * 2) generate DDS by calling constructDDS()
54 * 3) generate AnalysisGraph from the PDG via the
55 * PDG::constructAnalysisGraph() method.
56 *
57 * @param tree a boost shared_ptr to the AST tree
58 * @param dotFilename the filename to output dot-data
59 * @param configuration a pointer to the config instance
60 * @return a pointer to the created analysis graph
61 */

```

```

62     AnalysisGraph* constructPDG(const ASTptr& tree,
63                               const std::string& dotFilename,
64                               const Config* configuration);
65
66 protected:
67     // Represents a PDG node which does not exist
68     static const PDG::Node PDG_NON_EXIST = -1;
69     // Represents an AST node which does not exist
70     static const ASTptr AST_NON_EXIST; // Null ptr
71
72     /**
73      * Used to set the labels in the dot-output
74      * @return the next region node identifier as string
75      */
76     std::string getNextRegionId();
77
78     /**
79      * Used to set the labels in the dot-output
80      * @return the next predicate node identifier as string
81      */
82     std::string getNextPredicateId();
83
84     /**
85      * Used to set the labels in the dot-output
86      * @return the next statement node identifier as string
87      */
88     std::string getNextStatementId();
89
90     /**
91      * Starts the construction of the CDS, creating a PDG and in turns
92      * invokes descentCDS(). After this, takeCareOfGotosAndLabels() is
93      * called.
94      */
95     void constructCDS();
96
97     /**
98      * Recursively descends the AST to build the control dependence
99      * subgraph. The method in turns invokes other methods depending on
100     * the NodeControlType of the AST node.
101     *
102     * @param currentPdgNode The parent PDG node on which new nodes are
103     * to be added.
104     *
105     * @param currentAstNode The current AST node on which the function
106     * builds new PDG nodes from.
107     *
108     * @param breakNode the breakNode holds the backtracking PDG Node,
109     * this node is forwarded so that control dependence edges can be
110     * added for implicit control flow from break statements.
111     *
112     * @param continueNode the continueNode holds the previous loop
113     * region node. it is used whenever a 'continue' statement is
114     * found.
115     *
116     * @param relation the kind of relation attribute any new PDG node
117     * is set to have to its parent PDG node.
118     *
119     * @param nextStmt if the descentCDS is called on the last
120     * statement in a statement list, the nextStmt is set to the next
121     * statement in the program. This is used to add control flow edges
122     * from i.e. the last statement of a while loop, if loop, and so
123     * forth.
124     *
125     * @param depStack used to keep track of extra dependencies generated by
126     * conditional breaks and continue statements.
127     */
128     void descentCDS(const PDG::Node currentPdgNode,
129                   const ASTptr& currentAstNode,
130                   const PDG::Node breakNode,
131                   const PDG::Node continueNode,
132                   const std::string& relation,
133                   const PDG::Node nextStmt,
134                   NestingStack& depStack);
135
136     /**
137     * Decides wether a given AST node actually will generate a
138     * PDG node.
139     *
140     * @param statement the AST node to check
141     *
142     * @return true if the statement should generate a PDG node
143     */
144     bool isActualStatement(const ASTptr& statement) const;
145
146     /**
147     * Decides wether there are any gotos after an index in a statement list
148     *
149     * @param statementListNode the statement list node to check in
150     * @param index the index to check from
151     *
152     * @return true if the statement list contains any gotos after index

```

```

153     */
154     bool containsGotosAfterIndex(const ASTptr& statementListNode, int index) const;
155
156     /**
157     * Decides wether a node generates the last node in a statement
158     * list, given that some AST nodes do not generate statements.
159     *
160     * @param statementListNode the statement list node to check in
161     * @param index the index to check from
162     *
163     * @return true if there arent any actual statements after index
164     */
165     bool isLastStatement(const ASTptr& statementListNode, int index) const;
166
167     /**
168     * Decides wether a node generates the first node in a statement
169     * list, given that some AST nodes do not generate statements.
170     *
171     * @param statementListNode the AST statement list node (ptr) to check in
172     * @param index the index to check from
173     *
174     * @return true if there arent any actual statements before index
175     */
176     bool isFirstStatement(const ASTptr& statementListNode, int index) const;
177
178     /**
179     * Counts the number of actual statements of an AST statement list
180     * node.
181     *
182     * @param statementListNode the AST statement list node (ptr) to count
183     * actual statements in
184     *
185     * @return number of actual statements
186     */
187     unsigned numActualStatements(const ASTptr& statementListNode) const;
188
189     /**
190     * Decides wether a given AST statement list node contains any
191     * unstructured program statements that will generate extra
192     * dependencies.
193     *
194     * @param statementListNode the AST statement list node (ptr) to check in
195     *
196     * @return true wether the statement list contains any unstructured
197     * program statements.
198     */
199     bool generatesExtraDependencies(const ASTptr& statementListNode) const;
200
201     /**
202     * Decides wether a given AST node represents an unstructured
203     * control-flow statement, such as goto, break or continue.
204     *
205     * @param node the node (ptr) to check
206     *
207     * @return true wether the statement represents an unstructured cf-statement
208     */
209     bool modifiesControlFlow(const ASTptr& statement) const;
210
211     /**
212     * Transfers the use and definition sets from an AST node to a PDG
213     * node, will in turn call the pullUseDefs from the trees to pull
214     * use and def information from sub-expressions.
215     *
216     * @param pNode the PDG node to transfer to
217     * @param aNode the AST node to transfer from
218     */
219     */
220     void setUseDefs(PDG::Node pNode, const ASTptr& aNode);
221
222     /**
223     * Recursively calls itself and adds all uses and defs in the AST
224     * subtree to pRoot. This is useful since a statement can consist
225     * of several other sub-expressions.
226     *
227     * @param pRoot the PDG node to add to
228     * @param subtree the subtree to recursively descend and pull use and def
229     * information from
230     */
231     */
232     void pullUseDefs(PDG::Node pRoot, const ASTptr& subtree);
233
234     /**
235     * Creates a placeholder node for later use in the PDG.
236     *
237     * @param father the node that the placeholder will have control dependency
238     * to
239     * @param relation the label of the control dependence edge
240     * @param depStack the dependency stack containing extra dependency
241     * information
242     *
243     */

```

```

244     * @return the created placeholder PDG node id
245     */
246     PDG::Node createPlaceholder(PDG::Node father, std::string relation,
247                               NestingStack& depStack);
248
249     /**
250     * Creates a child to a given PDG node with the given information.
251     *
252     * @param father the node to which the newly created node should relate to
253     * @param label the label of the node, used for dot output
254     * @param relation the label of the edge, used for dot output
255     * @param eType the type of the edge
256     * @param nType the type of the node
257     *
258     * @return the newly created PDG node id
259     */
260     PDG::Node createChild(PDG::Node father, const std::string& label,
261                          const std::string& relation, PDG::EdgeType eType,
262                          PDG::NodeType nType, NestingStack& depStack);
263
264     /**
265     * Creates a new statement PDG node.
266     *
267     * @param fatherNode the parent node to relate to
268     * @param relation the label of the control dependency edge, used for dot
269     * output
270     * @param depStack the dependency stack containing extra dependency information
271     *
272     */
273     PDG::Node createStatement(PDG::Node fatherNode, const std::string& relation,
274                              NestingStack& depStack);
275
276     /**
277     * Creates a new region PDG node.
278     *
279     * @param fatherNode the parent node to relate to
280     * @param relation the label of the control dependency edge, used for dot
281     * output
282     * @param depStack the dependency stack containing extra dependency information
283     *
284     */
285     PDG::Node createRegion(PDG::Node fatherNode, const std::string& relation,
286                            NestingStack& depStack);
287
288     /**
289     * Creates a new predicate PDG node.
290     *
291     * @param fatherNode the parent node to relate to
292     * @param relation
293     * the label of the control dependency edge, used for dot output
294     * @param depStack the dependency stack containing extra dependency
295     * information
296     *
297     */
298     PDG::Node createPredicate(PDG::Node fatherNode, const std::string& relation,
299                              NestingStack& depStack);
300
301     /**
302     * Handles a STATEMENT_LIST ast node in the APDG generation. Will
303     * by default add a new region and add the statements of the AST
304     * node by calling addStatements().
305     *
306     * This method is virtual and might have to be overloaded depending on
307     * the type of the AST.
308     *
309     * params : same as descentCDS()
310     */
311     virtual void handleStatementListNode(const PDG::Node, const ASTptr&,
312                                         const PDG::Node, const PDG::Node,
313                                         const std::string&, const PDG::Node,
314                                         NestingStack&);
315
316     /**
317     * Iteratively goes through the statements of a STATEMENT_LIST ast
318     * node and generates statements from the statement list by
319     * invoking other methods depending on the control type.
320     *
321     * This method is virtual and might have to be overloaded depending on
322     * the type of the AST.
323     *
324     * params : same as descentCDS()
325     */
326     virtual void addStatements(const PDG::Node, const ASTptr&,
327                              const PDG::Node, const PDG::Node,
328                              const std::string&, const PDG::Node,
329                              NestingStack&);
330
331     /**
332     * Handles an AST node that contains use and def information. These
333     * are by default ast nodes with control type FUNCCALL, ASSIGNMENT
334     * and PROGRAM_EXIT.
335     */

```

```

335     * This method is virtual and might have to be overloaded depending
336     * on the type of the AST.
337     *
338     * params : same as descentCDS()
339     */
340     virtual void handleUseDefNode(const PDG::Node, const ASTptr&,
341     const PDG::Node, const PDG::Node,
342     const std::string&, const PDG::Node,
343     NestingStack&);
344
345     /**
346     * Handles a DETECT_LOOP ast node.
347     *
348     * This method is virtual and might have to be overloaded depending
349     * on the type of the AST.
350     *
351     * params : same as descentCDS()
352     */
353     virtual void handleDetectLoopNode(const PDG::Node, const ASTptr&,
354     const PDG::Node, const PDG::Node,
355     const std::string&, const PDG::Node,
356     NestingStack&);
357
358     /**
359     * Handles an IF_ELSE_BEGIN ast node.
360     *
361     * This method is virtual and might have to be overloaded depending
362     * on the type of the AST.
363     *
364     * params : same as descentCDS()
365     */
366     virtual void handleIfElseBeginNode(const PDG::Node, const ASTptr&,
367     const PDG::Node, const PDG::Node,
368     const std::string&, const PDG::Node,
369     NestingStack&);
370
371     /**
372     * Handles a CONTINUE ast node.
373     *
374     * This method is virtual and might have to be overloaded depending
375     * on the type of the AST.
376     *
377     * params : same as descentCDS()
378     */
379     virtual void handleContinueNode(const PDG::Node, const ASTptr&,
380     const PDG::Node, const PDG::Node,
381     const std::string&, const PDG::Node,
382     NestingStack&);
383
384     /**
385     * Handles a BREAK ast node.
386     *
387     * This method is virtual and might have to be overloaded depending
388     * on the type of the AST.
389     *
390     * params : same as descentCDS()
391     */
392     virtual void handleBreakNode(const PDG::Node, const ASTptr&,
393     const PDG::Node, const PDG::Node,
394     const std::string&, const PDG::Node,
395     NestingStack&);
396
397     /**
398     * Handles a GOTO ast node. Is by default already handles in
399     * addStatements().
400     *
401     * This method is virtual and might have to be overloaded depending
402     * on the type of the AST.
403     *
404     * params : same as descentCDS()
405     */
406     virtual void handleGotoNode(const PDG::Node, const ASTptr&,
407     const PDG::Node, const PDG::Node,
408     const std::string&, const PDG::Node,
409     NestingStack&);
410
411     /**
412     * Handles a LABEL ast node. Is by default already handled in
413     * addStatements().
414     *
415     * This method is virtual and might have to be overloaded depending
416     * on the type of the AST.
417     *
418     * params : same as descentCDS()
419     */
420     virtual void handleLabelNode(const PDG::Node, const ASTptr&,
421     const PDG::Node, const PDG::Node,
422     const std::string&, const PDG::Node,
423     NestingStack&);
424
425     /**
426     * Returns true wether a given AST pointer is not pointing to
427     * AST_NON_EXIST (null ptr) which represents that the AST node does
428     * not exist.
429     */

```

```

426 inline bool nodeExists(ASTptr p){ return p != AST_NON_EXIST; }
427
428 /**
429  * Returns true whether a given PDG is PDG_NON_EXIST (-1)
430  * which represents that the PDG node does not exist or have not yet
431  * been created.
432  */
433 inline bool nodeExists(PDG::Node n){ return n != PDG_NON_EXIST; }
434
435 /**
436  * Goes through all accumulated pdg nodes representing goto
437  * statements and adds correct control flow edges between them and
438  * the corresponding control flow target by using the goto and
439  * labelmaps.
440  */
441 void takeCareOfGotosAndLabels();
442
443 /**
444  * Adds extra control dependencies from a dependency stack to a PDG
445  * node, also adds an extra region to the graph to summarize these
446  * dependencies.
447  *
448  * @param to Node to add the extra dependencies to
449  * @param depStack the dependency stack containing (predicate)
450  * nodes that generate extra dependencies.
451  */
452 void giveExtraDependencies(PDG::Node to, NestingStack& depStack);
453
454 /**
455  * Constructs the Data Dependence Subgraph. Iteratively calls the
456  * descentDDS and does so as long as any OUT sets has been changed.
457  */
458 void constructDDS();
459
460 /**
461  * Recursively descends the PDG to generate data dependency edges
462  * using the USE and DEF information that each node have.
463  * At each step IN, OUT, GEN and KILL sets are calculated for
464  * the current node.
465  *
466  * @param node the PDG node to descend
467  * @param visits a set of previously visited nodes
468  * @param reaching OUT set of the previous node
469  */
470 void descentDDS(const PDG::Node node,
471               std::set<PDG::Node> visits,
472               VarSet reaching);
473
474 /* The lMap maps labels to node ids (strings to ids) */
475 LabelMap labelMap;
476
477 /* The gMap maps node ids to labels (ids to strings) */
478 GotoMap gotoMap;
479
480 /* The AST root pointer */
481 ASTptr root;
482
483 /* A pointer to the PDG instance, will be created and destroyed for
484  * each new PDG construction.
485  */
486 PDG* g;
487
488 /* A pointer to the configuration instance */
489 const Config* conf;
490
491 /* Used for DDS generation to tell whether any OUT set has changed */
492 bool continueDDS;
493
494 /* Used for DDS generation to tell how many times a nodes is
495  * allowed to be visited
496  */
497 unsigned maxDDSVisits;
498
499 private:
500 PDGFactory(const PDGFactory&);
501 PDGFactory& operator=(const PDGFactory&);
502 };
503
504 class NestingStack
505 {
506 public:
507 // The DependencyData type represents a node
508 typedef signed int DependencyData;
509
510 // The Stack type is a set of DependencyData
511 typedef std::set<DependencyData> Stack;
512
513 // For short-cutting the iterator type
514 typedef Stack::iterator iterator;
515
516 NestingStack();

```



```
517
518 // Copy constructor
519 NestingStack(NestingStack&);
520
521 /**
522  * Merges the dependency stack with another.
523  *
524  * @param other the dependency stack to merge with
525  */
526 void merge(NestingStack& other);
527
528 /**
529  * Pushes a PDG node to the dependency stack
530  *
531  * @param extra the PDG node to push
532  */
533 void pushExtraDependency(PDG::Node extra);
534
535 /**
536  * Checks whether the stack is empty
537  */
538 bool hasExtraDependencies() const;
539
540 // Clears the dependency stack.
541 void clear();
542
543 // Returns an iterator to the beginning of the dependency stack
544 iterator begin();
545
546 // Returns an iterator pointing to just past the end of the stack
547 iterator end();
548
549 private:
550     Stack stack;
551 };
552 }
553
554 #endif // PDGFactory_h
```

Appendixes/PDGFactory.h

Appendix C

Configuration file for benchmarking

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

# 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
```

```
## MONO | SUB-ISO | EXACT-ISO
PDG_GRAPH_MATCH_TYPE=MONO
PDG_SHOW_LEGEND=TRUE
PDG_MATRIX_OUTPUT=TRUE
PDG_DUMP_MAPPINGS=FALSE
PDG_VERBOSE_OUTPUT=FALSE
PDG_MATCH_TIMEOUT=10
PDG_NODE_FREQ_THRESHOLD=20
PDG_EDGE_FREQ_THRESHOLD=25
PDG_NODE_RATIO_THRESHOLD=40
PDG_EDGE_RATIO_THRESHOLD=70
PDG_EDGE_NUM_THRESHOLD=3
PDG_NODE_NUM_THRESHOLD=3
```

Appendix D

Default configuration file

```
AST_ALL_MATCHES_OUTPUT=TRUE
AST_DETAILED_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=MONO
PDG_MATCH_TIMEOUT=3
PDG_MATRIX_OUTPUT=TRUE
PDG_NODE_FREQ_THRESHOLD=5
PDG_EDGE_FREQ_THRESHOLD=5
PDG_EDGE_NUM_THRESHOLD=8
```

```
PDG_NODE_RATIO_THRESHOLD=10
PDG_NODE_NUM_THRESHOLD=5
PDG_SHOW_LEGEND=TRUE
PDG_SHOW_TIME_INFO=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
```


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/>

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/>