Detecting Co-Derivative Source Code – An Overview

Tommi Elo Tero Hasu
Helsinki University of Technology
Telecommunications Software and Multimedia Laboratory
{tommi.elo,tero.hasu}@hut.fi

Abstract

There is a body of literature concerning or related to the detection of source code texts that have the same origin. This paper presents a survey of such literature, and provides an overview of the topic.

KEYWORDS: source code analysis, plagiarism

1 Introduction

Within the last decade or so, a number of publications on detection of co-derivative source code texts have been published. This paper presents a survey of some the existing, public knowledge that is related to the topic.

By *co-derivative*, we mean texts that have the same origin [16], i.e. we call two texts co-derivative if they have both been created via modification of a common ancestor, or if one of the texts is the ancestor of the other. For discussion on exactly what we mean by the term *source code*, see Section 2.

It is not possible to determine with absolute certainty whether two texts are co-derivative simply by examining the texts themselves. Some information external to the texts would be required to do so. Therefore, when performing co-derivative detection with only source code as the input, we believe the best that can be done is to look for sufficiently long texts that have such similarities that there is reason to believe that the texts are co-derivative. In Section 3, we cover different factors that one could consider in determining similarity, and their usefulness in the context of co-derivative detection.

In Section 4, in turn, we move on to discuss the "durability" of the similarities, by which we mean the amount of work it takes to remove them, either unintentionally (as the source code evolves) or deliberately (perhaps to hide plagiarism). We list some of the things that one might do to deliberately alter the appearance of source code, and mention some of the tools available that can be applied for this purpose.

To make similarity quantitative, we must be able to measure it in some manner. To do so, it is necessary to choose a metric for the similarity of a pair of source code texts, and to find a way to measure the similarity in accordance to that metric. Section 5 presents a number of different approaches to measuring source code similarity.

To allow the similarity measurement methods themselves to remain simple, it may be necessary to in some way preprocess all the source code being compared. This is so that some (or ideally all) uninteresting information that might cause the chosen measurement method to fail to identify similar texts can be dropped. Section 6 discusses some transformations that could be performed prior to source code similarity measurement.

For some metrics, measuring merely involves calculating token frequencies for each source code file, and then comparing them; such comparisons are simple and efficient. However, when using "structural metrics", one is typically required to determine the longest common code sequences that appear in both of the compared bodies of source code. When the amount of source code being analyzed is large, it is important for the string matching algorithm being applied to the task to be reasonably efficient. Section 7 gives a summary of some of the string matching algorithms that have been presented in literature.

Having presented metrics for measuring source code similarity, as well as techniques and algorithms which assist in performing the actual measurements, we continue in Section 8 by describing and comparing some of the technologies that actually apply some of these ideas in practice.

Finally, we discuss possible future work in Section 9, and then conclude with Section 10 by summarizing what has been presented in the preceding sections.

2 Source Code

Software programs are almost invariably written and maintained in the so-called source code form, instead of directly constructing and modifying binaries suitable for execution in a particular environment. Before a program can be executed, its source code must be translated into a form suitable for execution; sometimes this is done transparently just prior to running a program, and sometimes explicitly using a program called a *compiler*.

The term source code is often defined quite loosely, to encompass binary code as well, but in the context of this paper, we shall define the source code of a software system as being a textual description of the computation performed by that software system, in a language designed to be readable by both humans and machines. We assume that such a language has a limited vocabulary of keywords and predefined identifiers (words with special meaning in the language), and a well-defined grammar and semantics. Natural languages are thus excluded from this definition.

We do not insist that source code would always have to be written by a human; indeed, there are many tools capable of generating or modifying source code according to the instructions provided by the user. Naturally, the utilization of such tools present challenges for the detection of co-derivatives.

2.1 Programming Languages

We shall define the term *programming language* as a language in which source code may be written. When looking for similarities in two source code files, the simplest approach would be to disregard the programming language and treat source code like any other text, and then, for instance, find those lines of text that differ with the UNIX `diff` command [10]. This way, one would not need to care about the language in which the source code has been written.

If, on the other hand, one does know the source code language and makes use of that knowledge in source code analysis, one can not only analyze the structure of the text, but also the structure and semantics of the program itself. If one wants to measure something other than the textual similarity of source code, it is necessary to take on the burden of explicitly adding support for each of the languages whose source code is to be analyzed.

Even if a tool already supports some programming languages, making it understand an additional one can still involve a lot of work, as there are considerable differences between languages. The amount of work required is greatly affected by the depth of the analysis required; parsing expressions of a particular language takes far more effort than recognizing strings that look like keywords in that language. It may be desirable for the sake of performance alone to avoid solutions that require the building of a full semantic analyzer for each supported language, especially when it comes to complex languages such as C++.

2.2 Compilers

When someone says he or she has implemented a programming language, it usually means that he or she has implemented a compiler or an *interpreter* for it (possibly among other things). As already mentioned, a compiler is a program that translates source code into a form suitable for execution in a particular runtime environment; we refer to code produced by a compiler as *object code*.

An interpreter, in turn, is a program that accepts source code as its input and executes the program described by the source code. Many interpreters do not execute the provided source code as is, but rather they first convert it into some internal form that is faster or easier to execute. Such an interpreter could thus be regarded as a combination of a compiler and a runtime environment. We shall, however, exclude such embedded compilers from our discussion in this section, and concentrate on standalone compilers that output object code and write the code into one or more files.

The process of converting source code into object code involves multiple phases, which may interleave to an extent. For instance, it is common to have the syntactic analysis phase “drive” the lexical analysis phase by having the lexical analyzer scan for more tokens only as they are required by the syntactic analyzer. Different compilers have different functionality, and therefore their compilation process may involve different sets of phases; the following set is typical:

Lexical analysis. In the lexical analysis (or scanning) phase, the compiler converts a sequence of characters into a sequence of lexical items, which are often called *tokens*. Any characters not of significance in later phases (e.g. those constituting comments or delimiting whitespace) can already be dropped at this stage of compilation simply by not including them in any of the generated tokens. Any character sequences that may not appear in the language should be caught.

Each token that does get generated is given a type; the type is determined based on the textual content of the token, and possibly also the context in which the token appears. The textual content of the token may be recorded in the token as is or after a conversion to some other form, or it may even be left out altogether; the choice of what to do depends on what information is required in the later phases.

Syntactic analysis. In the syntactic analysis (or parsing) phase, the token sequence generated during the scanning phase is analyzed in order to group tokens into grammatical phrases; token sequences that are not valid in the source language are naturally identified as well.

A parser typically produces a tree-like data structure as its output, containing a hierarchical representation of the parsed program. As languages commonly have constructs that may nest – say a conditional statement could contain other statements, with practically unlimited nesting – it is natural to build a tree instead of a sequence for storing the results of this phase. The data structure produced during syntactic analysis is often referred to as an *abstract syntax tree* (AST).

As with the tokens produced by a scanner, the nodes of an AST are also assigned a type. Any additional information associated with a node and required in later phases must also be recorded, usually in the data structures representing the nodes.

Semantic analysis. In the semantic analysis phase, the meaning of each phrase appearing in the program code is determined. This involves relating variable references to their definitions, as well as collecting type information. In statically typed languages, type checking is performed.

Intermediate representation generation. During this phase, an intermediate representation (IR) that is not tied to any particular source language or target machine architecture is produced [2]. This makes it possible to have multiple front ends and back ends in the same compiler, which in turn facilitates support for multiple languages and target platforms.

Optimization. During this phase, optimizing transformations are applied to the intermediate representation with the aim of making the code smaller and/or faster to execute.

Code emission. During this phase, the intermediate representation is translated into object code suitable for execution on a particular platform.

Linking. During the linking phase, one or more relocatable object code files are linked to form an executable. This phase is sometimes performed by a separate program called a *linker*, or at runtime by the execution environment.

Most compilers perform multiple *passes* over the program being translated into object code, changing the representation of the program in some way with each pass. The passes do not necessarily correspond to the phases listed above, as it is possible to have multiple phases in one pass, or to perform multiple passes in one phase.

2.3 Decompilers

A decompiler is a tool that attempts to reverse the transformation performed by a compiler, i.e. given object code produced by a compiler, the tool attempts to derive the original source code. In practice, at least some information always gets discarded during compilation, and thus it is reasonable to assume that source code will not stay the same if piped through a compiler and a decompiler. The level of success that a decompiler can attain depends on factors such as:

- The abstraction level of the source language; it is easier to transform object code into assembly code than into an object-oriented (OO) language.
- The abstraction level of the object code; if, for instance, the object code contains explicit information about the class structure that appeared in the original source code, it is far easier for a decompiler to deduce that information.
- The amount of symbolic information in the object code; any naming information in the object code should help increase the readability of the generated code, as descriptive identifier names are important for understanding code semantics.
- The level of optimizing transformations that were performed during compilation. Less optimized code would generally be easier to decompile, as any transformation performed during compilation is likely to result in further deviation from the source code.

2.4 Pretty Printers

There are tools that are similar to compilers in the sense that they accept source code as input, but instead of outputting executable code they produce output in the same language as that of the input, with the purpose of ensuring that the appearance of the code follows certain rules by making modifications as necessary. Terms such as *source code formatter*, *source code beautifier*, and *pretty printer* are all being used to refer to such tools, as there presently is no single, established term for that purpose. We shall favor the last of these terms.

Pretty printers are invariably designed to produce output that is computationally equivalent to the input, and they are typically used to achieve consistent formatting, perhaps because of a requirement to adhere to a coding standard of some sort, or simply to make code easier to read. Indeed, software engineers tend to spend much more time reading rather than writing code, and thus any solution that makes code faster to read and understand is likely to help save time.

Pretty printers tend to be configurable, to allow for different tastes in formatting. It is beyond the scope of this paper to conduct a survey on the available pretty-printing tools to determine exactly what code transformations they might make, but we have come across tools that do some or all of the following:

- addition and removal of whitespace
- modification and addition of comments
- changing the casing of keywords and identifiers (when the language is case insensitive)

3 Similarities in Source Code

Before considering how to measure source code similarity and to identify possible co-derivatives based on the measurement results, we first have to consider what kind of similarity there might be, and which similarities might indicate co-derivation. This section lists some aspects of source code that one may want to consider when looking for similarities, and the kinds of similarities that one might find when doing so. Naturally, we cannot cover everything, since between all conceivable programs and languages there are any number of possible similarities to consider.

The reader should note that not all of the similarities covered herein are applicable to all languages, due to considerable differences between programming languages.

3.1 Comments

Comment syntax. Does a C++ programmer like to use `/* comment */` or `// comment` style? If both, which comment syntax is being used for which purpose?

Comment placement. Where do the comments appear: before a statement; the same line after a statement; before a function declaration; after function arguments, and before function body; ...?

Natural language. Are the comments written in English? Is the writer using the language in a somehow unusual way? How many spelling mistakes and grammatical errors are there?

Formatting style. How are the `*` characters used in C comments? Is a “standard” template for function comments being used?

Tags. Are there any tags understood by an API documentation generator tool, and if so, which tool? Which of the supported tags are being used?

Content. What are the comments about? How much text is there?

Some similarity in comments may be due to the use of tools, as comments are sometimes created using code generators, along with skeleton code for the programmer to fill in. Also, some text editors provide functionality for formatting comments after they have been written; the use of such facilities can affect syntax, placement, and formatting style of comments. There are also pretty printers that can automatically generate comments with content derived from the context in which the comment appears; such content could for instance include tags understood by a documentation generator.

3.2 Spacing

In many languages, whitespace can be used relatively freely to format source code. For instance, in C whitespace is ignored everywhere apart from string literals, and there thus can be significant differences between C programmers in the way spacing is used.

Indentation. Are tabs only, spaces only, or both being used for indentation, and how many of them? Are they being used consistently?

“Tails” (whitespace at the end of a line). How large a portion of lines have tails? How long are the tails? In what context do tails typically appear?

Empty lines. How large a portion of the lines are empty? How many consecutive empty lines are there? Where do the empty lines typically appear?

Spaces as separators. Where are optional separators being used? E.g. is it `for(;;)` or `for (;;)`? Is it `1,2` or `1, 2`? Is more than one space being used as a separator? Are other whitespace characters being used for the same purpose?

Line breaking. Where are line breaks used? E.g. is there a line break after the opening bracket of each block? Is each function argument on its own line in function definitions? (In some languages line breaks function as statement separators, but if they are optional in the sense that e.g. a semicolon could be used instead, there may be differences in their use.)

Similarities in spacing can be introduced when using a pretty printer (see Section 2.4) or a text editor that can do indenting. Especially the latter case is common, as there are a number of text editors that in some way support consistent indentation of a number of languages; however, editors that enforce a particular indentation style are rare, so people using the same editor with the same settings can still produce differently indented code.

3.3 Identifiers

Name length. Is it `sid` or `ShowInformationDialog`?

Naming style. E.g. one of the following:

```
CapitalStyle
camelStyle
CONST_STYLE
ruby_style
_underscorePrefixStyle
iProperty
aArgument
LeavingFunctionL
```

Type reference style. Often there is more than one way to refer to the same type. E.g. is it `int* foo` or `int *foo`? Is it `char a[]` or `char[] a`?

Type modifier usage. How frequent is the use of "non-essential" type modifiers in declarations? For instance, in Java, `const` and `final` are seldom essential, and mostly just help in avoiding programming mistakes; `volatile` is almost never used, as `synchronized` is easier to understand and can be used for the same purpose.

3.4 Synonymous Expressions

Unconditional loops. Does the programmer use `for(;;)` or `while(true)` or something else?

Negative conditionals. Is it `if not a` or `unless a`?

Incrementing. In C, one can write `i++`, `i += 1`, `i = i + 1`, or even `i -= -1`, i.e. one can increment a number with many different operators.

Comparisons. `if a + 1 == b` or `if a == b - 1`?

Blocks. `loop do puts string end` or `loop {puts string}`?

Conditional assignment. Does the text say `p ||= q` instead of `p = q unless p`? Especially this kind of "advanced" usage is interesting, being uncommon.

3.5 Redundancy

Especially those programmers who are not intimately familiar with a language or the APIs being used are likely to write some unnecessary code, due to not knowing exactly what each statement does. It takes time to find that information from documentation, assuming that the documentation even exists. Thus, it may seem like a good idea to write all of the desired operations explicitly, even if that might result in some duplication of computation, and a negative effect on performance. Below are examples of typical redundancies that might result if the programmer chooses that approach.

- Redundant error checks, e.g.:

  ```
  Object o = new Object();
  if (o == null)                     // redundant
      throw new OutOfMemoryError(); // redundant
  ```

- Redundant initialization, e.g.:

  ```
  TBuf<10> buf;
  buf.Zero();           // redundant
  buf.Append(_L("foo"));
  DoStuff(buf);
  buf.Zero();           // redundant
  buf.Copy(anotherBuf);
  ```

- Duplication of functionality built into a statement, e.g.:

  ```
  if (p)        // redundant
      delete p;
  ```

- Duplication of checks built into an API, e.g.:

  ```
  if (timer.IsActive()) // redundant
      timer.Cancel();
  ```

The redundancy examples listed above are fairly typical, and as such, they may not be a strong indication of co-derivation. Redundancies that cannot be accounted for by lack of knowledge on part of the programmer would be more valuable as a co-derivation indicator. One example of such redundancy is below.

```
def print_array_elems(array)
  array.sort # has no effect as result not saved
  array.each {|elem| puts elem}
end
```

3.6 Error Tolerance

Some programmers care about error tolerance, while some do not pay much attention to it, and thus there can be considerable differences. Naturally, when differences are common, a lack of them implies similarity.

- Checking of error values, e.g.:

  ```
  if (resource.Open() == SUCCESS)
      resource.Use();
  ```

  instead of

  ```
  resource.Open();
  resource.Use();
  ```

- Robust error handling using a syntax-based exception mechanism, e.g.:

  ```
  foo.open
  begin
      foo.do_stuff
  ensure
      foo.close
  end
  ```

  instead of

  ```
  foo.open
  foo.do_stuff
  foo.close
  ```

- Robust error handling using a custom exception mechanism, e.g.:

  ```
  Foo* foo = new (ELeave) Foo;
  CleanupStack::PushL(foo);
  foo->ConstructL();
  CleanupStack::Pop();
  ```

  instead of

  ```
  Foo* foo = new (ELeave) Foo;
  foo->ConstructL();
  ```

3.7 Structure and Ordering

Programming languages have major differences when it comes to the selection of structural constructs available to the programmer when attempting to organize a program in a manner that helps maintenance. There are constructs that

- may not be used to store data nor describe any computation; they are static, may not be instantiated, and are only used to describe other structures; e.g. a Java **`interface`**
- are strictly data structures; may be used to store data, but may not describe any computation; e.g. a C **`struct`** [1]
- may both contain data and describe computation (typically operations on the data); e.g. a Java **`class`**
- purely describe computation, and may not be used to store data (except perhaps for the duration of the computation); e.g. a Java method

Regardless of the differences between such constructs, we shall simply refer to all of them as *structural elements*. We shall also consider a source code file as a structural element, and in some cases, a single structural element can consist of multiple files (as with a Java **`package`**).

When looking for structural similarities in two programs, we could for instance consider:

- the average length of a line (in characters)
- the average length of a structural element (in character, words, or lines)
- in an enclosing structural element, the average number of items such as: other structural elements, constant declarations, instance variable declarations, global (non-instance) variable declarations, or statements
- the ordering of items in each enclosing structural element: e.g. the ordering of modules in each file, etc. (it should be noted that in many languages it is necessary to declare identifiers before referring to them, and that dictates ordering to an extent)
- the amount of duplicated code; some people do a lot of copy pasting, and some do next to none; there is likely to be a correlation between the average structural element size and the amount of duplication, as *clones* (code duplicates) tend to require more text than references to other structural elements

[1] In some languages (such as LISP) computation and data are indistinguishable, as the computation implicitly results from the evaluation of data structures; there are no explicit statements commanding the underlying machine.

Highly dynamic languages such as Ruby may make it difficult to inspect class definitions, for instance, as the definitions may change at runtime. With such languages one might decide to only consider the syntactic structures appearing in source code, instead of entire definitions.

3.8 Similarities Not Indicating Co-Derivation

As we have discussed, there are many different kinds of similarity that one could attempt to measure, but it is important to note that not all similarity is an indication of co-derivation.

3.8.1 Code Describing Different Functionality

Suppose we have two source code texts that are similar in terms of style and layout. Further suppose that those texts have no commonalities in the data or computation expressed by them, even at the lowest possible level expressible in the language, and at least one of the texts contains no comments. Clearly those texts should not be considered co-derivative, unless we know otherwise. If there is no common functionality and no textual similarity between comments, then there is nothing to indicate that the two texts would have been derived from a common ancestor.

The above example is mostly theoretical in the sense that few programs are such that they have no common code or data at all. However, even in a more realistic scenario, it may be possible (at least for a human) to determine that two programs describe more or less different things, and thus, even if they look similar stylistically, there is no reason to assume co-derivation.

3.8.2 Generated Code

There are a number of tools that are capable of generating source code based on instructions provided by a programmer. Especially the use of the so-called project wizards is a popular way to start implementing a new software application, as it is common for an application framework to require even the most basic application to implement a large number of interfaces. In such cases a wizard can provide the programmer with an application that does everything the underlying system requires, and the programmer then simply needs to customize and add application-specific functionality to the generated code.

Typically, code generators are not general-purpose tools, but rather they generate code for a particular purpose. For instance, it could be that one uses one tool to generate a skeleton application, an another one to create a parser for the application configuration file, and a third one to generate a description of a data structure used by the application. Thus, the code generated by such a tool tends to be highly similar in terms of functionality, not to mention formatting and commenting. This would indicate co-derivation even though, strictly speaking, two source code texts are no co-derived just because they have been created with the same tool.

4 Similarity Reduction

If one possesses a copy of some existing source code, there are many different reasons why the copied code might eventually get modified. If and when that happens, we can no longer call that code an exact copy of the original; instead, we say that it has been derived from the original. Whenever derived code is modified, it is likely that the similarity between it and the original code gets reduced in the process. In this section, we discuss some of the reasons why source code might get modified, and the kinds of changes that could be considered typical for each of those scenarios. This will hopefully give some idea as to what kinds of differences one should expect co-derivatives to have.

4.1 Software Evolution

The most common way for source code changes to get introduced is via software evolution. Such evolution takes place when a system is implemented, and later as its source code gets revised by programmers due to defect fixes or changing requirements. Some of the most common activities performed as software evolves are given below, and their effect on source code similarity is discussed in the corresponding section.

Activity	**Typical source code modifications**	**Section**
feature set modification	large chunks of code added or removed	4.1.1
defect fixing	small amounts of code altered	4.1.1
refactoring	some code rewritten, large amounts reorganized	4.1.2
source code tidying	formatting changes, comments added and updated	4.1.3

4.1.1 Alteration of Functionality

Most changes made to source code have the aim of in some way changing the functionality of the software. The reason for wanting such a change could be either that the software does not work according to its specification, or that the specification has changed.

Source code modifications made for the sole purpose of altering functionality typically involve the addition or removal of code sections (implementing functionality being added or removed), and small modifications here and there (references to the aforementioned code sections, parameter changes, etc.). In most cases most of the source code will remain exactly the same as in the previous revision, and indeed, many version control systems take advantage of this property by merely storing the differences between revisions, and not all of the program text of each revision.

Changes in source code formatting are typically not made, or are fairly local, just like the functionality changes. Many programmers use `diff`-like tools to see what changes have been made between revisions that actually affect functionality, and as wide-scale formatting changes would make that more difficult, programmers are effectively discouraged from making such changes. This is especially true when a version control system is being utilized throughout a project, as that ensures that the revision one wants to compare against is available.

Thus, to match different revisions of functionally evolving source code texts, it may well be enough to use a tool for finding textual differences, and then to compare the amount of code that has changed against the amount that has not; this approach should be quite effective in determining the extent of similarity between two different revisions of a file.

4.1.2 Reorganization

Software evolution typically also involves an increasing feature set, and this can easily lead to increased complexity of program code, especially if the requirement for the added features was not anticipated when the system was designed. When complexity makes software prohibitively difficult to maintain, the only alternatives – apart from a complete reimplementation – are to either to abandon maintenance, or to refactor the program code. The purpose of refactoring is to make the program structure easier to understand, and it involves adding, removing, splitting and merging of programming interfaces, and organizing the existing code to reflect the new interfaces.

As refactoring tends to involve code being moved around a lot, possibly to different files, to a tool such as `diff` the changes can appear far more extensive than they actually are, in terms of how much of the original code is still there. For this reason, file-based detection of shortest edit distances using insert and delete operations is not particularly effective for assessing similarity, unlike we found in Section 4.1.1.

Structural metrics (cf. Section 5.4) should fare better, especially if they can detect similar code across files. However, sometimes refactoring can have such a profound effect that unless one finds a way to detect similarity in functionality, it may not be possible to notice enough similarity to identify refactored code as co-derivative. Fortunately, as refactoring is time consuming, especially larger systems tend to be refactored a small portion of the code at a time, and for some code refactoring may never be necessary.

4.1.3 Tidying

For as long as a software product is released as binary only, and its source code is accessible in-house only, one may not be too concerned about the appearance of the source code. However, if a decision is made to release in source form, one may wish to ensure that the code looks professional, and some tidying up of the code may be in order. This often includes changes in indentation and formatting to give the code a consistent look and to improve its readability.

The commenting of source code is particularly important if the code is to be used or modified by others, and is typically given considerable improvements before first release. It is also common to generate some or all of the API documentation based on source code alone, and this is when comments become particularly important, and may be required to contain custom directives for the documentation generator.

When not preparing for a source release, it is probably rather uncommon to make extensive superficial changes to source code. Still, it does happen that the coding style of a program is modified just to comply with company coding standards or the personal preferences of the programmer. The likelihood of something like this happening is greatly affected by the personality of the programmer.

Changes in formatting of code could have a very significant impact on similarity measures based on the textual equality of lines, for instance. This is because it is particularly effortless to make such changes in a manner that affects a large number of lines in the program text. Many text editors have built-in support for formatting code in a variety of programming languages, and standalone pretty printers are available as well. Some similarity measures address this problem by ignoring some or all whitespace characters, and that is indeed effective, as most formatting tools do not add or remove anything but whitespace. However, care should be taken here, as whitespace functions as a separator in most programming languages (Fortran being perhaps the most well-known exception), and indiscriminate removal of all of it could make subsequent analysis more difficult.

As comments are more or less free-form text, relatively free of any restrictions imposed by the source language (the language in which the source code is written), they tend to contain more information than programming language expressions of the same length. A number of long comments with the same text in two source code files could be considered strongly indicative that those two files are co-derivative. However, with a similarity metric based on textual comparison of comment texts alone, similarity measures could be significantly affected by superficial program text changes. For this reason, some similarity measures ignore them [3, 26], despite of their potentially high value as similarity indicators.

4.2 Porting

The term *porting* refers to the act of translating software so that it runs on a different (physical or virtual) machine. This might involve one or more of the following activities:

Porting across machine architectures. Especially when software has been written in a low-level language (such as assembly), or when the software interfaces more or less directly with hardware (as in the case of physical device drivers), it may be necessary to switch to using a different instruction set, make changes in memory management, or otherwise account for differences in the underlying machine architecture.

Virtual machines are sometimes utilized to protect from porting work associated with hardware changes, as they tend to hide the specifics of the underlying machine more or less completely.

Porting across operating systems. Operating systems may have differences in filesystem organization, for instance, and porting software onto a different system could thus mean that the software must be changed to look for files from different locations, interpret the contents of files differently, use a different set of executables, communicate with system services with different protocols, etc.

More generally, ported software may need to make use of the system facilities differently to acquire the same set of functionality as on the source OS. Sometimes the required facilities may not even exist on the target platform, and in such a situation additional implementation work may be required.

Porting across APIs. Sometimes it is necessary to port software for an environment that does not have all the APIs referred to in the source code. In such cases, one may wish to replace references to any missing API with references to another API that is available and also provides the required functionality.

Porting across APIs is always required when porting across languages, and often required when porting between operating systems. In extreme cases, none of the APIs used in a program exist for the target OS. For instance, the APIs shipped with the Symbian Platform are rather unique – apart from a partial implementation of the standard C library, there are few if any APIs that are available on any other platform.

Porting to another GUI toolkit tends to be particularly laborious, as in a typical GUI application most of the code is GUI-related; thus similarity between a ported application and the original one could be severely reduced in the process of switching between GUI toolkits. Luckily, developers also try to avoid such major tasks, for instance by not using the "native" toolkit directly, and instead using a platform-agnostic GUI API that has been implemented for multiple platforms. Trolltech's Qt is a good example of such a toolkit, as it has been implemented for all of the most popular desktop platforms, and allows GUIs to be constructed for all those platforms from the same codebase.

Porting across languages. Translating a program into another language may be required in situations when no compiler or interpreter for the language is available for the target platform. This situation typically arises when porting between operating systems, especially if the target OS is relatively new. There are also cases when one simply wants to switch languages, as the target language is considered a better tool for the job. For instance, the text processing capabilities of Ruby are far superior to those of C.

A port into a new language requires considerable rewriting, and is often written from scratch, especially if the languages are not syntactically similar. Thus, while it may be difficult to detect co-derivation as differences between languages increase, at the same time the likelihood of co-derivation should decrease.

> Even when porting into a language with a completely different syntax, it can be helpful to base the port on existing source code; some advocate keeping the old code in comments and writing the new code next to it; if there are problems, one can check whether the new code actually corresponds to the functionality specified in the old code. However, if the languages have been designed for an altogether different programming paradigm, the structure of the port is likely to be so different that the above approach is not sensible.

A ported piece of software is called a *port*. Ports are typically produced manually by a software developer. While there are some tools that assist in porting between languages and GUI toolkits, for instance, such tools have typically been designed for select few porting tasks, and are useless for a developer faced with a task that none of the available tools support.

When not porting across languages, it may be feasible to keep the same codebase for all targets, i.e. to keep support for the source environment while adding support for the target environment. This may take some arranging if there is no conditional compilation support builtin in the language, but one can always use a separate preprocessor to attain conditional compilation. Another alternative is to make do without conditional compilation within files, and to separate platform-specific parts from platform-agnostic parts on the file level, having the build system choose which source files to use for which target. In any case, if the original code is kept relatively intact when porting, co-derivative detection is likely to be easier.

4.3 Plagiarism Hiding

We define *software plagiarism* as an act of taking existing source code, and then reusing and passing off that code as one's own, either in modified or unmodified form, without crediting the original author. We further assume that the plagiarized code does not come with a license that permits uncredited reuse.

The above definition is fairly narrow, and does not account for the purporting of ideas, algorithms, or information from existing source code. However, as we are focused on co-derivative detection, it makes sense to exclude anything but direct reuse of source code from our definition.

An attempt to hide software plagiarism differs from software evolution in the sense that in the former case, similarities are removed intentionally, with the aim of making it difficult to notice that plagiarism has taken place. The reason for plagiarism would typically be one or more of those listed below, with perhaps the general theme being the desire to do less work than would be required without plagiarism:

1. The plagiarist does not understand the programming system, and does not want to spend the time and effort required to learn it.

2. The plagiarist is not familiar with the problem he/she should solve with the software program, or does not know how to solve it, and does not want to spend the time and effort required to come up with a solution.

3. The plagiarist does not want to spend the time and effort required to write the program.

4. The plagiarist feels that it would be stupid or uninteresting to reinvent the wheel, i.e. to solve a problem that someone else already has. (It may even be that the task of covering up the plagiarism seems like a more interesting challenge.)

Of course, it could simply be that a plagiarist does not know it is not okay to take someone else's code and reuse it without crediting the source, or is mistaken about the what the license of the software allows, but we will not consider such cases here, as plagiarism hiding then probably will not take place.

When looking at the reasons for plagiarism listed above, it becomes apparent that in all but case 4 the plagiarist probably does not understand the plagiarized program completely, and does not want to take the time to gain that understanding. This means that any plagiarism hiding is likely to be done so that little understanding of the workings of the program is required, and significant changes affecting the computation would be risky without that understanding.

Thus, typically a plagiarist would modify the program text in a manner that makes it look different from the original, while introducing little or no change in the computation itself. Such modifications can, to an extent, be made in an automated manner by utilizing translation tools or editor functionality, and could for instance involve modifying the formatting of the program text, removing comments, renaming local identifiers, or replacing expressions with synonymous ones.

The authors of [26] examined a number of programs, some of which were either actual plagiarisms by students, or explicitly made plagiarisms. According to their findings, the most common attacks used by plagiarists would appear to be:

- modification of code formatting
- insertion, modification, or deletion of comments
- moving subexpressions into new auxiliary variables, or vice versa
- inlining of small methods, or moving parts of existing methods to new ones
- reordering of non-interdependent statements
- exploiting mathematical identities (e.g. by replacing $x + 1$ with $1 + x$, or $\tan \alpha$ with $\frac{\sin \alpha}{\cos \alpha}$)

- voluntarily introducing defects by removing or adding additional statements, or modifying constants
- adding or removing unused code

The authors of [23] have done a similar study, and they in turn believe the following attacks to be common among plagiarizing students:

- changing the layout of the program
- changing the commenting of the program
- changing the names of identifiers
- replacing defined constants with literals and vice versa
- changing the definition order of constants, types and variables
- changing the order of procedures and functions
- adding or removing compound statements, empty statements and parentheses
- changing the values of literals and defined constants
- changing the number and contents of output statements

Changes in formatting and commenting would seem particularly popular, perhaps due to being safe in the sense that in most languages (barring exceptions such as Python) they do not affect computation in any way. If we also consider the fact that pretty-printing tools are available for a number of languages, and can adjust the formatting of code and comments automatically throughout a program, we can conclude that the style and layout of programs are particularly vulnerable. It is quite reasonable to assume that anyone worried about getting caught of plagiarism would take the time to at least run the plagiarized program through a pretty printer.

Let us now consider the two extremes of attitude that we might find among plagiarists:

A careless plagiarist might well rely on no one finding the original program to compare against, thus trying not to cover up his plagiarism at all.

In such a case the plagiarism should be trivial to detect when comparing against the original, unless the plagiarized version had evolved considerably since the copying took place.

A super-careful plagiarist would spend almost as much time (or even more) hiding the plagiarism as it would take to reimplement the whole program from scratch.

In this case it might not be possible to detect any indication of plagiarism, but perhaps there comes a point when there is so little of the original code left that no copyright infringement should any longer be considered to be taking place.

4.4 Obfuscation

A lot of software contains intellectual property that is considered valuable, whether it is the code itself, or some information (such as cryptographic keys) embedded within it. Even if the software is in binary form, the intellectual property could get compromised via *reverse engineering*. One method of attempting to protect intellectual property contained in a software program is to perform *code obfuscation*, which means modification of the program in a way that makes the program harder to understand, and thus also harder to reverse engineer. The goal of obfuscation is typically to maximize obscurity without causing a significant detrimental effect on execution time.

There are a number of obfuscators that do nothing more than scramble the identifiers used in a program [7]. However, in [9], Collberg et al suggest transformations that alter the control flow of a program, and in [8] they present transformations for obscuring data structures. A combination of such transformations made throughout a program could potentially make it very difficult to identify the obscured version as a co-derivative of the original, as it could be that very little – apart from the high-level semantics of the program – would have been preserved in the transformation. It is difficult to detect such similarity using an automated tool, and even if that was possible, any similarity found could merely imply that the programs have been implemented to the same specification – claiming co-derivation would be hard to justify.

Fortunately, due to the difficulty of comprehending obfuscated programs, no sensible developer would attempt to maintain a software product in an obfuscated form (obfuscation is typically only done before each "binary" release; indeed, many obfuscators only accept binary code as input). Therefore, the developer of the product should also have an unobfuscated version, which should be used if one wishes to perform co-derivative detection. The story could be different if there were "obfuscators" that modified code in a manner that did not make it more difficult to understand. However, to our knowledge, such "obfuscation-hiding obfuscators" do not presently exist.

5 Measuring Similarity

Software similarity measurement can be done on many different levels. In this section we go through methods that evaluate similarity on the file level, and methods which understand the syntax and semantics of the source code they are evaluating.

5.1 Similarity Measure Properties

We call a similarity measure *ideal*, when we get the highest possible score when we match a document with itself [16].

Due to the nature of the surveyed systems, they operate on a set of programs, pairwisely checking whether either element of a candidate pair is too similar to the other and thus considered a plagiarism. If this is the case, the pair is then called a plagiarism pair. In order to assess the soundness and completeness of a plagiarism detection system, we need concepts which can describe how well a similarity measure is functioning. *Precision* and *recall* are widely used metrics for information retrieval systems, and for a given data set and similarity measure, we could apply these in evaluating the similarity measure [16].

Let us define the total number of programs in the test set to be n, the number of plagiarism pairs in that set g, and the number of programs marked as plagiarized by our system m. Out of m marked pairs, we have t true plagiarism pairs. Then *precision* is defined as $100 * t/m$. It is thus the percentage of correctly positively marked plagiarism pairs of all positively marked pairs. *Recall* is defined to be $100 * t/g$, thus percentage of correctly positively marked pairs out of all plagiarism pairs.

In fact, the above measures are more widely known in the field of statistical testing as *positive predictive value* and *sensitivity* (of a test in question), respectively. Notice that the use of *sensitivity* as an appropriate measure for the soundness of a similarity measure is appropriate only if the prevalence of plagiarism is high enough in the target group.

A plagiarism detection system is said to achieve *perfect discrimination* when the lowest similarity value among plagiarism pairs is higher than the highest among non-plagiarism pairs [26]. This also means system has both perfect recall and perfect precision.

If we want an *absolute* similarity score, the result must be normalized to a value between 0 and 1, where 1 means exactly the same, and 0 means no similarity [16]. [26] points out that we can also define a similarity measure to be 1 when a candidate pair is not identical, but the other completely includes the other. All of the systems in the literature we have studied use the former definition.

5.1.1 Goal-Oriented Classification

We should also classify methods that measure software similarity based on their accuracy in different situations. Some methods may be good at screening for potential plagiarism but give inadequate assurance of the result. This kind of methods can be called *screening methods*. Other methods may be give high assurance for positive results - minimizing the risk of blaming the innocent - but perform slowly or miss some positive results altogether. These methods can be called *assurance methods*. Depending on the motivation for plagiarism detection, the method of choice may vary considerably.

5.2 Fingerprinting

Fingerprinting aims to produce a compact description, a *fingerprint*, of each file to be compared. A fingerprint can be produced by selecting substrings from a file and running a mathematical function on them. The function, typically called a hash function, produces output, which is in the context of file fingerprinting referred to as *minutia* [16]. All the generated minutiae for a particular document together form a document fingerprint.

There are number of properties we would like a fingerprinting system to have [16]. First, a fingerprint needs to be reproducible; that is, every time a given input is processed, the same result must be obtained. Second, the fingerprint generating function should produce a uniform distribution of output, and third, output should lie between two bounds. Fourth, it should be rare that two minutiae share the same output. Fifth, the function should be fast.

The reason for wanting to create fingerprints instead of directly comparing the documents is efficiency. In the context of fingerprinting, we want to maximize the performance of comparing one document, called a query document, against n files. This can be done by storing only the fingerprints of the n files to the database instead of the files themselves.

One thing to note is that the runtime of many of the plagiarism detection systems presented in Section 8 increases quadratically with the number of programs in the program set. This is because all programs in the program set are compared against every other program in the set. However, with fingerprinting, things increase linearly with the size of the database.

5.2.1 File Equality with Fingerprinting

Suppose we are looking for similarities from two large software systems, each consisting of hundreds or thousands of source code files, and are only interested in whether we find the same non-empty file in both sets. Say we have M files in the first set, and N in the second. In a naive solution, we have to make MN comparisons of file contents, which could take too long to be practical. However, we can query the filesystem for file sizes, and use hash values to avoid reading the content of any file more than once. This way we can reduce the effective complexity from $O(MN)$ to $O(M + N)$, if we assume that anything not involving content reading is negligible in terms of the overall performance.

In other words, we can get better performance when comparing files for equality by using fingerprinting over the whole file, see Section 5.2. One method for generating a fingerprint for file equality purposes is to hash a whole document into just one minutia. Obviously, the length of the hash must then be sufficient in order to not generate too many collisions. Another way is to employ an all substrings selection strategy, where all non-overlapping substrings are calculated.

5.2.2 File Similarity with Fingerprinting

Substrings to be hashed can be selected based on statistical properties or relevant structural information. The problem is how to select the relevant parts, and the same problem applies for source code specific attribute-counting systems, which are discussed in the next section.

In short, fingerprinting can considerably speed up the process of comparing large sets of source code files, if we can calculate the fingerprints to the database beforehand. The accuracy of the fingerprinting system depends on which parts of the files are selected, and how similarity is quantified.

5.3 Attribute-Counting Systems

Attribute-counting systems are such that they compute for each program n different software metrics, so that each program is mapped to a point in an n-dimensional cartesian space. The systems then consider sets of programs that lie close to each other to be possible plagiarisms. [26]

The earliest plagiarism detectors fall into this category. The very first attribute-counting metric known to us was Halstead's *software science metrics* [14], and others followed, with a larger number of attributes. However, regardless of the number of attributes, summing up a metric across the whole program throws away so much structural information that structure-based metrics tend to be superior.

5.4 Structure-Based Systems

There are several approaches for performing comparisons that are capable of pinpointing those parts of two texts that are likely to have the same origin, based on program structure comparison. In this section we present some of them.

[28] compares structure and attribute-counting systems. Its conclusion is that the attribute-based metrics are typically better with entire files, but fall short when only a part of a file has been copied.

5.4.1 Token Sequence Comparison

Roughly, a token sequence based similarity comparison method is done in the following way.

1. Convert text to a token sequence (see Section 6.3)
2. Compare token sequences for similarity, for each sequence pair, using a string matching algorithm (see Section 7)

In addition to the string matching algorithm and its parameters, the token set we want to generate affects the results obtained. After string matching we need to decide, which percentage of similarity implies plagiarism (either in a screening sense or in an assurance sense). This value is called a cutoff-threshold.

Examples of systems utilizing this structural approach are: YAP3 [31], JPlag [26].

5.4.2 Token Tree Comparison

A token tree based similarity comparison method consists roughly of the following tasks.

1. Convert text to a token tree
2. Compare token trees to determine differences, using a generalization of a string matching algorithm

A token tree is an abstract syntax tree, which is a result of a parsing phase. See Section 2.2 for detailed discussion on compiler phases.

One of the systems using token tree comparison is described in [32]. It uses a tree variant of an LCS algorithm for quantifying the similarity of the token trees.

5.4.3 Syntax Graph Comparison

There are some papers that talk about comparing syntax trees based on graph algorithms, which attempt to detect the same or similar graphs, and do not do any sort of tokenization.

The problem here is what kind of a graph to build – what nodes, attributes, arcs, and so forth to include in it. There appears not to be so much literature on the topic. Similar problems are of course faced by all structural systems in some form or another. Even token comparison systems need to decide which token set to use. An example of a system using syntax graphs is described in [20].

5.5 Information-Based Metrics

Information-based metrics are also based on token sequence comparison, but the comparison is not done using a string matching algorithm. In these metrics, a shared information distance is calculated instead of doing simple string matching. One cannot compare all subsequences appearing in a token sequence as that would be too slow. In this particular sense this approach is weaker than those presented in Section 5.4. But it can – at least in principle – consider all the information in a program instead of just some attributes, so it has an advantage over attribute-based systems. Theoretically, information distance can also consider information which would be complicated to take into account with string matching based structural systems.

The whole measurement process consists of these steps.

1. Convert the input text into token sequence
2. Compute shared information distance for each pair by using compression algorithms to determine the amount of shared information

[6] describes a system which works as described above. It is said to approximate Kolmogorov complexity, to determine the amount of mutual information between two source code texts. It is not clear exactly how the information distance can be calculated, nor do they mention what token set they use.

5.6 Execution Analysis

Malmi et al have used execution analysis to count how many times statements of different types (**`if`**, **`case`**, **`while`**, etc.) get executed when two Pascal programs are run with the same input [23]. Naturally, this approach only works when two programs accept the same input.

Execution analysis is problematic to automate, as we need to be able to do at least the following.

- Compile the programs
- Know how to run (where is the entry point)
- Know what input must be supplied, if any, and where (stdin, file)
- Run the programs (possible security risks, runtime incompatibilities)

In short, execution analysis seems to be possible automatically, but only if we have a restricted domain of applications (i.e. programming assignments). Doing this by hand might still be very important for forensic approaches in a case by case situation. Another weakness resulting from restrictions stated above is that one is restricted to complete runnable programs. Sometimes one might want to compare only some parts of programs with each other.

5.7 Measuring Similarity Across Languages

As mentioned in Section 4.2, programs written in different languages could be co-derivative, as one often takes a program in the source language as the basis for the one in the desired target language. In this section, we consider co-derivative detection across languages. Naturally, when making comparisons across languages, a lot depends on the set of languages that are supported.

If all of the supported languages have a similar syntax, it may well be adequate to generate token sequences from ASTs, for instance, and then compare the sequences. If there are differences in syntax, but the languages are still conceptually similar (say they are all imperative object-oriented languages), the same method might still give satisfactory results, assuming that the tokens are high-level enough to ignore differences in syntax, but contain information about concepts such as method definitions, invocations, etc.

In section 2.2, we mentioned that there are compilers that, during compilation, translate programs into an intermediate representation intended to be relatively independent of the source language(s) supported by the compiler. If one wants to compare programs written in different languages, the IR form of the programs might be a good starting point for generating whatever representation is intended for similarity comparison. The advantage of the IR form is that it is abstracted further away from the source language(s) than ASTs, while hopefully still remaining reasonably independent of the supported target platforms. An IR form would typically contain less semantic information than an AST, however, which might be considered a significant drawback in some applications.

While there are many languages with a similar set of conceptually the same constructs, there tend to be few such similarities between languages of different programming paradigms. Thus, it is likely to be very hard to get satisfactory results when making comparisons across languages such as C++ and Prolog. Even if there was some way to fairly mechanically port a C++ program into Prolog, the programmer may not have used that method of porting. Especially if the programmer has been creative and made non-systematic, radical changes during the port, it will likely be very difficult to spot any significant structural similarity to the original. It might still be possible to find textual similarity from comments, for instance, so all hope is not lost.

6 Source Code Transformations

In this section we cover a selection of transformations performed on source code that have been described in literature. We define a *source code transformation* as an operation that translates source code into some other form, regardless of whether the target form is source code. If we want to emphasize that we are referring to a source code transformation that also results in source code, we may use the term *source-to-source transformation* instead.

6.1 About Implementing Transformers

Compiler technology has been researched for decades, and it is a fairly mature area of computer science. As constructing a compiler involves applying techniques that allow one to both interpret computer programs and to transform them into a different form, it is clear that many of the techniques designed for compilers also have applications when transforming source code into a form designed to facilitate similarity comparison.

Possibly due to the complexity of some programming languages, many of those implementing a tool for analyzing source code similarity have opted to base their implementation on an existing compiler, rather than face the task of writing a lexical, syntactic, or semantic analyzer for the language(s) they wish their tool to support. As some compilers have multiple front ends, it may even be possible to find a single compiler that supports all of the languages one wants the similarity analyzer to accept as input.

In Section 2.2, we listed the phases that a compilation process typically has. In the rest of this section we shall consider some of the tasks that source code similarity analysis might entail, and doing so, we shall also consider their relation to the compilation phases presented earlier. Giving some thought to implementation is important, as a method that would produce good results in theory is of little practical use if creating a usable implementation is infeasible.

6.2 Standard Preprocessing

Some programming languages assume the use of a preprocessor; for instance, it would be inconvenient to write a large system in C without the ability to `#include` declarations from other source files or to `#define` constants using preprocessor directives. (Note that we are using the word directive as a catch-all term for all preprocessor-recognized statements and expressions, including macro definitions and references.)

Depending on the preprocessor language and the "main" source language and their interaction, it may or may not be feasible to make the "preprocessor" directives an integral part of the source language, which would facilitate semantical analysis by a compiler. If it is infeasible, then realistically speaking the preprocessing has to be done as a separate compilation phase, or perhaps integrated as a part of the lexical analysis phase. This is unfortunate as some information then is lost before the semantical analysis phase; especially macros tend to have a higher abstraction level than their expanded form. Consider the following example:

Listing 1: Code with preprocessor macros.

```
#define FIRE_BUTTON_MASK 0x10
#define IS_FIRING(x) (x&FIRE_BUTTON_MASK)
#define MAX_NUM_CONTROLLERS 4

for (int i=0; i<MAX_NUM_CONTROLLERS; i++) {
    if (IS_FIRING(controller[i]))
        fire();
        break;
    }
}
```

Listing 2: Code without macros.

```
for (int i=0; i<4; i++) {
```

```
    if (controller[i] & 0x10) {
        fire();
        break;
    }
}
```

The former is easier to understand semantically, while in the latter the computation is shown more explicitly. Thus, if we want to compare computation, the comparison is easier after preprocessing. On the other hand, if we are more interested in semantics, then it would be better to be able to compare the similarity of source code texts without preprocessing, but that may prove difficult, and compromises may be required.

When one wants to fully parse source code, it is likely to be far easier to preprocess than not, at least in case of languages such as C and C++. For instance, the CCHECK tool described in [23] fully parses the C code given as input, and its developers did opt to use a C preprocessor while transforming programs into the form used in comparisons.

Problems with parsing non-preprocessed code have been faced by some of those developers implementing tools for generating API documentation directly from code, because API documentation is meant to explain the meaning of an API, and should preferably contain constants and functions that are meant to be public, even if they have been defined as preprocessor macros. A close look at tools such as Doxygen [11] might provide more insight into this matter, but we shall leave such details beyond the scope of this paper.

6.3 Tokenization

With structural similarity metrics (see Section 5.4), it is standard practice to transform source code to a token sequence before comparison. Structure-based similarity measurement tools such as JPlag [26], YAP3 [31], and dup [3] all tokenize their input.

6.3.1 Motivation

One of the reasons for tokenizing source code texts before comparison is that the comparison is performed using a string matching algorithm, and the running time of those algorithms is typically greatly affected by the input size. It takes far longer to compare two 1000-character programs character by character than it takes to compare two 100-token programs token by token. Each token can be given a single integer value, just like each character in practice is, and thus comparing sequences of n tokens can be just as fast as comparing sequences of n characters.

At least in the context of plagiarism detection, many also advocate tokenization as a way to discard information that is easy change, and leave behind something that captures the "essence" of a program. If we wanted perfect discrimination in detecting programs that have the same essence, we would have to be able to find a token set that captures all of the essence of a given program and nothing else.

Depending on the definition for the term program essence, it may or may not be possible to find such an ideal token set, but in this section we shall assume that for methods that yield the best results in co-derivative detection, the definition would be such that compromises between recall and precision will have to be made.

6.3.2 Considerations for Token Set Selection

There would not appear to be clear guidelines for choosing a token set in literature. The process appears to be along the lines of making educated guesses of what might be good choices, and then measuring and adjusting until one gets optimal results with a data set. With this approach a lot depends on the data set; it should be large and representative of real-world data that is to be measured with the tool. Also, although measuring with good data is an effective way to determine a suitable minimum match length, it is not possible to perform measurements with all conceivable token sets.

When selecting the token sets to try, we should decide whether the purpose of the token sequence comparison is to act as an assurance method or a screening method. If we are basing our similarity measurement solely on a token sequence comparison method, then we should probably clearly favor precision over recall to avoid false positives. In general, we can improve precision by altering the parameters of our method so that one or both of the following becomes true:

- we have tokens generated from smaller source code elements, and as the number of tokens being emitted then gets higher, we also increase the minimum match length
- we increase the number of distinct tokens in the token set, but do not decrease the minimum match length

We are basing the above guidelines on the assumption that if token sequences are made longer, then the minimum match length should be longer as well to achieve similar results, and that the opposite is true if sequences are made shorter. The results given in [26] would appear to be in line with these assumptions. However, apparently changes in the minimum match length will not have a major effect on the results unless the length is far from optimal.

To improve recall, the above adjustments apply in the opposite. One might be required to make such adjustments if one was using token sequence matching as a screening method, and getting too many false negatives. We do feel that token sequence matching is best suited as a heuristic for efficiently selecting code sequences that should be subjected to further, more accurate analysis.

If one does require results that are as accurate as possible, using sequence matching alone, it may be necessary to complicate matters by foregoing the use of a fixed parameter set for all input. If one were to adjust both the token set and the minimum match length according to the nature of the input, it is reasonable to expect better results than with parameters measured to be optimal in the average case, especially if the data being measured is atypical.

6.3.3 Example Tokens

The tokens produced for similarity measurement purposes are not to be confused with those produced internally by a compiler for the source language; they are typically different.

For instance, consider the Java statement:

```
// count the amount of standard input
while (System.in.read() != -1)
  count++;
```

While a typical compiler would produce a token sequence such as:

WHILE LPAREN IDENTIFIER PERIOD IDENTIFIER PERIOD IDENTIFIER LPAREN RPAREN NEQ NUMBER RPAREN IDENTIFIER INC SEMICOLON

JPlag instead produces the sequence [26]:

APPLY BEGINWHILE ASSIGN ENDWHILE

The latter sequence is considerably more high-level than the former, and discards a lot of information, such as the nature of the assignment (something is being incremented by one). One commonality between both sequences is that neither contains any information regarding comments or the amount of whitespace that originally appeared between tokens, and that in itself can be useful if easy-to-change information is to be ignored in comparisons.

For an example of a full token set used in an existing application see [25], which gives JPlag's default token set for Java, along with some measurements of their relative frequencies; the set contains 40 tokens, and JPlag uses the minimum match length of 9 as standard.

6.3.4 Token Sequence Generation

The choice of the token set affects the amount of work involved in tokenization, as with some tokens the choice of the next token to emit requires a better understanding of the source text than with others. Below we discuss the process of generating token sequences, in the order of exceeding difficulty according to the type of the token set.

Lexical tokens. It is possible to use tokens such that it takes little (if any) knowledge of the context of a source text to choose the ones to emit. If this kind of a "low-level" token set is being used, it should be possible to generate a tokenizer with a lexical analyzer generator tool such as Lex. Many compilers implement their tokenizer in the same way.

The use of lexical tokens makes it easier to implement a tokenizer, but this ease comes with the drawback of the tokens having less semantic content.

In Section 6.3.3 we gave a sample of a token sequence that might be generated by JPlag from Java code. However, JPlag's C/C++ token set is scanner based, and thus it cannot include tokens such as BEGINWHILE, ENDWHILE, BEGINCLASS, etc. Such discriminations are all replaced by just OPEN_BRACE or CLOSE_BRACE. Complex tokens such as APPLY (for function calls) do not occur at all; parenthesis tokens are used instead. [25]

Syntactic tokens. Let us now move a step higher in abstraction than lexical tokens. To know whether the characters `{` and `}` should correspond to the tokens BEGINWHILE and ENDWHILE, we must know whether they were preceded by whatever else is required for constituting a valid **`while`** statement. It is the purpose of a syntactic analyzer to construct an AST that contains such information, and a viable way of creating such an analyzer is to use a parser generator such as Yacc.

The Java version of JPlag is an example of a tool that uses syntactic tokens; all of its 40 tokens can be generated based on the information contained in a syntax tree. We do not know why JPlag does not parse C and C++ code fully, but we suspect the reason might be either preprocessing or a more difficult grammar.

Semantic tokens. Let us now consider tokens that contain even more semantical information than those that a syntactic analyzer could generate. For instance, consider the Java statement:

```
count = 1;
```

A syntactic analyzer would be able to determine that the above statement is an assignment statement; however, syntax analysis cannot tell us whether we are assigning a **`byte`**, an **`int`**, a **`long`**, or a **`float`**. To know which implicit type conversion (if any) must be applied to the **`int`** literal 1, we must know the type of the variable `count`. Thus, if we want to use tokens such as BYTE_ASSIGNMENT, INT_ASSIGNMENT, etc., mere syntactic analysis is inadequate. Likewise, if we want to distinguish between local, instance, and class variables, we must know how the variable `count` currently in the assignment scope was declared.

As mentioned in Section 2.2, variable references are typically associated with variable declarations during the semantic analysis phase, in which a so-called *symbol table* is constructed. If type information is required by a tokenizer, one could consider taking an existing compiler, and adding a phase after the semantic analysis during which the tokens are emitted.

6.3.5 Opcode Sequences

A special case of a token set is one whose tokens map one-to-one to an instruction set of a machine. It is worth considering this case separately, as it is could perhaps allow us to more closely compare the computation that is performed by two programs. However, the problem here is choosing how to translate a program into a stream of instructions. For instance, consider the following Java source code:

```
count += 10;
```

We could translate the above code to either one of the following instruction sequences:

```
iload 1       ; push count onto stack
bipush 10     ; push int 10 onto stack
iadd          ; add the two integers
istore 1      ; store the result in count

bipush 10     ; push int 10 onto stack
iload 1       ; push count onto stack
iadd          ; add the two integers
istore 1      ; store the result in count
```

We would also attain the correct result (i.e. code that increases the value of local variable `count` by 10) with less straightforward code, such as subtracting the value -10 from `count`, and needless to say, with a longer program we would have even more alternatives on how to emit the instructions for it. However, with source languages that are so low level that they already specify the instructions and their order, we do not need to choose the instruction sequence ourselves. An example of such a language is TCL [2].

In [13], Haikala describes how, for the purpose of analyzing program similarity, he translated TCL programs into operation code (opcode) sequences. Due to the primitivity of the source language, all that was necessary to do this was to always emit the opcodes fully in upper case, and to leave out any other parts of the instructions, as well as spaces, tabs, and comments. After translation, the common opcode subsequences of length 4 or more were then located, and statistics collected to help in establishing the likelihood of plagiarism.

Apparently the above method was reasonably effective, as it automatically found most of the similar sequences that had been found manually by another party. However, the comparison of opcode sequences is unlikely to be as effective a solution generally, for two reasons:

1. Most programs today are written in high-level languages, and with them it is not clear how to generate the opcode sequences for optimal results in plagiarism detection. This is a problem in particular if we wish to compare source code to object code.

[2] We are not talking about Tcl, the more widely used high-level scripting language.

2. TCL is unusual in the sense that it is difficult to reorder instructions, due to relative jump instructions. This naturally helps in plagiarism detection, as reordering should be less frequent than in cases where absolute jump addresses are used.

6.4 Comment Removal

A number of source code similarity tools ignore comments when comparing texts. As the textual content of comments typically does not conform to the syntax of the source language, it does not make sense to attempt to parse the content according to the rules of the source language. Thus, it makes sense to filter comments out already before the syntactic analysis, in the lexical analysis phase. If they should be needed for some purpose, each comment can be stored into a single token.

One should note, however, that in some languages it is possible to write comments that cannot be identified as comments based on lexical analysis alone. For instance, Ruby and Python do not have explicit multi-line comments (such as those in C++), and it is common to use multi-line strings instead; in both languages, statements consisting only of a single string literal are legal, but such statements have no effect when not being used as return values, and may in those cases be considered equivalent to comments. However, such "comments" are probably best detected in the syntactic analysis phase.

6.5 Code Canonization

In the context of source code, we use the term *canonical form* as given in Definition 6.1. Furthermore, we define *canonization* as a source-to-source transformation that transforms a given source code text into its canonical form. Canonization may well be essential in some source code analysis applications, and has been used for instance in [23] and [27].

Definition 6.1 *Let $P = \{p_1, ..., p_N\}$ be the set of all source code texts that are equivalent in a certain respect. Let there also be a single text p_C that has been previously agreed to be representative of all elements in P. Then $\forall p_i \in P$, p_C is the canonical form of p_i. If the canonical form is expressed in the source language, then it additionally holds that $p_C \in P$.*

6.5.1 Motivation

One of the main applications of canonization is facilitating straightforward comparisons, and that is also our interest in it here. When comparing two programs, we might wish to transform both of them into their canonical forms before doing some kind of a comparison between them. If all we wish to do is determine whether two programs are equivalent in a certain respect, say whether they are semantically equivalent, converting them into their semantically equivalent canonical forms, and then comparing the results character-by-character is enough to establish equivalence. For instance, consider the following two C++ functions, which are semantically equivalent:

```
int one(Obj& obj)
{
    obj.value = 1;
    return obj.value;
}

int one(Obj& obj)
{
    (&obj)->value = 1;
    return *&obj.value;
}
```

In the case of the above two programs, a character-by-character comparison indicates that the functions are different. However, suppose that we consider `a.b` as the canonical form of `(&a)->b`, and `a` as the canonical form of `*&a`, and we then canonize both functions. The canonization yields textually identical texts.

If it were possible to create a perfect canonizer for some language, i.e. a tool that would transform any program in that language into a form into which any other semantically equivalent program would also be transformed into, then it would be trivial to write a routine for comparing canonical forms in linear time to determine whether any two programs in the language have the same semantics. Such a solution would also yield perfect discrimination. Unfortunately, it is unrealistic to expect to attain perfect canonization for a typical generic-purpose programming language, but applied correctly canonization can still be used in many cases to improve recall without a reduction in assurance.

6.5.2 Semantic vs Computational Equivalence

While it is fairly easy to determine whether a certain source code text is semantically equivalent to another one, referring to the programming language specification as necessary, it is good to keep in mind that semantic equivalence does not always imply computational equivalence. One often needs to resort to guessing about computational equivalence, as a lot depends on the compiler being used and the optimizations it performs on the code.

For instance, most Java compilers probably treat `for(;;)` and `while(true)` as equivalent expressions in the sense that exactly the same object code would be emitted for them. However, this is not to say that there could not be a compiler that would produce differing code. Thus, one should generally be prepared to accept computational changes when making source-to-source transformations intended to maintain existing semantics. Even merely reordering declarations (and not touching any statements) could affect computation. Consider the following C snippets, for instance:

```
int f()
```

```
{
    int b, a;
    a = fa();
    b = fb(a);
    return fc(a, b);
}

int f()
{
    int a, b;
    a = fa();
    b = fb(a);
    return fc(a, b);
}
```

In the latter snippet the local variables have been placed in an alphabetical order. This is likely to affect the order in which the data appears in the stack frame, if they are placed on stack at all, and register assignments might also be affected. The functions will still return the same value and have the same side effects (if any), but there may be differences in the way they get executed by a machine.

6.5.3 Canonical Style and Commenting

Pretty printers are probably the most common canonizers. They were discussed in Section 2.4, and could be used attain canonical formatting when used with certain known settings. The capabilities of pretty printers are not necessarily limited to modifying spacing to attain a uniform indentation and delimiter usage, as some pretty printers also alter casing of identifiers, for instance. Even when a pretty printer is not powerful enough to perform the desired transformation, it might still be a good starting point for implementing the required canonizer.

Some pretty printers may be capable enough to enforce uniform syntax and formatting and placement of comments, which should ensure that only comments with different content result in a different canonical representation. However, depending on the application it might be desirable not to include comments in the canonical representation, and have them removed altogether as suggested in Section 6.4.

6.5.4 Example Canonizations

There are numerous canonizations that one could conceive for a typical general-purpose programming language, each canonizing some aspect of the code in some respect. It would be futile to attempt to give an exhaustive list, but below we give some examples, most taken from existing literature.

Single statement blocks. In C, for instance, the following two code snippets have the same semantics.

```
if (a()) { b(); } else { c(); }
```

```
if (a()) b(); else c();
```

We might decide that the canonical form of `{ b(); }` is `b();` i.e. that the canonical form of a block containing a single statement is the single statement contained in the block.

Multi-variable declarations. In C, one may declare multiple variables of the same type with a single declaration as follows:

```
int a, b, c;
```

Before comparing program pairs, CCHECK [23] would transform the above declaration into the following semantically and computationally equivalent form:

```
int a; int b; int c;
```

According to [23], CCHECK also performs the following transformations on applicable pointer expressions:

Operand removal `a->b` becomes `(*a.b)` [3]

- `(&a)->b` becomes `a.b`

Separating initializations from declarations. CCHECK replaces initializations in variable declarations by assignment statements after variable declarations [23]. For instance, `int a = 0;` would be replaced with `int a; a = 0;`. This kind of canonization should increase the likelihood of two variable declarations of the same name and type having the same canonical representation.

For more canonization ideas, refer to Section 3.4, which discusses synonymous expressions; each synonym could perhaps be replaced with an equivalent, canonical expression.

7 String Matching Algorithms

In order to compare two pieces of source code, we need algorithms to do the comparison on a computer. Such algorithms would need to be able to express the differences between two pieces to be compared with some resolution. It is trivial to conclude whether two pieces are exactly the same; we would just do a bit-by-bit comparison of the pieces. This

[3] The reader may notice that `(*a.b)`, unlike `(*a).b` is not valid C code (assuming that `a` is a pointer), but as CCHECK expresses its canonical forms in a custom language, there is no requirement to produce valid C expressions.

is what computers do best, detect and report even minor differences. What we would like to do here, is to get some kind of an idea of how similar pieces of source code are, even if they are not exactly the same.

When we are producing a similarity score (between 0 and 1), we have to be using some metric. We must thus be able to measure the distance between the two entities somehow. Defining the exact properties of this metric intelligently is crucial in order to have meaningful results. In this survey, we have previously looked at methods of transforming source code to forms in which unwanted differences are removed. In this section, we turn to the problem of quantifying the differences between such transformed codes.

Generally, comparing two pieces of transformed source code with each other can be looked as a comparison of two strings. One meaningful way of quantifying the similarity between two "plain" strings is to find a longest common subsequence. A Longest Common Subsequence or LCS is a string that is a subsequence of both of the strings and is no shorter than any other such sequence. For example, "striper" and "tiger" have an LCS of "tier".

We can now take a crucial step in quantifying the similarity of two strings. By defining the similarity score as 1 if the strings are exactly the same or one string completely includes the other we get a useful metric. By dividing the longest common substring with the length of the longer string, we get another often used metric [26]. The advantage with the first definition is it returns 1 if and only if the files are exactly the same. Which property – the former or the latter – is more suitable, depends on the application i.e. whether we want to consider primarily whole pieces of software or include major inclusions also. Naturally, the choice is also affected by the motivation to construct either a screening method or an assurance method.

7.1 Evolution in String Matching

The wide area of string and sequence matching, having applications in computer science, computational biology and speech recognition, is collectively known as sequence analysis [5].

Many of the currently popular string matching algorithms are based on a common approach called sparse dynamic programming [5]. The first LCS approach to use sparse dynamic programming was Hunt-Szymanski [17]. Since the original Hunt-Szymanski the algorithms have advanced significantly and depending on which performance criteria are considered most important, there are a variety of choices. The running time of all dynamic programming based algorithms is shown to be $O(n^2)$ [21], where n is the size of the input.

GNU `diff` uses the algorithm described in [24]. GNU diff has a running time of $O(nd)$ where d is the size of the minimal edit script. In the context of `diff`, it is important that the minimal edit script is found in the process and that the running space grows only linearly. This makes versioning and patching of different versions of large text files practical. GNU `diff` uses a dynamic programming solution for the LCS.

Let us emphasize, the Longest Common Subsequence problem or LCS is a strictly defined computational problem. $LCS(x, y)$ of two strings x and y is defined in the following way.

1. It is a subsequence of both x and y
2. It is as long as any such sequence

The LCS problem has a close relation with a Minimal Edit Scripts problem. If the allowable operations are insertion and deletion of characters (as in `diff`), the minimal edit script problem (what operations are required to transform one text into another) is computationally equivalent to finding the longest common subsequence of two strings [4]. In fact, it can be shown that a distance d between x and y (in the edit script sense) is always $d(x, y) = |x| + |y| - 2 * |LCS(x, y)|$ [1].

7.2 Parameterized LCS

Parameterized LCS is a reconstruction of the original LCS problem. The idea is to allow not only insertion and deletion of characters, but also systematic replacements.

In practice, changing texts includes not only insertions and deletions but also replacements. Thus, the notion of edit distance is extended to allow also global substitutions via parameterized match [3, 4]. For example, strings "Supra suprenum supra suprenum supra" and "Supra fword supra fword supra" is considered a perfect match under a parameterized match, as the substring "suprenum" has been globally replaced with the substring "fword".

7.3 Heckel

Heckel's string comparison algorithm [15] is able to take into account moved blocks. The running time is also linear ($O(n)$), which is very efficient in the field of string matching. However, Heckel's algorithm is less suitable for our needs, i.e. the detection of co-derivative source code, as it off-syncs badly in cases of added lines of text. Adding unnecessary lines, statements or declarations, for example, is something that is easy for a plagiarists to do.

7.4 RKR-GST

Two of the surveyed plagiarism detection systems – namely YAP3 [31] and JPlag [26] – use a novel approach called Running Karp-Rabin Greedy String Tiling (RKR-GST) [30]. It is also able to take into account moved blocks. Although it has a worst case complexity of $O(n^3)$, the experimentally derived complexity in a certain biological application is shown to be $O(n^{1.12})$. Using minimum match length of 3 a complexity of $O(n^{0.90})$ has been achieved.

In RKR-GST we are in effect tiling a longer sequence with the smaller one and trying to produce a maximal cover. A minimum match length is a parameter for the matching algorithm specifying how long matches must at least be to be accepted. Using a minimum match length of 2, in the the example above, we would have LCS of "er", since at least two consecutive characters would need to appear in both strings for the matching to be triggered.

The minimum match length has been determined empirically for the surveyed plagiarism detection systems. Strong caution is advised here, since manipulating this parameter leads to significantly different quantitative results. In [26] it is noted, that the results do not vary very easily if minimum match length is changed somewhat (say from 3 to 5 for example). This reasoning is backed up with some amount of real world experiments. However, this still does not change the fact that there is no easily determined natural value for a minimum match length. It is thus possible to manipulate results by changing this parameter. The best motivation for selecting a minimum match length we have come across during this survey is the following: longer matches are preferable to short ones, as they are more likely reflect significant similarities rather than chance similarities [30].

8 Existing Technologies

In this section we go through some of the existing plagiarism detection tools, including tools that were not designed specifically for the purpose of detecting plagiarisms, but can be applied for that purpose.

8.1 Donaldson et al System

Donaldson et al system [18] is a hybrid system utilizing both attribute counting and structural analysis for Fortran programs. The system's attribute counting phase counts the total number of the following attributes.

- Variables
- Subprograms
- Input statements
- Conditional statements
- Loop statements
- Assignment statements
- Calls to subprograms

Since attribute counting systems are considered inferior [28], it is more interesting that this is the first structural metrics based plagiarism detection system we have been able to find. It is instructive to consider the details of the earliest system utilizing this approach.

Structural analysis consists of a scanner tokenizing statements. Some attempt is made to mitigate actions of the plagiarists by compressing the resulting token string. In practice, this means that sequences of identical contiguous tokens are replaced with one occurrence of that specific token. This means that splitting statements would not affect the resulting token sequences.

The analysis phase is a straightforward token per token comparison, which can only report exact matches of the two token sequences. It is noted that the use of `GOTO` statements could easily confuse the used structural analysis.

8.2 Accuse

Accuse is an attribute counting system for detecting plagiarism in Pascal programs. It is stated in [12] that efficiency was the primary reason for choosing this approach instead of a more involved structural analysis.

The paper presents a correlation scheme for judging whether plagiarism has occurred. The correlation scheme is scaled with a group of 43 programs. Three contributions were known to be a result of team work. The parameters of the correlation scheme of Accuse were scaled so that these three programs ended up judged as suspected of plagiarism. The analysis in this paper is not particularly thorough; it seems attribute counting is more of an experiment.

8.3 Plague

Plague is one of the first plagiarism detection systems using structural metrics instead of attribute counting. In [29] attribute counting systems and structural metrics systems are – to our knowledge – compared comprehensively for the first time. The author of Plague argues convincingly, based on measurements, that attribute counting systems are not able to achieve sufficient recall to be used as a screening method. Much of the current terminology and concepts used to evaluate efficiency of plagiarism detection systems are from [29]. Most notably, the key concepts of precision and recall, in the context cf plagiarism detection, originate from the author of Plague.

Plague operates in 3 stages. The first phase extracts a program's control structure into a proprietary format, which is then used for a pairwise comparison. The purpose of this phase is to select a limited number of pairs for later stages. The first phase has an intentionally low precision in the hope of finding all or most of the essential matches. To defeat this stage, a plagiarist would need to make his copy resemble some other submission more than the original. This phase is thus a first phase of a screening method in our terminology.

The second phase is designed to have high precision. This phase is based on parsing the programs in question and some additional modifications which discard superficial changes. Procedures are inlined and ordered by a depth-first traversal of the calling graph. In our terminology, this preprocessing constitutes a first phase of an assurance method.

The third phase is the usual LCS comparison phase, where modified token sequences are compared. Plague also offers Heckel's algorithm [15] as a replacement for the usual dynamic LCS. The use of Heckel, instead of the usual order preserving LCS, gives the advantage of taking into account relocated blocks of code.

8.4 PAHTA and CCHECK

PAHTA and CCHECK [23] are companion programs from the same authors. We will first go through PAHTA and then point out relevant differences in CCHECK. One of the main distinction between PAHTA and CCHECK is that the former operates on Pascal programs, while the latter inspects C programs.

PAHTA is a hybrid system including both attribute counting and structural metrics for Pascal programs. It is interesting that PAHTA also does a third type of analysis, namely execution analysis. It is hypothesized that this will give better results than only using one of the approaches. PAHTA does full parsing and does not use simple LCS matching for a linear token stream as many other systems do. Instead it operates with an Abstract Syntax Tree of the program under analysis. The exact algorithm for comparing ASTs is not given. We consider it an open problem to evaluate, whether using full ASTs is more accurate than using linear token stream with LCS matching.

Execution analysis is done by interpreting the aforementioned AST. The system then counts how many times each statement type is executed for a given input. It is of course in general hard to give predefined input, as the method of input depends on the type of program and its user interface. It is mentioned that this is not a problem for well defined simple course exercises used as a basis in [23].

An interesting result from the authors of PAHTA is that all the used methods failed at least once, but in none of the cases all methods failed at the same time. This would seem to suggest that a good screening system would utilize all the aforementioned methods. This would need further independent verification. The simple coincidence could well explain the anecdotal evidence in [23].

CHECK is mostly the same as PAHTA. The structural analysis also does full parsing, but before analysis the code is normalized (canonized) to make the comparison itself easier. The main reason for this is the richness of the C language: it would be more burdensome not to do the normalization. PAHTA does not do execution analysis.

8.5 Dup

Dup [3] is not a plagiarism detection tool per se. It is meant as a tool for maintaining large software systems and it can detect duplication and near duplication of code inside a given software. The duplication of entire fragments of code sometimes happens inside a software system, because of factors quite similar to those mentioned in Section 3.5. The redundant and perhaps minorly modified code resulting from a programmers' cut & paste attitude makes the program larger, more complex and thus more difficult to maintain.

Dup has been tested using several millions of lines of production code. The level of duplication that the authors found was quite big: A system with 1.1 million lines of code, 605 000 lines after comment and whitespace removal was found to have approximately 20% code duplication of which an estimated 13% could be removed. It is worth noting that the studied system did not have machine-generated code. This gives assurance that identical unnecessary duplication of functionality in two distinct systems may be an indication of co-derivation, as we noted in Section 4.

Dup does both exact matching per line and efficient parameterized matching per line. For our purposes parameterized matching or *p-matching* is interesting, since it can identify code segments as essentially the same even though they have been modified superficially. Two sections of code are said to be a p-match if there is one-to-one function that maps parameters from one section onto a set of parameters in the other section. Dup considers identifiers, constants, field names and macro names as parameters.

Efficient p-matching is achieved by generating p-strings with the help of a lexical analyzer. A p-string is a string where both parameters and their position is included. It is a result of concatenating both non-parameter symbols and parameter symbols. It is worth noting that only lexical analysis – not full parsing – is performed.

Dup uses a data structure called *parameterized suffix tree* to do the comparison efficiently. Overall running time has been found linear even though worst-case complexity is $O(n^2)$.

8.6 YAP3

YAP3 [31] is a structure metric system, and as the name implies, it is a third version of a program from the same author. All the versions work in two phases. In the first phase, source texts are used to generate token sequences (tokenization). The second phase is a comparison phase. Here, all the token sequences under investigation are compared to produce a quantitative estimate of their similarity for each pair.

Basically the first phase abstracts and normalizes the input so that the comparison phase can achieve more meaningful results. This phase is mostly similar in all versions of YAP. The transformations performed are the following.

- Comments and string constants are removed

- Uppercase letters are lowercased
- Synonymous functions are normalized (e.g. `strncmp` is transformed to `strcmp`)
- Functions are reordered in to their calling order
- The first call to a function is expanded and the subsequent calls are replaced by a generic function token (FUN)
- All tokens not in the target language lexicon are removed

One of the main differences in the first phase of YAP3, compared to earlier versions, is that it produces numerical tokens instead of string tokens.

The evolution of YAP has concentrated on the comparison phase. YAP1 used UNIX `sdiff` in the comparison, which uses dynamic programming to solve LCS, whereas YAP2 used Heckel's algorithm in the second phase. Finally, the comparison phase of YAP3 uses RKR-GST.

YAP3 does not do a full parse of the target language. For this reason, it is more easily modified to support new languages, even those with an unknown formal grammar. (As an example, a version for the English language has been devised.) This approach also makes the resulting analyzer faster. Still, not doing a full parse means that some semantic information in the analyzed programs worth taking into account might not get extracted.

8.7 PDiff

PDiff [4] is a parameterized version of the LCS solution, and it is thus not a fully functional plagiarism detection system. It could, for example, serve as a generic back end for a system detecting co-derivative source code.

PDiff solves a parameterized version of the edit distance problem. In practice, this means that parts of the strings under comparison are considered to match, even if there are systematic replacements. Parameterized match models well a changing of source code text, which has been cut&pasted to a different context. The algorithm in this PDiff is the same as that used in dup [3]. This publication, however, proves that parameterized matches can be done in linear space and time.

8.8 JPlag

Authors of JPlag [26] give considerable information of its design principles and supply extensive amount of statistical data of its performance when Java is used [26]. JPlag uses a structure-based strategy with full parsing for Java and Scheme. Comparison of token sequences is based on Running Karp-Rabin Greedy String Tiling and is nearly linear in the average case [19].

JPlag supports fully automated decisions. Authors provide extensive analysis of a meaningful determination of a *cutoff criterion*, which is the borderline similarity value dividing supposed plagiarisms and supposed non-plagiarisms. A fixed value of 50% is deemed as a good cutoff value when recall is considered 3 times as important as precision.

8.9 Comparison

It is clear that considerable technological progress has been made in the field of automatic detection of co-derivative code since the first non-trivial systems appeared in the 1970's. Table 1 summarizes the properties of the systems listed above.

	Target language(s)	**Attribute counting / Structural metrics**	**String comparison algorithm**	**Full parsing**
Donaldson et al	Fortran	both	exact match	no
Accuse	Pascal	attribute	N/A	no
Plague	Pascal, Prolog, Bourne Shell scripts	structural	LCS, Heckel	Pascal
PAHTA	Pascal	both	N/A (AST)	yes
CCHECK	C	both	N/A (AST)	yes
Dup	N/A	N/A	parameterized LCS	no
YAP2	C, Lisp	structural	Heckel	no
YAP3	C, Lisp	structural	RKR-GST	no
PDiff	N/A	N/A	parameterized LCS	no
JPlag	Java, Scheme	structural	RKR-GST	Java, Scheme

Table 1: Comparison of the surveyed systems

9 Future Work

In the process of conducting research for this survey we came across some questions to which we could not find satisfactory answers from the existing literature, and we shall list some of them here as they might give raise to possible future work. On our list, we may also include research topics that have explicitly been mentioned as ongoing or future work in the existing literature.

Benchmarking. For comparing the performance and accuracy of different similarity measurement methods, it would be important to have an open benchmark suite and results archive, as suggested in [22]. Apparently there already are benchmark suites available for other purposes, e.g. the TREC database of test collections for testing information retrieval methods [22].

Lexical vs syntactic tokens. Section 6.3.4 discusses the generation of lexical and syntactic token sequences, but while we have a good idea of how to generate both kinds of token sequences, we do not know how significant the choice between lexical and syntactic tokens is in terms of measurement accuracy. We would like to see measurement results with two systems that are otherwise identical, except that one of them does full parsing and the other one does not.

Tree vs sequence comparison. As mentioned in Section 8.4, we consider it an open problem to determine, which kind of a program syntax representation allows one to attain more accurate co-derivative detection when using a (generalized) string matching algorithm for the comparison: a hierarchical representation, or a "flattened" representation (such as token sequences). The extra information in a hierarchical representation may tend to increase assurance and decrease recall, but it would also be possible to encode some syntax tree information into a token sequence to presumably the same effect.

10 Conclusion

The present systems for the detection of co-derivative source code concentrate their analysis mainly on the structural similarities of programs. Structural similarity can indeed be a strong indication of co-derivation, but it is certainly worth noting that similar program structure is not the only co-derivation indicator. Structural comparison does, however, appear to be the method which produces the soundest results according to the surveyed scientific research.

It should be made equally clear that similarity in structure or function can never by itself conclusively prove co-derivation. On the other hand, the lack of sufficient structural and functional similarity could well be seen as proof that no infringing plagiarism has happened. In some instances, simple non-functional similarities, such as identical commenting, may give further indication as to whether we have co-derivative source code or not. That is because such similarities are mostly independent of the structural similarities of programs, and can thus provide extra assurance on top of any structural similarity measurement results indicating co-derivation.

The state of the art of co-derivative source code detection seems to consistently result from the need to solve the problem of plagiarism in the context of university education. This is a problem setup where considerable gains can be had through the use of screening methods. The field of study of plagiarism detection – even in the detection of co-derivative source code – is not new, and first concrete systems date back to the 1970's. Despite this, there still has been interesting progress relatively recently.

References

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

[2] Andrew W. Appel. *Modern Compiler Implementation in C*. Cambridge University Press, 1998.

[3] Brenda S. Baker. On finding duplication and near-duplication in large software systems. In *Second Working Conference on Reverse Engineering*, Toronto, Ontario, Canada, July 1995.

[4] Brenda S. Baker. Parameterized Diff. In *ACM-SIAM Symposium on Discrete Algorithms (SODA'99)*, Baltimore, Maryland, USA, January 1999.

[5] Brenda S. Baker and Raffaele Giancarlo. Longest common subsequence from fragments via sparse dynamic programming. In *European Symposium on Algorithms*, Venice, Italy, August 1998.

[6] Xin Chen, Ming Li, Brian Mckinnon, and Amit Seker. A theory of uncheatable program plagiarism detection and its practical implementation. Manuscript, May 2002.

[7] Christian Collberg and Clark Thomborson. Watermarking, tamper-proofing, and obfuscation – tools for software protection. Technical Report 2000-03, Department of Computer Science, University of Arizona, Tucson, Arizona, USA, February 2000.

[8] Christian Collberg, Clark Thomborson, and Douglas Low. Breaking abstractions and unstructuring data structures. In *IEEE International Conference on Computer Languages (ICCL'98)*, Chicago, Illinois, USA, May 1998.

[9] Christian Collberg, Clark Thomborson, and Douglas Low. Manufacturing cheap, resilient, and stealthy opaque constructs. In *Principles of Programming Languages 1998 (POPL'98)*, San Diego, California, USA, January 1998.

[10] *diff(1) man page*, September 1993.

[11] Doxygen. http://www.doxygen.org/.

[12] Sam Grier. A tool that detects plagiarism in pascal programs. *ACM SIGCSE Bulletin*, 13:15–20, February 1981.

[13] Ilkka Haikala. Lausunto: Ohjelmistojen samankaltaisuus, VF Partner vs. Systek, August 1997.

[14] Maurice Howard Halstead. *Elements of Software Science*. Elsevier Science, June 1977.

[15] P. Heckel. A technique for isolating differences between files. *Communications of the ACM*, 21:264–268, April 1978.

[16] Timothy C. Hoad and Justin Zobel. Methods for identifying versioned and plagiarised documents. To appear in the *Journal of the American Society for Information Science and Technology*.

[17] J.W. Hunt and T.G. Szymanski. A fast algorithm for computing longest common subsequences. *Communications of the ACM*, 20:350–353, 1977.

[18] Ann-Marie Lancaster John L. Donaldson and Paula H. Sposato. A plagiarism detection system. In *Twelfth SIGCSE Technical Symposium*, St Louis, Missouri, USA, February 1981.

[19] R.M. Karp and M.O. Rabin. Efficient randomized pattern matching algorithms. In *IBM Journal of Research and Development*, pages 249–260, March 1987.

[20] Jens Krinke. Identifying similar code with program dependence graphs. In *Proceedings of Eighth Working Conference on Reverse Engineering (WCRE'01)*, Stuttgart, Germany, October 2001.

[21] Joseph B. Kruskal and David Sankoff. *An Overview of Sequence Comparison*. Addison-Wesley Publishing Company, 1983.

[22] Arun Lakhotia, Junwei Li, Andrew Walenstein, and Yun Yang. Towards a clone detection benchmark suite and results archive. In *Proceedings of the 11th IEEE International Workshop on Program Comprehension (IWPC'03)*, Portland, Oregon, USA, May 2003.

[23] Lauri Malmi, Mårten Henrichson, Timo Karras, Jari Saarhelo, and Sari Särkilahti. Detecting plagiarism in Pascal and C programs. Technical Report TKO-B78, Department of Computer Science, Helsinki University of Technology, Espoo, Finland, 1992.

[24] Eugene W. Myers. An O(ND) difference algorithm and its variations. *Algorithmica*, 1(2):251–266, 1986.

[25] Lutz Prechelt, Guido Malpohl, and Michael Phlippsen. JPlag: Finding plagiarisms among a set of programs. Technical Report 2000-1, Universität Karlsruhe, March 2000.

[26] Lutz Prechelt, Guido Malpohl, and Michael Phlippsen. Finding plagiarisms among a set of programs with JPlag. *Journal of Universal Computer Science*, 8(11):1016–1038, 2002.

[27] Eric S. Raymond. *comparator(1) man page*, September 2003.

[28] Kristina L. Verco and Michael J. Wise. Software for detecting suspected plagiarism: Comparing structure and attribute-counting systems. In *First Australian Conference on Computer Science Education*, Sydney, Australia, July 1996.

[29] G. Whale. Identification of program similarity in large populations. *The Computer Journal*, 33(2):140–146, April 1990.

[30] Michael J. Wise. Neweyes: A system for comparing biological sequences using the Running Karp-Rabin Greedy String-Tiling algorithm. In *Proceedings of the Third International Conference on Intelligent Systems for Molecular Biology*, Cambridge, England, UK, July 1995.

[31] Michael J. Wise. YAP3: Improved detection of similarities in computer program and other texts. In *SIGCSE Technical Symposium*, Philadelphia, Pennsylvania, USA, February 1996.

[32] Wuu Yang. Identifying syntactic differences between two programs. *Software - Practice and Experience*, 21(7):739–755, July 1991.