

Bachelor project

19th May 2004

A Plagiarism Detection Tool

by
Christa Søgaard Fotel
Lars Langer

Contents

1	Introduction	1
2	Plagiarism	2
2.1	Plagiarism in software	2
2.2	Copy detection software already out there	3
2.2.1	Basics about copy detection software	3
2.2.2	YAP3	4
2.2.3	Winnowing	5
2.2.4	MOSS	5
2.3	Counter measures and counter counter measures	6
2.3.1	Counter measures	6
2.3.2	Counter counter measures	6
2.4	Copy detection tool - desirable properties	7
3	XML Store	8
3.1	XML Store and copy detection	8
3.1.1	Benefits	9
3.2	Input	10
4	Design	11
4.1	SML to XML	11
4.1.1	The MosML compiler	12
4.1.2	Dumping XML	13
4.2	Plagiarism detection	13
4.2.1	Plagiarism Attributes	13
4.2.2	Counter measures	15
4.2.3	Public document parts	15
4.2.4	Criteria	16
4.3	Output	17
5	Implementation	20
5.1	Front-end	20
5.2	Copy detection	20
5.2.1	Traversing trees using XML Store	20
5.2.2	Computing the plagiarism attributes	21
5.2.3	Traversing the document trees	21
5.2.4	Storing the plagiarism attributes	22
5.2.5	Public document parts	22
5.2.6	Getting the user defined parameters	23
5.2.7	Applying the criteria	23
5.3	Measure of similarity	23
5.4	Output	24
5.4.1	Weak attributes in XML Store	25

6	Evaluation	26
6.1	Strategy	26
6.2	Tuning of parameters	26
6.2.1	The tuning	27
6.3	The evaluation	29
6.4	Counter measures	31
6.5	Conclusion of the evaluation	32
7	Conclusions	33
	References	35
A	med_i.sml as XML document	37
B	SML files	38
B.1	Tuning	38
B.2	testdepth.sml	39
B.3	Applying counter measures to tun1.sml	39
B.3.1	Renaming of identifiers	39
B.3.2	Transposed code	40
B.3.3	Adding redundant statements	41
B.3.4	Changing fixtivity	42
B.3.5	Changing the structur of selection statements	43
B.4	Changing datatypes	44
C	Source code	46
C.1	Font-end - the XML dumping code	46
C.1.1	Changes made in the SML compiler	46
C.1.2	The XML dumping file	46
C.2	The detection tool	51
C.2.1	The main class	51
C.2.2	Reading the user defined parameters	53
C.2.3	Traversal of the public document tree	54
C.2.4	Traversal of the document trees	55
C.2.5	Traversal of subtrees in Pre-Order	56
C.2.6	Applying the plagiarism criteria to the nodes	57
C.2.7	Storing of the plagiarism attributes	59
C.2.8	Compute the attributes	61
C.2.9	Retrieving objects from a hashtable	62
C.2.10	Making the output for the HTML file	63
C.2.11	Retrieval of locations from SML files	66
C.2.12	Writing of information to HTML file	67
C.2.13	Storage of total percentage	68

Abstract

Plagiarism in student programming assignments is a possibility which needs to be taken into account when a group of students are working on the same project.

Other plagiarism detection tools work by selecting a number of fingerprints for each document to be compared. These fingerprints are then pairwise compared with the fingerprints of the other documents. XML Store is a storing facility for XML documents, which saves XML documents as a tree structure and allows sharing of nodes. By using XML Store as a host for a copy detection tool we can gain the benefits that no fingerprints are needed and no pairwise comparison is needed. Furthermore, shared trees are identified by XML Store, and we have the chance of analysing whether it brings anything to the field of copy detection to look at the structure of programs.

We have designed and implemented a plagiarism detection tool as an application for XML Store. We have shown that this tool works as well as other copy detection tools. We have used XML Store's facilities and can guarantee a fast and always correct identification of copied parts between two or more documents.

1 Introduction

A copy detection software is a piece of software able to identify equal parts between two or more files. This ability can be used in the quest of finding plagiarism in assignments handed in by students at academic institutions. A further discussion of plagiarism in software is found in section 2, which also contains a description of software currently available within this field.

In this project we will use XML Store as host for a copy detection tool. XML Store will identify equal parts of XML documents and save these shared parts only once. Our tool should find these shared parts to flag possible plagiarism, defined by user given parameters. Since XML Store saves XML files in a structure maintaining way, as a DVM tree, we will have the possibility of detecting plagiarism in programs by looking at the syntax of the programs rather than just lexically. XML Store is more closely described in section 3. DVM trees are described in section 3.1

Our tool will concentrate on detecting possible plagiarism in SML programs. We will do this by (1) developing a plagiarism detection tool for XML Store (2) developing an SML front-end for our tool. The front-end should produce XML versions of Standard ML (SML) programs handed in by students. The XML versions of the SML programs should reflect the structure of the original program. The tool should, with the help of XML Store, identify and analyse the shared parts. If these parts are within the user given parameters, e.g. size or indegree, the parts should be reported as possible plagiarism. A human should then take a closer look at these parts and manually determine whether it is plagiarism. The discussion of the design of this front-end is found in section 4.1.

The source code of the developed tool can be downloaded from www.plan-x.org/projects/plagiarism/pdt_v100.htm

2 Plagiarism

Plagiarism is a legal term referring to the use of someone's ideas, information, language, or writing, when done without proper acknowledgment of the original source. ... Like most terms from the area of intellectual property, plagiarism is a concept of the modern age and not really applicable to medieval or ancient works.

en.wikipedi.org

Since the introduction of the Internet, plagiarism has become a growing business. Sites on the Internet offer free or almost free papers for download by students feeling the pressure of performing well and passing exams. For our field of study, the computer science, it is also possible to download solution manuals for most textbooks or find solutions somewhere on the Internet. This is not a critical kind of plagiarism since the exercises, and their solutions, most often are meant as a help to students. Software plagiarism is not only known within academic institutions but also among private enterprises. We know of the SCO versus IBM and Linux case and the IBCOS Computers Ltd. and Barclays Mercantile Highland Finance Ltd case, [Clo].

A kind of plagiarism more serious, at academic institutions, is the exchange of code, or even theft, which is likely to happen when many students are working individually, or in groups, on larger grade giving assignments. By using plagiarism here the students learn less and seem to be able to do more - a difficult situation for the teachers and future employers. It is therefore essential to find and stop plagiarism when used in these projects.

Pure text assignments and programming assignments have some things in common. However there are some differences, which we will take a look at in the following section.

2.1 Plagiarism in software

In ordinary written texts there are a number of parameters to be taken into consideration - sequences of common text, punctuation, order of text, frequency of words etc., [Clo], p. 5. The techniques, and the sophistication, of plagiariser varies. Some use “copy-and-paste”, others paraphrases.

In computer programs direct copying can easily be discovered by using ordinary plagiarism detection tools. The hard part comes when students re-order statements, rename variables, change comments, split up or paraphrase functions or methods. When designing the plagiarism detection tool it is important that we carefully consider, what should be legal for students and what not. E.g. advanced paraphrasing of code requires in-depth understanding of the functions and a good understanding of the programming language. We can therefore argue that such plagiarism is not a kind of plagiarism important. However paraphrasing software is still the theft of ideas. Teachers or instructors may also have given a number of functions, freely to be used by the students - it is, of course, not plagiarism to use these. A way of getting around this is to allow the user of the tool to parameterise the tool, i.e. give a list of public parts (parts which can freely be used by all) as input.

Because computer languages are well defined it is considered more easy to detect similarities. Namely the parse-tree will often look the same in the original program as well as in the plagiarised program. It could therefore be a good idea to look at the structure and syntax of a program rather than the semantics.

2.2 Copy detection software already out there

In this section we will take a closer look at some existing software within the field of software copy detection. We will take a closer look at how they get around some of the problems that may arise. The software have been identified with the help of [Clo] and the original papers. We will also introduced some basic knowledge of how copy detection is normally done.

2.2.1 Basics about copy detection software

Most copy detection tools work in the following way. First the input is normalised. For input in the form of software programs this could mean renaming of identifiers to a single value, e.g. “V”, remove white spaces, translate upper case to lower case etc. After the normalisation routine the programs are further analysed. The tokens are combined in to overlapping shreds, so-called grams, as shown in figure 1.

```
val n = numb + tempnum;
```

(a) Some text

```
valn aln= ln=n n=nu =num numb umb+ mb+t b+te empn mpnu pnum num;
```

(b) The sequence of 4-grams derived from the program statement - spaces removed

Figure 1: n-grams definition

The number of n -grams generated this way is equal to

$$\text{number of characters in document} - (n - 1)$$

Due to the vast amount of n -grams generated in this way for large files, the n -grams are hashed to reduce the need for memory. A further reduction in the amount of hashed n -grams has to be made for the copy detection software to function in a reasonable¹ way for large input files. This is normally done by selecting a subset of fingerprints for each document. A subset of the hashed n -grams are chosen as sample for the entire document, this subset is sometimes referred to as the documents *fingerprints*. These fingerprints are chosen in different ways depending on the particular piece of software; a often used way is to use the $0 \bmod p$ approach for some fixed size of p [ASW03]. That is, if the hash value of a n -gram modulo p equals 0 that n -gram is chosen as one of the fingerprints and added to the fingerprint subset of that document. This process is applied to all the documents to be compared.

After the normalisation and fingerprinting procedure the documents are now ready to be compared. The documents are compared pairwise, meaning that first the fingerprints of document1 is compared with the fingerprints of document2 and so on - document2

¹With concern to the running time

is then compared with document₃ etc. and finally document_{*i*}. This takes quite a number of comparisons, as seen below:

$$(i-1)+(i-2)+(i-3) + \dots + n-(n-1) = \frac{i * (i - 1)}{2}$$

Moreover, each fingerprint of a document needs to be compared with the fingerprints of the the other document. As we will show later in this report, using XML Store as a host for a copy detection tool will decrease the number of comparisons to the number of nodes in the DVM tree, which in worst case will be the number of nodes in all the documents being compared - this discussion will be continued in section 3.

According to Michael J. Wise - the author of YAP - a plagiarism measuring software should have three properties [CFL⁺03]:

1. Each token in a string should be reported only once.
2. Transposed code segments should have minimal effect on the resulting similarity score.
3. The score must degrade gracefully in the presence of random insertions and or deletions of tokens.

Which counter measures one needs to take into consideration when designing the software will be discussed later in section 2.3.1.

2.2.2 YAP3

A description of YAP3 is found in [Wis] and in [Clo] pp. 13-14. This software guards itself against counter measures taken by students in the following way. Since some plagiarisers reorder, YAP3 (the successor of YAP1 and YAP2) uses the *Running Karp-Rabin Greedy String-Tiling*, which is used to detect transposed substrings. This algorithm is also used within DNA research to help find DNA substrings. The Running Karp-Rabin algorithm is used to find a substring of a certain length *n* within a much longer string. Karp-Rabin compares all *n*-grams within the long string with the *n*-gram searched for. It is expensive to hash a long string in this way so Karp-Rabin suggest the use of a rolling hash function - rolling in the sense that it is possible to calculate the next hash value fast from the previous. The exact algorithm is given in [ASW03] section 2.2.

More generally YAP3 works in the following way (for further details please see [Wis]): While tokenizing the text it:

- Removes comments and string constants.
- Translates upper case to lower case.
- Maps synonyms to common form - e.g. “function” to “procedure”.
- Reorders functions in their calling order.
- Removes tokens not from the lexicon of the programming language.

The tokens obtained this way are grouped into sequences of a certain size and are stored in a hash table. The sequences are compared to other documents and if equal a match is reported. It is then up to a human to take a closer look at the matches. The problem with YAP3 is that it is order preserving. If a large chunk of text is moved YAP3 regards this as line differences rather than as a block move, see [Clo] pp. 13-14.

2.2.3 Winnowing

Basically the Winnowing algorithm works in the same way as described in section 2.2.1, but separates itself from the rest by the way it selects the fingerprints. Other algorithms select a number of the hashed n -grams as fingerprints for the documents by using the $0 \bmod p$ approach. In this way they may leave large gaps between the fingerprints and thereby allow copied parts to go undetected. To avoid this Winnowing works with a window of consecutive n -grams. By making their algorithm select at least one fingerprint from each window, Winnowing can guarantee to detect at least one n -gram in a shared substring of length $w + n - 1$ or longer ([ASW03] section 1). The Winnowing algorithm will always select the minimum hash value from each window. This is done to increase the chance of matches being caught.

The Winnowing algorithm works with a *guarantee threshold* t . If there exist a shared substring longer than t , a match is guaranteed. The algorithm also works with a *noise threshold* k , meaning that winnowing will not detect matches shorter than k . A small value of k will therefore, potentially, find a lot of false positives². A larger value will find less false positive but might not detect some true positives. More over, a large value of k makes it harder to detect reordering of code, since substrings smaller than k will not be detected. A theoretical and experimental analysis of this trade-off is also given in [ASW03].

2.2.4 MOSS

Measure of Software Similarity³, MOSS, is a free online copy detection tool. Via a Perl script a user can submit software documents for comparing. MOSS returns a URL where the user can find the result of the measure of similarity. MOSS returns how large a percentage it guesses copied. The percentage seems to be calculated in the following way:

$$\left(\frac{\text{number of lines copied}}{\text{number of lines in document}} \right) * 100$$

There has been released a paper about the underlying algorithm Winnowing [ASW03] and, to a part, MOSS. This paper states that it should be possible for the user to change the length of the n -gram and the user should be able to define a window to contain w consecutive hashes of n -grams, it is however not possible to change this setting in MOSS. Parameters available for the user to change are indegree, the language⁴, submitting files or directories and submitting a base file. By submitting a base file the teacher or instructor is able to allow functions predefined in the base file - MOSS will not regard text matching the text in the base file as possible plagiarism.

Like Yap3 MOSS works on the lexical level, on the string of tokens, and seems to change function and variable names in the software before running the algorithms on the documents. In section 6.3 MOSS will be tested and the results discussed. In that section MOSS will also be compared with our tool.

²A false positive is defined as a wrongly identified copied part (false alarm) - a true positive as a truly copied part

³Found at <http://www.cs.berkeley.edu/~aiken/moss.html>

⁴Among others, MOSS currently supports ML, C++, Java, pascal and lips.

2.3 Counter measures and counter counter measures

Counter measures are measures taken against something or someone. When designing copy detection systems with the intention of discovering plagiarism, one has to assume that people will try to circumvent it. In software there are a number of things people can do to try to hide plagiarism - rearrange code, rename variables, etc. For a plagiarism detection software to be successful we need to take these counter measures into consideration - we can do this by implementing counter counter measure. This requires an in-depth analysis of possible counter measures.

2.3.1 Counter measures

People afraid of being caught will try to disguise the plagiarism as well as possible. In [Wha90] Whale pinpoints the following ways to try to hide plagiarism:

- Change comments
- Change data types
- Change identifiers
- Add redundant statements or variables
- Change the structure of selection statements
- Change the order of independent statements
- Combine copied and original program statements

Figure 2: List of counter measures

Less advanced programmers will copy text more directly, often without understanding what they have copied. They will then change the comments to make the program appear different from the original. Students a bit more advanced in the art of programming will be able to change comments, transpose code and change identifier names. Advanced programmers will be able to change data types and change the structure of selection statements which is closer to what we called paraphrasing in the previous section. This is harder and it should not be necessary to plagiarised if you have a good understanding of the problem and the program.

When designing our copy detection tool we should carefully consider how to account for the, in figure 2, stated counter measures.

2.3.2 Counter counter measures

In figure 2 we identified ways in which people will try to bypass plagiarism detection tools. Figure 3 show how one could, if possible, counter these. The counter counter measures are listed in the same order as the counter measures.

Our implementation should as far as possible implement the in figure 3 defined counter

- Remove comments
- Ignore the types of data
- Change all identifiers to the same value - e.g. *V*
- Ignore redundant statements or variables
- Break up the structure of selection statements
- Let transposed code have little influence on the copy degree
- Split up program statements

Figure 3: List of possible counter counter measures

counter measures. A further analysis of counter counter measures will be discussed later in section 4.2.2.

2.4 Copy detection tool - desirable properties

Below is a list of desired properties we think *our* copy detection tool should pose.

Configurable For the tool to be useful for many types of data, it is necessary for it to be configurable. In that way it can be tailored to flag suspicious equalities between documents in different groups of programs, and not just in Standard ML programs.

Not vulnerable to counter measures Our tool should not be vulnerable to counter measures taken against it. It should be able to handle the counter measures defined in section 2.3.1.

Exact output The output provided should be exact in the way that no false positives⁵ should appear. False negatives⁶ should also not occur. A human can eliminate false positives, but false negatives will go undiscovered because these will never be reported in the first place.

Predefined input It should be possible for the user to give the tool a list of allowed routines for the tool to ignore, when detecting copied parts.

Readable output Since a person has to read and evaluate the output for possible plagiarism, the copy detection tool should give output in a clear manner.

We should always have the above stated properties in mind when designing this tool.

⁵Possible plagiarised part flagged, however *no* plagiarism occurred

⁶A plagiarisem part not identified by the system. The part did not seem critical enough, however plagiarism *did* occur.

3 XML Store

XML Store is a distributed value-oriented storage facility for storing Extensible Markup Language (XML) documents. XML is capable of containing simple data, like text and web pages on to more complicated structures like software programs and relational databases. Basically XML Store works in the following way [PP02]:

1. Read XML data from an external source.
2. Serialize the flat file.
3. Parse data to a DVM representation - see figure 5 . DVM allows sharing of nodes within and between XML documents.
4. Unparse and save each node into a value stream and save to disk.

This project will concentrate on item 3. XML Store parses the XML documents in to DVM (Document Value Model) trees. A DVM tree is a directed acyclic graphs (DAG). Thereby XML Store offers a tree structure representation of XML documents. The way the DVM representation works by allowing sharing of nodes, makes it very well suited for hosting a copy detection application, as we will see in section 3.1. An example of XML documents is found in figure 4, an example of a DVM tree is found in [PP02] p. 34 and an example of the DVM version of the, in figure 4 defined, XML documents is found in figure 5, section 3.1.

3.1 XML Store and copy detection

In figure 4 we see two simple XML documents. As we can see a part of the structure of these documents is shared.

```

<Doc1>
  <n1>
    <n3>
      L1
    <n3>
  </n1>
  <n2>
    <n4>
      L2
    </n4>
    <n5>
      L3
    </n5>
  </n2>
</Doc1>

<Doc2>
  <n2>
    <n4>L2</n4>
    <n5>
      L3
    <n7>L4</n7>
  </n5>
  </n2>
  <n6>
    L5
  </n6>
</Doc2>

```

Figure 4: Two XML documents

In figure 5 we see the XML Store representation of these two documents. As we can see XML Store has noticed that a part of both documents is equal. XML Store reacts

to this equality by making the two documents share the equal structure. Thereby XML Store stores the equal data only once. Due to this sharing XML Store does not allow users to manipulate nodes since this would have an effect on more documents. XML Store is value-oriented, nodes cannot be changed, instead a new node is created.

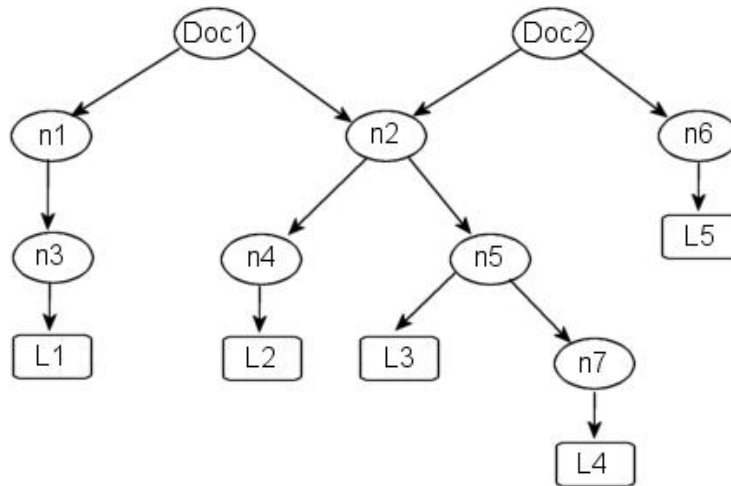


Figure 5: XML Store sharing data

The documents will still be two different documents for the user to request. Any number of documents can share a given node, the subtree(s) of that node has/have to be exactly alike.

In other words, the XML Store system *saves* and *identifies* all identical trees and subtrees. By saving two or more programs with XML Store, we know that structurally identical parts are shared and saved only once. With this knowledge we should effectively be able to use XML Store as host for a copy detection tool. This is one of the ideas behind this project. Also, usage of XML Store allows accumulation and sharing of years of data, because it is persistent and distributed.

Now the task of a copy detection application is to identify these, by XML Store identified, shared nodes. Furthermore, the tool has to use the user given parameters on the detected nodes and their subtrees. In section 4.2.4 we will discuss which parameters a potential user would like to influence.

3.1.1 Benefits

In section 2.2.1 we saw that other copy detection tools do a large amount of work before comparing documents. Also the comparisons take up a great deal of resources because the fingerprints of every document is pairwise compared. By using XML Store as a host we can reduce all the work to almost nothing. XML Store saves documents by saving one node at the time. By doing so it allows sharing of nodes and subtrees. Node values, a node value being the value of a node and its children, are hashed⁷ and a

⁷For having XML Store uses the cryptographic hash function MD5 to hash to a 128-bit number -> incidental hash collisions is only a theoretical possibility.

hash value is returned. If the returned hash value already exists a reference is made to the node with this value, otherwise a new node is created. With this in mind we are not just working with a document's fingerprints, but rather entire documents - *no* shared parts should be able to slip undetected through our tool due to badly selected *n*-grams.

Another clear benefit is that we need only traverse the DVM tree of the documents once to check whether they have shared parts with *any* of the other documents to be compared with. This is clearly better than the pairwise comparison done by other copy detection tools - see section 2.2.1.

3.2 Input

As mentioned in section 3, XML Store takes XML documents as input. We want to input Standard ML programs to perform copy detection on. Other plagiarism detection tools take as input flat text files and perform tokenising routines on the software programs, see section 2.2. However, with XML Store we have the opportunity to maintain the structure of the input programs by parsing them to XML documents in a syntax maintaining way.

In this project we concentrate on Standard ML programs, but the theory discussed in this section is valid for most programming languages. In the process of compiling a program, the compiler, more specifically the parser, will produce an abstract syntax tree (AST) which can be seen as the compiler's internal representation of the program. Since tree structures can easily be transformed into XML it should be possible to make a Standard ML compiler dump XML code while compiling SML programs. By providing input to XML Store in this way, we have the possibility to research whether it adds anything to copy detection to look at the secondary structure of a program, the AST, as to the primary structure of the program, the sequence of tokens. This topic seems to be virtually undiscussed.

By using an already existing SML compiler, we can gain some of the benefits from it. It will automatically remove comments for us. Although one could argue, that if students copied comments it would almost be certain that they had copied code, almost all known copy detection tools remove these [Clo]. This is because even novice students will change comments when plagiarising, whereas students not plagiarising will often keep the comments in code handed out by the teachers, if this is done. Moreover, a parser evaluates and simplifies expressions. A good example of this is given in section 4.1.1, where a student is trying to bypass the detection by redefining an infix operator.

Another good property gained by using ASTs, and later XML documents, when comparing two software files, is that it will be hard for students to rearrange code. This is due to the tree structure. If a student moves a statement from one place in his/her program to another, that student will also move the entire tree of that statement. If he/she e.g. moves `val foo = bar(c, k);`, that statement will be recognised as copied by our tool, since the XML Store representation will remain the same, and the tree will be shared with the original document anyway. This is also done in most other tools, but by using XML Store this is done for "free", whereas other fingerprinting needs to retain fingerprint information. This property will become more clear later in this report.

4 Design

In this section the design of the plagiarism detection tool will be discussed.

We adopt a “*not changing the XML Store implementation*” approach, and will use XML Store as it is - with one exception: the use of weak attributes in XML Store (see section 4.3). In short plagiarism detection on a collection of SML programs will go through these steps (see also figure 6):

1. Compiling, which, besides compiling the programs, provides XML documents of the programs.
2. Saving the XML documents in XML Store.
3. Using traversals of the document trees in XML Store to find subtrees that are copied in different documents.
4. Applying the plagiarism criteria on the nodes found in the above step.
5. Making an HTML document with the results of the plagiarism detection.

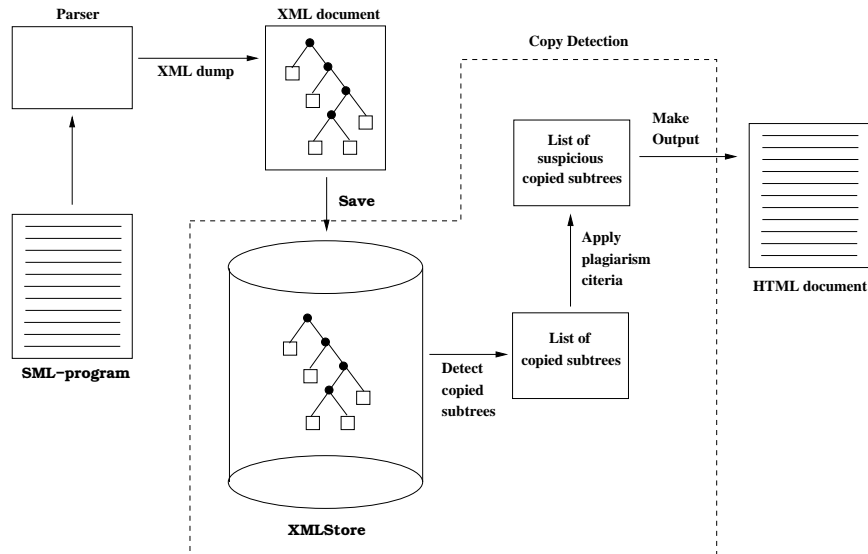


Figure 6: Design

4.1 SML to XML

To be able to dump XML code in an SML compiler, as decided in section 3.2, we needed to find a suitable compiler. After some time of searching we decided to use the MosML⁸ compiler. This compiler was to a large degree chosen because we had access

⁸MosML is a Standard ML implementation and is found at <http://www.dina.dk/~sestoft/mosml.html>

to some expert knowledge about this compiler via the people related to Plan-X⁹ which this project is a part of, mainly Ken Friis Larsen, who also kindly made a prototype to show us how we could implement the idea.

The transformation of SML to XML is to be seen as a front-end to our plagiarism detection tool. It is important that this front-end *in no way* is too closely connected to the rest of the tool, as it should be possible to replace it with front-ends for other types of programming languages. A way to ensure this property is to let the front-end produce XML versions of SML programs and then load these into XML Store and our tool. In this way it will be completely independent of the main tool, and can easily be replaced by another XML producing front-end.

4.1.1 The MosML compiler

The idea is to dump XML code somewhere in the MosML compiler. It could be at one or more levels of the compiling process. The levels of the MosML compiler are described in figure 7. Different levels in the compiler represents different abstraction levels of SML programs.

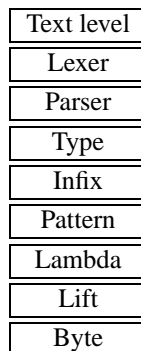


Figure 7: The structure of the MosML compiler

At top level we have the SML-programs represented as text files. At the different levels from the parser we obtain different levels of abstract syntax trees. We must carefully choose at which level we want to generate our XML-code. Do we wish to keep comments? These are already removed at the lexer level (at this point no AST has been produced). Do we want

```
nonfix +;
val x = +(1,2);
```

to be represented the same way as

```
val x = 1+2;
```

If so we might want to generate XML at the infix resolution level. Do we want alpha conversion? - If so, the compiler can do that for us as well. We could then produce

⁹See www.plan-x.org

XML at the lambda level - however this property will not be hard to implement at some of the other levels, as the MosML compiler has identified all identifiers for us. The ultimate plagiarism would be if exactly the same byte code is produced.

4.1.2 Dumping XML

We have decided that the right place in the compiler to produce XML, is at the infix level, see figure 7. When dumping code we need to take some things into consideration. Which information about the SML program do we want to be carried on into our plagiarism detection tool? Is it necessary to carry on all variable names over into the XML documents, and thereby into our plagiarism detection tool? Other copy detection tools (e.g. MOSS and YAP3 [Clo]) rename identifiers anyway, and if we were to follow their example it would be unnecessary. However, in section 2.4 we decided that our tool should have a high degree of configurability, and with that in mind we should let the users make this choice. This is also one of the reasons why we chose the infix level, which gives us the ability to let the user choose whether to change identifiers or not.

In section 2.4 we also weighted readability of our tools output rather high. This was because the output needs to be evaluated by a human, since a human is the only one who can decide if the copied part *is* plagiarism - the tool can only give an indication. In the MosML compiler we have access to locations of the declarations and expressions among others. If we brought these locations with us into the XML documents, we would be able to tell the user exactly where to look in the programs - we could even print out the identified part, for the user quickly to decide whether to take a closer look or not.

By dumping XML in this way, via the compiler, we gain the benefit of letting the user choose whether to replace identifiers and we give the user a better way of tracking back the copied parts to the SML programs, and decide if he/she wants to do more about it. Since compiling SML programs with the MosML compiler virtually takes no time, we can use the entire compiler on SML programs, and let the XML documents be a bi-product of the compiling procedure. It is, however, necessary that the SML files can compile.

4.2 Plagiarism detection

After compilation of the SML documents the XML documents dumped by the compiler are saved by XML Store, which only saves similar subtrees once and instead lets the documents share these subtrees (see section 3).

As for trees an XML Store node can be either a non-leaf (a node with children) or a leaf (a node with no children). In the following the term *node* will be used for non-leaves.

The document trees must, after all documents have been saved, be traversed to find the plagiarism attributes (see section 4.2.1) used for copy detection. Traversal will be discussed further in section 5.2.1.

4.2.1 Plagiarism Attributes

To be able to find any plagiarism between the documents some attributes are needed. These attributes must tell whether a particular node is shared by different documents and something about the sizes of the subtrees, which both contribute to determining

whether there might be instances of copying. A list of these attributes is shown in figure 8 and is described in the following paragraphs.

height (or *maximum depth*) of the subtree. The longest way to a leaf in the subtree of the node. Together with *mindepth* it shows how far from the leafs of the tree the node is, and thus, together with the *nodes*, how large a part of the tree the node “represents”.

minimum depth of the subtree. The shortest way from the node to a leaf in the subtree of the node. See *height*.

number of nodes in subtree. The number includes the node itself, and all leaves as they too contain information relevant for plagiarism. Together with *height* and *minimum depth* it shows how large a part of the tree the node “represents”.

parent list A list of the parents of a node. The length of this can conveniently be used as indegree (see this).

indegree The number of edges into the node. Equal to the length of the *parent list*.

document list A list of all the documents that can reach a particular node. This is needed to determine whether sharing occurs, as a node only shared within the same document tree is not a plagiarism candidate. And for the writing of output (see section 4.3).

Figure 8: Plagiarism attributes

Indegree

For the sharing of subtrees one very important attribute is needed. This is the *indegree*, or sharing, attribute. This attribute tells how many parent nodes this particular node has, that is how many edges¹⁰ that go into this node. A list of the parents of a node could be used to show this. The length of the *parent list* (see further below) will equal *indegree*. Only edges from different documents should be counted when applying the plagiarism criteria, as it cannot be called plagiarism to copy from oneself.

Height, minimum depth and number of nodes

Attributes for showing something about the size of the shared subtree are also required. For this purpose *height*¹¹ and *minimum depth* of the subtree are good to have as the higher the depths of the subtree the larger a part is copied. Also the *number of nodes* in the subtree shows something of the size of the subtree. As two subtrees can have the same *height* and *minimum depth*, but a different number of nodes¹², the depth and nodes together will be a good measure of the size of the subtree as illustrated in figure 9. The two trees have the same *height* and *minimum depth* but a different *number of nodes*. If only the depths are used, the two subtrees will be seen as having the same size which is clearly incorrect. Because nodes say more about the size of the subtree the *number of nodes* is chosen to represent the size of the subtree. This will give the most consistent picture of the sizes of subtrees.

¹⁰As the trees in XML Store are actually DAGs (see section 3).

¹¹The longest way from a node to a leaf in the subtree.

¹²Because the trees are not binary but can have a variable number of children.

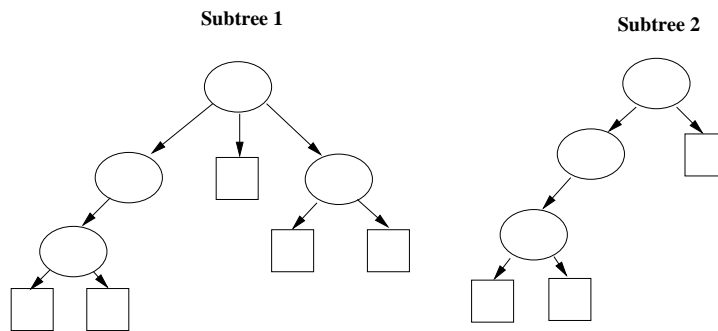


Figure 9: Different sized subtrees

Document list

After finding the copied parts the output is created (see section 4.3). It is necessary to show the user which documents that share which subtree. For this purpose a list of the documents that can reach a particular node is needed. These can then be written to the output when plagiarism detection is complete.

4.2.2 Counter measures

As stated in section 2.3.1 we need to take the student's counter measures into account when applying the criteria and designing the tool. In section 4.1 we saw that, by dumping XML at the right place in the MosML compiler, we could gain some benefits in our fight against counter measures. E.g. the dumping of XML at the infix level helped us avoid not detecting the advanced form of plagiarism we saw described in section 4.1.1 (changing infix to nonfix). In that section we also decide to provide the user with the possibility of changing all identifier names to a constant - e.g. "V". The compiler removes all comments for us, so our system will not be foiled by students changing the comments.

If students add redundant statements or variables this would make their assignments come out as less copied than if not. However, if there were added too many of these statements, more than 5 to 10 percent, it would make the teacher wonder and maybe take a closer look at it anyway. Our tool will account for this, to a degree, since we will be counting all found copied parts and add these, see section 4.3. If a student has copied 56 percent from another and add 10 percent redundant statements, there is a good chance that our tool will still report a copy percent of 51. This is however only valid for a sufficiently small value of the minimum allowed number of nodes in a subtree and minimum allowed depth of that tree. If students add redundant statements within a structure that structure will become broken up, and the two new trees may be below the thresholds, and will not be counted in when computing the percentage and the plagiarism degree.

4.2.3 Public document parts

For the plagiarism detection tool not to return copied parts which are public, i.e. publicly available for use by anyone, as plagiarised, it is necessary to have a way of de-

termining if a shared subtree is a part of the public document. This could be done by setting a maximum boundary of how many documents that can share a subtree before it is counted as a public part. Another, and better, way of doing this is to have a document that define what the public parts look like. In this way the tool will have more control of the detection of public parts. If any of the documents share subtrees in this public document they are not counted as suspicious copying. This could either be as a choice by the user whether to have this public document or not or a standard public document which is always used. We would like our tool to be configurable and therefore the public document must be supplied by the user if he/she chooses to use this feature. The maximum number of documents that can share a node will be used too, as parameter that can be set by the user. In this way there is even more control over which parts that are counted as plagiarism candidates.

4.2.4 Criteria

To determine whether the sharing of a given subtree is a possible plagiarism some criteria, rules that must be satisfied, are needed. These could be:

- Sufficiently large subtrees are shared, e.g. it would not make sense to call a shared leaf plagiarised, as this would be a very small part of the document tree.
- 2 or more documents share a subtree. Trivial, because if only one document shares a subtree the subtree is unique to this document and it is not plagiarism to copy from oneself.
- Whether the shared part is a standard method. If a very large number of trees share a subtree it could be a standard method and is thus not plagiarism or if the subtree is a part of the public document (see section 4.2.3).
- How high up in the tree the node is (if it is high in the tree it is a long equal sequence and thus likely to be plagiarism).

Combinations of the above criteria would be a good schema for plagiarism detection.

The size of a subtree

Because different subtrees can have the same depths, but a different number of nodes (see section 4.2.1 and figure 9) it might be an idea to have a way of representing the size of a subtree. This could be some weight between the depths and the number of nodes, for example as the *number of nodes* divided by the difference between the *height* and *minimum depth*. This might give false plagiarism hits though, as a very large subtree might have the same subtree boundary as a very small one. Instead the *number of nodes* in the subtree will be used as the size of the subtree. This is more consistent although it does not tell anything about the depth of the subtree. For this the *depths* are used.

Determining which nodes are plagiarised

The first step in determining whether a particular node is a candidate for plagiarism is to see what the *indegree* for the node is. If this is ≥ 2 it is a candidate for plagiarism, again because it is not plagiarism to copy from oneself. In this way many nodes will be cleared of suspicion as their indegrees are 1. The next thing to do is to check if the node is a node in the public document. This will root out more nodes that are not plagiarism. Then the plagiarism criteria will be applied to the nodes that are still suspicious. The criteria applied will be:

- Does the number of documents that shares this node lie within the indegree criteria?
- Does the subtree size, i.e. the number of nodes in the subtree, lie within the node criteria?
- Does the subtree depths lie within the depth criteria?

If all of these are observed for a node, the node is taken as being possible plagiarism and sent to the output for manual inspection.

User defined input

It must be possible for the user to control the criteria for plagiarism detection by setting plagiarism parameters when starting the program. In this way the user will have an amount of control over the measure of the copying detected. The parameters are chosen for the purpose of giving the user as much control of the detection as possible, and therefore the parameters closely match the plagiarism attributes. The parameters that can be set is boundaries for the depths (*height* and *minimum depths* of the subtrees), the nodes (*maximum* and *minimum number of nodes* of the subtrees) and *maximum indegree* and also the public document (see section 4.2.3). The depths and number of nodes are chosen to control the sizes of the subtrees detected, and indegree to limit the number of documents that can share a subtree. Finding some good guidelines for these parameters is described in section 6.2.

4.3 Output

One of the properties we would like our tool to have (see section 2.4) is that the result of the plagiarism detection must be shown in a easy readable way so it is easy to find and compare the documents which the program has detected have possible plagiarism. Preferably as the part that is copied followed by which documents that contains the copied part and where in the documents they occur. For this the locations provided by the SML compiler are used (see section 4.1).

Weak attributes in XML Store

In order to be able to make usable output for the user, some way of showing the copied part is needed. The SML parser (see section 4.1) can provide the location of the subtrees in length of characters. These are stored in an attribute in the XML tag which the XML Store should *not* look at when “comparing” two nodes¹³, e.g. the tag `<tag1 location=(24,115)>` should be equal to `<tag1 location=(107,198)>` if their subtrees are equal. This results in the fact that the location should be a weak attribute, which is an attribute that will not be compared when XML Store compares and saves nodes, otherwise sharing would not be detected, as the nodes will then be different if only in the attributes. The weak attribute will ensure that two identical subtrees, but for the attributes in the XML tag, will be counted equal, and only stored once, thereby making it possible to detect the copy. As it is, XML Store does not have any weak attributes, so this must be changed slightly. This will not be done by changing the functionality of XML Store, but rather adding extra functionality¹⁴. This will be as a programming choice whether to use weak attributes or not.

¹³When hashing nodes and later comparing two nodes to see if they are equal.

¹⁴Following the “*not changing the XML Store Implementation*” approach.

One disadvantage of the weak attributes is that it will not be possible to retrieve the location for all documents, as the first time the node is stored the node's location in the first document that contains this node is stored and the location in another document that refers to this node would not be stored. To use the above tag example again: The first document stores the node `<tag1 location=(24,115)>` which saves the location (24,115), when another document with the `<tag1 location=(107,198)>` is compared to this node only the tag `tag1` is compared and found equal to the first. Thus the location of the node in the second document, (107,198), is lost.

Plagiarism degree

The idea behind the plagiarism degree is to give the user some way of knowing the size and likelihood of the plagiarism. The higher the plagiarism degree, the more likely plagiarism of that part is. The *plagiarism degree* (see figure 11) will be a number that shows how likely plagiarism the copied part is seen as. This could be the number of nodes in the subtree (the size of the subtree, see section 4.2.4) or some computed measure of the *plagiarism degree*. If the number of nodes in the subtree is given as *plagiarism degree* together with the number of nodes in the different documents (the size of the document) and the percentage of the document the copied part is, the user will have a good idea of how copied the part is.

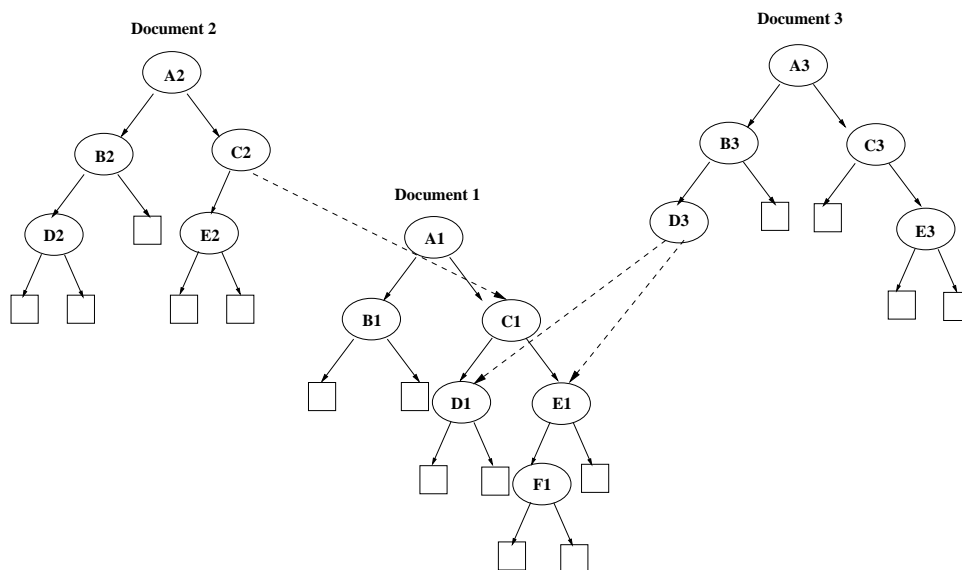


Figure 10: Plagiarism degree

The idea is illustrated in figure 10. In both document 2 and 3 a node refers to the same subtree in document 1. Document 2's node **C2** refers to node **C1** in document 1 and document 3's node **D3** refers to the two children, **D1** and **E1**, of **C1**. The copying done by document 2 is more serious than the copying done by document 3 as the copying is higher in the tree. Here the plagiarism degree for document 2 would be 9, the number of nodes in the copied subtree (**C1** and down), and the plagiarism degree for document 3 would be 3 and 5, because it copies 2 subtrees in document 1. Thus the plagiarism

degree is much larger where plagiarism is done from a “higher” subtree, than for the “lower” subtrees, even if the difference in depth is only one, as in the above example. As the percentage of the copied part of the document (see figure 11) will be computed from the nodes, these two things are roughly the same.

Layout

The output generated by the plagiarism detection tool, could either be written to the command window or to a file. For the user of the tool the file would probably be the best solution, as this gives the user the possibility of returning to the results at some later time, without having to run the tool again, with the same set of parameters as before.

A good possible file format is HTML. This format gives the possibility of showing the results in a clear way. The layout of the document could be something like the layout shown in figure 11.

The following part, with plagiarism degree 9, seems to be copied in these documents:

Document 1: doc1.sml	Nodes in document: 180	percentage copied: 5.0%
Document 2: doc2.sml	Nodes in document: 212	percentage copied: 4.25%
Document 3: doc3.sml	Nodes in document: 324	percentage copied: 2.78%

```
fun divMod (a,b) = (a div b, a mod b);
```

The following part, with plagiarism degree 17, seems to be copied in these documents:

Document 1: doc1.sml	Nodes in document: 180	percentage copied: 9.44%
Document 2: doc4.sml	Nodes in document: 145	percentage copied: 11.72%
Document 3: doc5.sml	Nodes in document: 879	percentage copied: 1.93%

```
fun gentag (0,x) = []
  | gentag (n,x) = x::(gentag(n-1,x));
```

Figure 11: Layout of HTML output

The output will also include the total percentage of a document that is copied.

5 Implementation

In this section the implementation of the design of the plagiarism detection tool (see section 4) is described. The source code can be found as appendix C.2.

5.1 Front-end

This section contains a short description of the implementation of the SML front-end. The first draft of this implementation was made by Ken Friis Larsen. We expanded it to support most SML productions, bring over locations to the XML files and to produce parsable XML output - support for structures and signatures has not been implemented.

In section 4.1.2 we decided to implement the renaming of variable names into the front-end. We do this by prompting the user to decide whether to replace all strings, integers, function names, variable names etc. by a constant value when dumping the XML, all relevant code can be found as appendix C.1. To produce XML we use the MosML Msp module¹⁵. This module allows us to add tags to SML statements. `Msp.Mark1 t ws` generates `<t>ws</t>` as a `wseq`. Furthermore it is possible to add attributes to the tags by using the function `Msp.Mark1a - wseq` - `wseq` is the type of efficiently concatenable word sequences. Attributes should carry the location of the SML statements over into the XML file to enable us to print out the copied parts to the user. These can be added to the tags by using the function `Msp.Mark1a a t ws` where `a` is the attribute, `t` is the tag and `wseq` is the word sequence. `Msp.Mark1 a t ws` generates `<t a>ws</t>`.

5.2 Copy detection

In this section the implementation of the copy detection done on the XML Store trees is described. Among this traversing the trees and storing of the attributes.

5.2.1 Traversing trees using XML Store

As mentioned before the trees built by XML Store are actually directed acyclic graphs (DAGs) and equal parts are only stored once. If other nodes have this equal part as part of their subtrees they refer to the root of the part they copy (see section 3). As the traversal of the XML Store DAGs is done as for trees, the term *tree* is used as synonym for DAG.

The interface for traversing trees in XML Store is the `Node` and `ChildNodes` classes [PP02, XML]. All nodes contain a list of their children. The `Node` objects of these children are accessed via this list.

In order to detect plagiarism in the document collection full traversals of the document trees are needed as every node is a possible candidate for plagiarism. In this way all nodes will be visited at least once.

Different Tree Traversals

There are various ways in which to traverse trees. The most common ones are *pre-order*, *in-order*, and *post-order*. Other useful traversals are *breadth-first* and *depth-first*.

¹⁵The MosML library can be found at <http://www.dina.dk/~sestoft/mosmlib/>

If we want to write out the document *pre-order* would be chosen as this would print the document in the right order.

The nodes in XML Store will be traversed for each document tree in turn as this will give the best computation of the node parameters. Which traversals to use where will be discussed where appropriate.

5.2.2 Computing the plagiarism attributes

As discussed in section 4.2.1, each node needs special plagiarism attributes used to store variables that indicate whether the node is possible plagiarism. The attributes needed are shown in figure 8.

Max and mindepth and nodes

To compute *maxdepth*, *mindepth* and *nodes* (*height*, *minimum depth* and *number of nodes* in figure 8 section 4.2.1) a *depth-first traversal* is needed as only the depths and number of nodes in the subtree of leaves are known in advance (*depth* = 0 and *nodes* = 1), the *DFTraversal* class implements this. The depth of a node which has children (i.e. is not a leaf) is computed by finding *max*(*maxdepth* of children + 1) ,i.e. finding the child with the highest *maxdepth* and adding one (the height from a node to its child). The number of nodes in the subtree is computed by adding all the node counts of the children together and adding one (the current node).

Parentlist and indegree

The parent must be remembered when traversing down the tree to compute the *parentlist* (the *parent list* in figure 8 section 4.2.1) and thereby also the *indegree*, as this is just the size of *parentlist*, of the nodes. At each node the parent (the *ValueReference* of the parent as the *ValueReferences* are unique for each node) is added to the *parentlist* whose size then increases with one.

Doclist

As for *parentlist* all the *doclist* (the *document list* in figure 8 section 4.2.1) requires is to remember which document is currently being traversed (the *ValueReference* of the document root) and for each node add it to the *doclist*.

Other attributes needed for the traversal

Another attribute needed is the attribute for whether a node is part of the public document. This is obvious to implement as a boolean value to be checked when applying the criteria . The public document is traversed first to set this attribute. A *pre-order traversal* is used as it is only the public document attribute that is set. The rest of the attributes will be set for the nodes when they are referred to from the documents being examined for copying.

5.2.3 Traversing the document trees

Traversal of the document trees is done with the XML Store interface provided to traverse trees (see section 5.2.1) by the *DFTraversal* and *PreOrderTraversal* classes. Because XML Store saves equal subtrees only once and any other use of this subtree

has a reference to the equal subtree, it is only necessary to compute the depths of a shared subtree once. Therefore a method to determine if a subtree has been computed must be used. For this it is only necessary to look at the child's `maxdepth` or `mindepth` parameter and see if it is the initial value or higher. For this it is chosen to examine `maxdepth` as both `maxdepth` and `mindepth` are set at the same time and if `maxdepth` is set it will always be \geq `mindepth`. Also a lookup in the table of all the nodes is done because if it is in this list it will already have been set. The examining of the `maxdepth` value is done because if the node is already in the list it could be a public document part not previously traversed. The public part must then be traversed to set the `maxdepth`, `mindepth` and `nodes` so that the currently traversed document will compute its size correct. The initial value for both `maxdepth` and `mindepth` is -1 as a depth can never be negative and it will hence be possible to use it as a means of checking whether the node has been visited as described above. If `maxdepth` and `mindepth` have already been set only the `doclist` and the `parentlist` will have to be computed¹⁶. Therefore the subtree where the depths and nodes have already been set can be traversed in *pre-order*.

5.2.4 Storing the plagiarism attributes

The attributes must be stored somewhere if they are to be accessed for the task of copy detection. The first thing that comes to mind is to store them in the `Node` objects, but as a new `Node` object is given, when getting `Nodes` from the `ChildNodes`, the stored attributes would be lost when the node is reached again. Therefore, following our “*not changing the XML Store implementation*” policy, the attributes are stored in an attribute object unique for each node, called `PlagAttributes`. The uniqueness comes from storing the attributes with the same `ValueReference` as the `Node` it belongs to since the `ValueReference` is unique to each node.

For the storing of the `doclist` in the `PlagAttributes` object a `Hashtable` will be superior to an `ArrayList` as it is fast to check if a certain object is in the list. An `Integer` object of the hashcode of the current document's `ValueReference` will be used as key for storage and retrieval of the document, and the hashcode of `ValueReference` of the parent as the key for the `parentlist`.

To access the `PlagAttributes` objects a place to store these is needed, and for this a table would be good. As it is not known in advance how many nodes there are in the trees an `ArrayList` or a `Hashtable` is a good place to store the attribute objects. But because the attributes must be accessed easily, and fast, a `Hashtable` is again the best solution as an object can then be found without searching through the whole table as would be the case with an `ArrayList` if the index where the object is stored is not known. An `Integer` object of the `ValueReference` of the `Node` will be used as key for the storage and retrieval of the `PlagAttributes` from the `Hashtable` called `nodelist`.

5.2.5 Public document parts

To eliminate nodes from suspicion by checking whether they are part of the public document, the program needs an XML document called `publicdocument.xml` which specifies what a public part is. This document must be in the same directory as the

¹⁶The number of nodes will have been set at the same time as `maxdepth` and `mindepth`

plagiarism tool, which checks if this is the case. If there is no such document in the directory the tool is run without the public document. This means that no nodes will be eliminated of suspicion of plagiarism by being part of the public document. Instead all that eliminates this is the `maxindegree` parameter set by the user, which gives the maximum number of documents that can copy a subtree before it is a public part (see section 4.2.4).

If a public document exists it is traversed first to create the `PlagAttributes` object of the nodes in the public document and set the `isstandardmethod` variable in these `PlagAttributes` object to `true`, the `SetStandardMethodsAttr` class (see appendix C.2.3). Only `isstandardmethod` is set at this time as the rest of the parameters are only needed for nodes in the documents that are to be examined for copy detection. Therefore it is necessary not only to check if the `PlagAttributes` of a node is in the table of plagiarism attributes, but also whether the depth has been set (see section 5.2.3).

5.2.6 Getting the user defined parameters

The parameters the user can set (see section 4.2.4) are implemented by prompting the user for the values of these parameters. The `getNumber` method in the `ReadPlagParams` is used to read what the user types in the command window. This method is highly inspired by Code 12.17 p. 361 in [Bar00], which has been modified to suit our need of reading from the command line instead of a file. The parameters are obtained in this way, as we are then assured to get the parameters in the right order. Also the user is sure what parameter he/she is setting as it is easy to forget in what order the input must be set. Setting the parameters in the wrong order could affect the detection negatively and produce unusable output. The parameters are stored for use by the `FindParameter` class, which apply the criteria to find shared nodes (see section 5.2.7).

5.2.7 Applying the criteria

When all the document trees have been traversed the criteria for whether a node is copied is applied to the `PlagAttributes` objects in `nodelist` (the list of the `PlagAttributes` for all the nodes). Nodes are cleared of suspicion by the criteria described in section 4.2.4. The nodes that are found to be copied are stored in an `ArrayList` called `plagnodelist` which, after ended criteria application, is passed on to be written out by the `MakeOutput` class (see appendix C.2.10). `nodelist` is kept as it is needed by `MakeOutput`.

As the `indegree`, the length of the `parentlist` (see section 5.2.2), can include parents which are inside the same document, the method `isDocumentsharing` of the `FindPlagiarism` class (see appendix C.2.6), ensures that the node is only counted as copied if at least one of its parents has a `documentlist` size that are ≥ 2 or \leq the `maxindegree` set by the user.

5.3 Measure of similarity

The plagiarism degree, the percentage the copied part is of the whole document and how many nodes the documents consist of, and the total percentages copied of the documents are shown in the HTML page, for the user to have a good idea of how much

copying is done for one document and how large a part the copied part is of the document (see section 4.3).

The plagiarism degree is the number of nodes in the copied part (i.e. the size of the subtree), as this will show a relatively large difference when a document is copying a subtree from one level higher than another document (see section 4.3). The number of nodes is obtained from the `PlagAttributes` object belonging to the root of the subtree.

The percentage a copied part is of a document sharing that part, is computed by

$$\left(\frac{\text{number of nodes in the shared subtree}}{\text{number of nodes in the whole document}} \right) * 100$$

The number of nodes in the whole document is obtained by getting the `ValueReference` of the root node of the object and getting its `PlagAttributes` object, where the number of nodes in the whole document is saved, from the `odelist`.

The total percentage copied of a document is shown at the top of the HTML page. These percents are computed by the `setDocpercentage` in the `MakeOutput` class, which test whether the percent a copied part is of a document should be added to the total percentage of the document. The `setDocpercentage` method finds the `doclists` of the parents which contain the document, and if just one of these `doclists` is ≥ 1 , the percent is added to the total percentage of the document. This is done because if the percent is just naively added, there is a danger of adding an extra percent to a document that shares the subtree from a higher level. This could in some cases give a total percentage larger than 100, which does not make sense.

5.4 Output

In section 4.3 we decided that the user should be presented to the actual code which might have been plagiarised. We also decided that the output should be persistent to enable the user to return to the output at a later time. To do this we thought that writing output to a HTML file would be a good idea. In this way the output would be both persistent, easy accessible and it should be possible to print out the copied code.

To print out the copied code from the SML files we need to know three things (1) where the files are stored (2) where to look in the files (3) the name of the file. The front-end provides both locations and names of the files as attributes. These are stored as attributes and can therefore be retrieved with the `XML Store` method for retrieving attributes from nodes. The name can be retrieved from the root node of the document while the location can be retrieved by retrieving the location attribute from the shared node. These actions are taken from within the `MakeOutput` class. After retrieving the name of the document `MakeOutput` will ask an instance of the class `printLocation` to return a string containing the code between the two locations. The `printLocation` class' method `pLocation` needs the location, as a sting on the form "(loc1,loc2)", where `loc1` and `loc2` are integers representing character numbers in the SML file. `pLocation` will then return all code as string between these two character numbers. `MakeOutput` modifies this string and adds information to it, i.e. subtree size, percentage copied, document name. The modified string is then appended to a HTML page by calling the method `appendHTML` on an instance of the `toHTML` class. `MakeOutput` will

call this method as many times as the number of plagiarised parts. When all parts have been appended to the HTML page `MakeOutput` will call the method `writeFile` on the `toHTML` object. The `toHTML` object will react to this by writing the complete page to disk. The output will be saved to a file named *output.htm*. A sample of the output file is found in figure 11 section 4.3.

As to where to find the SML files we have chosen that the user should always place these in a directory called `SMLdocs` relative to where the source code is placed.

5.4.1 Weak attributes in XML Store

A definition of weak attributes is found in section 4.3, where the need for these is also discussed.

We have decided not to change anything in the XML Store system, it is therefore our task to make the introduction of weak attributes an addition rather than a change.

The addition is done in the following way. A class containing a global boolean variable is created. Before updating the hash value with the attributes XML Store will check to see if this global boolean is set to true; if so the value of the attributes will have no influence on the value of the `MessageDigest` object. The code in file `DVMImmNode` is thus changed from

```
md.update(battr);
```

to

```
if(!GlobalSettings.globalWeakAttributes) {  
md.update(battr);  
}
```

where `GlobalSettings.globalWeakAttributes` is the boolean value indication whether weak attributes are wanted, `battr` is a byte array of the attribute and `md` is the `MessageDigest` object. The default value of `GlobalSettings.globalWeakAttributes` is false. For further information about the `DVMImmNode` class we refer to [PP02].

6 Evaluation

This section will describe the evaluation of our developed tool. First the parameters will be tuned to match SML programs, then a comparison of the results with the results of MOSS will be carried out.

6.1 Strategy

Our evaluation strategy will be as follows:

1. Tuning of parameters to SML programs.
2. Large scale experiments on more unknown assignments - preferably the same assignments if a large enough amount is available.
3. Try the same large scale experiment with MOSS.
4. Experiment on known files, where the counter measures defined in section 2 is applied to plagiarised programs.
5. Do 4 on the same collection of programs with MOSS.

The evaluation will start with the tuning of parameters. We want the parameters to be set optimal for SML programs. More about this topic is found in section 6.2.

We have not received as many assignments from the students at this departments, as we had hoped. A really large scale experiment will have to be carried out by the instructors on received assignments. We have however received some and will try our copy detection tool on these and compare the results with MOSS' results for the same input files.

6.2 Tuning of parameters

Our tool accepts some parameters. The most important ones are `indegree` (`indegree`), `height` (`maxnodes`) of the subtree and minimum number of nodes in the copied subtree. These are the parameters we will try to adjust. The reason for tuning these parameters are (1) to suppress noise (2) to ignore frequently used routines. Noise is copied parts too small to be plagiarism, and therefore these parts should not be presented to the user of the tool. E.g. we are not interested in the fact that the reserved word `ANDALSO` often appear in the code. In our implementation we have chosen to ignore leafs when searching for copied parts. It is possible for the user to take this a step further by adjusting the minimum number of nodes parameter. By doing so even more results will be sorted from the output. Lets take a look at an example - the SML statement `val i = 1;` will evaluate to the XML structure displayed in figure 12.

As we can see in figure 12, this small SML statement generates a fairly large tree. Actually the `VALdec` tree consist of, all in all, 6 nodes and three leaves. The height of the tree is 4. The SML statement `fun med_i x = foldr (fn (a,b) => b orelse a=x) false;`¹⁷ will generate a tree with 32 nodes and a depth of 17. These observations should give an indication of which values for `mindepth` and `minnodes` to start tuning with. We could argue that simple statements like `val i = 1;` should be ignored,

¹⁷XML version attached as appendix A

```

<VALdec location="(0,9)">
  <NONRECURSIVE>
    <ValBind>
      <VARpat location="(4,5)">i</VARpat>
      <SCONexp location="(8,9)">
        <INTscon>1</INTscon>
      </SCONexp>
    </ValBind>
  </NONRECURSIVE>
  <RECURSIVE></RECURSIVE>
</VALdec>

```

Figure 12: The generated XML structure of `val i = 1`

since the result would be a vast amount of output for the user to go through. Also we would like to discover plagiarism the size of the last SML statement (`fun med_i`) - but not much smaller. By this, it would be a good idea to start tuning with `minnodes = 25` and `mindepth = 10`. There are some overlap between these two parameters and just tuning the `minnodes` further, would be a good start.

Earlier in this section noise was mentioned. Noise is not just trees with a small number of nodes but noise could also be frequently used functions (like the above `foldr`). A way to avoid identification of these parts, again to spare the user from analysing too much output, is to set the parameter `indegree` to a low value. In the Perl script provided by MOSS, to run MOSS, they suggest an `indegree` value no higher than three or four. This sounds very reasonable, as a higher `indegree` would probably produce much output for a larger collection of programs. We will suggest the use of and use an `indegree` value of three. Copied parts with an `indegree` below four is likely to be a truly copied part.

6.2.1 The tuning

We will produce some SML files with known content. The SML files will be run with our detection tool and the number of hits generated for each parameter value of `minnodes`. `minnodes` will iteratively be increased by 5 until a good level is found. The results is shown in the table 1 and the actual SML files can be found attached to this report as appendix B.1.

The declarations in the files have been shuffled to avoid detecting too large parts. When parsing the programs we have made use of alpha conversion. The programs contain 898 nodes each. As we see in table 1 we have eradicated all but one part which should not, in our opinion, be counted as plagiarism. The problem is that lists in SML produce a huge XML structure. This SML statement `val numerals2 = [(1000, "M"), (900, "CM"), (500, "D"), (400, "CD"), (100, "C"), (90, "XC"), (50, "L"), (40, "XL"), (10, "X"), (9, "IX"), (5, "V"), (4, "IV"), (1, "I")];` alone produces a tree consisting of 255 nodes! And this is indeed the statement which is the fifth part where we only wanted four parts to be large enough to be counted in.

If we take a closer look at lists, via the generated XML files, we can see that not only do

Tuning of parameters					
minnodes	file1	file2	file3	hits	plag. parts
25	tun1	tun2	tun3	11	4
30	tun1	tun2	tun3	8	4
35	tun1	tun2	tun3	6	4
40	tun1	tun2	tun3	6	4
45	tun1	tun2	tun3	6	4
50	tun1	tun2	tun3	5	4

Table 1: The parameter minnodes iteratively being increased. First column shows the current setting of the parameter minnodes. Column five shows the number of reported possible plagiarised parts. Column six shows the number of parts we think should be counted in.

Tuning of parameters					
maxdepth	file1	file2	file3	hits	list in?
45	tun1	tun2	tun3	11	yes
40	tun1	tun2	tun3	8	yes
35	tun1	tun2	tun3	6	yes
30	tun1	tun2	tun3	6	no
25	tun1	tun2	tun3	6	no

Table 2: The parameter maxdepth iteratively being decreased

lists generate structures with many nodes, they also generate very deep structures. So if we do not want for lists to be taken into consideration we could focus on the maxdepth parameter as well. Nested structure like let and if-then-else do not produced such deep structures. A function (testdepth - see appendix B.2) contains a let-statement with an if-then-else expression nested four deep - the height of this tree is 20.

In table 2 we can see that we have succeeded in removing the list, when the parameter maxdepth was given the value 30 - so it disappeared somewhere between 30 and 35. In our opinion list does not bring much to copy detection - we therefore recommend that users give maxdepth a value no higher than 30. If we combine what we have learned by tuning these two parameters we could run the test again with parameters maxdepth = 30 and minnodes set to 50. The result is shown in table 3.

Tuning of parameters						
minnodes	maxdepth	file1	file2	file3	hits	plag. parts
50	35	tun1	tun2	tun3	4	4

Table 3: The result with parameters set to the optimal

Even though we here have used the values 35 and 50 we suggest that users use a smaller value for minnodes, e.g. no higher than 35. This is because we do not want to leave out

any plagiarised parts. It is better to have one possible plagiarised part too many than one too few; as long as it does not disturb the identification of plagiarism process too much. It may also make it easier to cheat the system by adding redundant statements to the programs if a too high value of `minnodes` is chosen, see section 4.2.2 for a discussion of counter measures.

In our implementation we have made it possible for the user to change the values `mindepth` and `maxnodes` - the minimum allowed depth and the max allowed number of nodes. We think that these values should always be set to 1 and ∞ . If `maxnodes` is set to a lower value, we loose the possibility of detecting plagiarism larger than that. We do not think that changing `mindepth` when working with SML programs will make us gain much. When we have chosen to include these parameters it is because we in section 2.4 decided to make our tool configurable to a high degree. Furthermore, we want the tool to be useful for other types of programs, and what that bring of needs we cannot know.

6.3 The evaluation

First we will run our tool on a collection of 18 XML versions of SML programs. The programs are a collection of small weekly assignments made by students at the Department of Computer Science, Copenhagen University. Almost every assignment exists in four different versions. Two of the versions have been prepared by regular students at the department; the last two versions have been composed by the instructors of the relevant course. We will follow our own recommendation from section 6.2.1 and user parameters `minnodes = 35`, `maxdepth = 50`, `mindepth = 1` and `maxnodes = ∞` . The result of this is shown in table 4.

The results shown in table 4 have been picked by random from the entire output to be compared with MOSS. The results from our tool, PDT, have been added up to get a result comparable with MOSS¹⁸. As we can see the results match pretty well. The percentage does not match that well, but one has to keep in mind that these percentages are calculated in two different ways (number of nodes vs. number of lines). Due to the fact that we only write the output from one of the compared SML files, it is very hard to do a direct comparison with MOSS. However, if one spend the time and put together the jigsaw it can be seen that MOSS and PDT identifies the same pieces most of the time. Our tool identifies the parts in smaller pieces compared with MOSS, but when added its basically the same.

For some reason MOSS does not find the equal parts in files `uge36_3.sml` and `uge36_5.sml`. The missed statement in `uge36_3.sml` looks like this `val h=fn (x,y)=>Math.sqrt(x*x+y*y);` and in `uge36_5.sml` like `val h = fn (x,y) => Math.sqrt (x * x + y * y)` - besides some spaces it is the same. Our initial guess was that the statement is too small to be counted in by MOSS, but when we submitted `val x = 2 + 4;` and `val x = 2 + 4` (without the `;`) a 85% match was reported. Since we do not know the size of n -grams in MOSS and the window size for SML programs we cannot say whether MOSS guarantees to detect copying within statements of the above size (see section 2.2.3). We do not think so.

¹⁸The full output from PDT can be found at <http://www.plan-x.org/projects/plagiarism/eva01.html>. The full output from MOSS can be found at <http://moss.cs.berkeley.edu/~moss/results/452823894/>

uge39_5		uge39_6	
PDT	MOSS	PDT	MOSS
10%	14%	14%	17%

uge39		uge39_5	
PDT	MOSS	PDT	MOSS
34%	36%	14%	13%

uge37_5		uge36_6	
PDT	MOSS	PDT	MOSS
38%	35%	52%	39%

uge38		uge38_6	
PDT	MOSS	PDT	MOSS
23%	30%	18%	13%

uge38		uge38_5	
PDT	MOSS	PDT	MOSS
16%	26%	15%	30%

uge36_3		uge36_6	
PDT	MOSS	PDT	MOSS
32%	NA	11%	NA

uge43_3		uge39_5	
PDT	MOSS	PDT	MOSS
15%	13%	9%	8%

Table 4: File uge39 has 34% in common with file uge39_5, which again has 14% of all of its content in common with uge39. For the same two files MOSS reports 36% and 13% respectively.

Out next test will be with some larger files. These programs are between 36 and 800 lines long. The result is shown in table 5. Again, after a closer look and comparison of the results¹⁹, it is seen that MOSS and PDT finds the same plagiarised parts. Our tool has a tendency to, in a few cases, skip a part of a declaration. E.g. our tool reports that the below part is copied in files Compiler02 and Compiler03.

```

let
    val t1 = "a" ^ newName ()
    val c1 = compileExp e1 table t1
    val (c2, regs) = compileExpList es table
in
    (c1@c2, t1::regs)
end

```

MOSS reports the following:

```

and compileExpList [] table = ([],[])
  | compileExpList (e1::es) table =
    let

```

¹⁹The full output from PDT can be found at <http://www.plan-x.org/projects/plagiarism/eva02.html>. The full output from MOSS can be found at <http://moss.cs.berkeley.edu/~moss/results/605883823/>

Compiler03		Compiler02	
PDT	MOSS	PDT	MOSS
37%	65%	41%	59%
Compiler		Compiler03	
PDT	MOSS	PDT	MOSS
3%	1%	6%	3%
Compiler		Compiler02	
PDT	MOSS	PDT	MOSS
1 %	0%	2%	1%
Limba		Funny	
PDT	MOSS	PDT	MOSS
17%	14%	22%	13%
Type		Interpreter	
PDT	MOSS	PDT	MOSS
2%	5%	3%	8%

Table 5: Result of the test with large files. The equality between Limba and Funny had to be read from std. out, since no location was given and they therefore no output could be written to the output file for that part. Again we see that the results between MOSS and PDT are comparable

```

val t1 = "a" ^ newName ()
val c1 = compileExp e1 table t1
val (c2, regs) = compileExpList es table
in
  (c1@c2, t1::regs)
end

```

6.4 Counter measures

In this section we will take a look at how our tool reacts to counter measures taken against it, we will also take a look at how MOSS reacts to the same counter measures. We will apply the counter measures to a simple known file, tun1.sml (see appendix B.1). A copy of tun1.sml is made and counter measures are applied to it, then the original file is compared to the changed copy and the result is evaluated. Without any counter measures applied and the parameters set to `minnodes = 30` and `maxdepth = 35` a match of 86.05% is reported by PDT - a match of 98 % is reported by MOSS.

As it is seen in table 6 the *renaming* of identifiers has no influence on either PDT or MOSS. If code is *transposed* this has no influence on PDT, however MOSS only reports the large transposed parts. This is probably because these other parts are below the, in section 2.2.3, mentioned guarantee threshold. The damage is not that bad since all larger functions are still captured by MOSS. By adding *redundant* statements our system can be foiled. This is because some of the previously identified subtrees is split up and is now below the `minnodes` threshold. And indeed, if the `minnodes` value is changed to 10 a 90% match is found - this is however up to the user to choose the wanted value. To MOSS this has little effect on the identified parts.

Counter measures		
	PDT	MOSS
Renaming of variables.	86.05%	98%
Transposed code	86.05%	58%
Redundant statements	68.68%	90%
Fixtivity	64.79%	39%
Selection statements	63.77%	66%
Datatypes	19.26%	31%

Table 6: The counter measures defined in section 2.3.1 applied the `tun1.sml` program. Fixtivity is defined as changing the status of an infix or nonfix operator - e.g. changing `+` from being an infix operator to a nonfix operator

By changing the *fixtivity* MOSS can be foiled but our system cannot to the same degree. MOSS reports a couple of places but misses two large functions. PDT reports the same places, where the infix has been changed to nonfix, as well as the two large functions. Two functions are for some reason missed by our system - we think this is because of how the infix resolution is carried out in the MosML compiler. As for the changing of selection statements our tool misses all but one place. MOSS seems to report places where e.g. `<` has been changed to `>` and so one, but has a hard time identifying that the `then`-part has been shifted with the `else`-part.

None of the systems works very well when changing datatypes. In `tun1.sml` we have changed all `ints` to `reals`. MOSS fails to find any of the changed parts and so does our system. The large percent difference is due to the fact that the list is not found when running this test - where in MOSS the list is only a few lines it consists of 255 of the 898 nodes in our our representation as a tree. MOSS and PDT reports the same two places copied in this case.

6.5 Conclusion of the evaluation

We have seen that our tool is able to find and flag parts copied between two or more SML files. We have tuned the parameters to what we think is optimal for SML programs. We have tested our tool against the copy detection tool MOSS. The results have shown that in many case MOSS and PDT find the same places and the same amount of shared data. MOSS and PDT have each their forces when counter measures are applied. PDT works well when variables are renamed, code is transposed and the fixtivity have been changed. MOSS works well when renaming of variables and addition of redundant statements are applied. Non of the systems works well when the structure of selection statements and the data types are changed. A more in-depth conclusion of the tool can be found in section 7.

7 Conclusions

The developed tool is able to find possible plagiarism in SML files. By adjusting the parameters the user can decide how large pieces of shared parts he/she wants to find. The presence of these parameters should also make it possible to replace the SML front-end with a front-end for another programming language, and then readjust the parameters to find the set of optimal parameters for that language.

PDT, our developed plagiarism detection tool, is able to identify the in XML Store shared subtrees and apply the user given parameters on them to sort out output. The result is given in a HTML-file for the user to check now or at a later time. This file is not as readable as we had hoped for. This is due to the fact that the implementation only allows us to store one location - the location from the first file. It is therefore hard for the user to compare the contents of the files, but by printing out the copied functions from one of the files the user is given a good indication.

A part of this project was to investigate whether it brings anything to the field of copy detection to compare program structures. It does not look that way, but then again the number of files we have had for testing were insufficient to do a really large scale test.

The comparison with MOSS showed us that the tool was able to detect the same amount of plagiarism as MOSS as well as the same parts. When counter measures was applied MOSS sometimes found more than PDT and sometimes the opposite. PDT worked well with *transposed* code and worked well when *renaming* of variable was tried. It worked well with transposed code because of the DVM tree structure the files are saved in by XML Store. When moving code around in a file the subtree representing that code remains the same; if it was within the given parameters before, it will still be - i.e. XML Store will still share that subtree and we will still find it. This is one of the major forces of PDT, and this is accounted to the tree structure. The strength when faced with renamed identifiers is due to the front-end which gives the user the possibility of converting all identifiers to the same constant. The use of parameters works very well and when adjusted more or less plagiarism is found. A recommendation of the optimal parameters for SML programs was provided in section 6.2.1.

In section 2.2.1 some basic good properties of plagiarism detection tools was pointed out. Our system fulfills two of these and to a part the third. Our system *does* only report each token (subtree) once. Transposed code *do* have minimal effect on the overall plagiarism score and it *does*, to a degree, degrade gracefully in the presence of random insertions and deletions. However this last property is largely influenced by the value of `maxnodes`.

The use of XML Store has provided us with the benefits described in section 3.1.1. These includes the identification of nodes sharing a common subtree which we can use to identify copied parts in two or more documents fast (one traversal per document). We do not need to compare a set of fingerprints pairwise between the documents but need only traverse the document's subtrees once to find all parts shared with other documents. The use of XML Store implies that we do not need to worry about how to select representative fingerprints for every document since XML Store saves entire documents. If a part is shared and it is within the user given parameters we *guarantee* to find it.

Future work done on this system could include an extensive testing, the development of a front-end for another programming language, implementation of support for all SML statements, improvement of the output and optimisation of the code.

References

- [ASW03] Alex Aiken, Saul Schleimer, and Daniel S. Wilkerson.
Winnowing: Local Algorithms for Document Fingerprinting.
Proceedings of the ACM SIGMOD International Conference on Management, June 2003.
- [Bar00] David J. Barnes.
Object-Oriented Programming with Java: An Introduction. Prentice Hall, 2000.
ISBN: 0-13-086900-7.
- [CFL⁺03] X. Chen, B. Francia, M. Li, B. Mckinnon, and A. Seker.
Shared Information and Program Plagiarism Detection. The Computer Journal, Vol. 33, December 2003.
<http://monod.uwaterloo.ca/papers/04sid.pdf>.
- [Clo] Paul Clough.
Plagiarism in natural programming languages: an overview of current tools and technologies.
<http://www.dcs.shef.ac.uk/~clougie/papers/Plagiarism.pdf>.
- [CLRS01] Thomas H. Cormen, Charles E. Leiserson, Ronald R. Rivest, and Clifford Stein.
Introduction to Algorithms. 2nd Edition, MIT Press, Cambridge, 2001.
ISBN: 0-26-203293-7.
- [Com] *Comparator*.
<http://www.catb.org/~esr/comparator/comparator.html>.
- [Mos] *Moss (Measure Of Software Similarity)*.
<http://www.cs.berkeley.edu/~aiken/moss.html>.
- [Pau] Lawrence C. Paulsen.
ML for the working programmer. 2nd edition.
- [PP02] Kasper Bøgebjerg Pedersen and Jesper Tejlgaard Pedersen.
Value-oriented XML Store.
Master's thesis, ITU and DTU, 2002.
<http://www.it-c.dk/~kasper/xmlstore/pdf/thesis.pdf>.
- [VW96] K. L. Verco and M. Wise.
Software for Detecting Suspected Plagiarism: Comparing Structure and Attribute-Counting Systems. the First Australian Conference on Computer Science, Sydney, 1996.
- [Wha90] G. Whale.
Identification of Program Similarity in Large populations. The Computer Journal, Vol. 33, pp. 140–146, 1990.
- [Wis] Michael J. Wise.
Improved Detection Of Similarities In Computer Program And Other Texts.
ACM Press, 1996,

<http://www.bio.cam.ac.uk/~mw263/ftp/doc/yap3.ps>.
ISSN: 0097-8418.

[XML] *XML Store source code*.
<http://plan-x.org/xmlstore/>.

A med_i.sml as XML document

Some locations have been replaced with a “location=” or “loc=” to make the document fit on page. This is the XML version of the SML program `fun med_i x = foldr (fn (a,b) => b orelse a=x) false;`

```

<MosML-program name="med_i.sml">
  <startprog>
    <VALdec location="(0,54)">
      <NONRECURSIVE></NONRECURSIVE>
      <RECURSIVE>
        <ValBind>
          <VARpat location="(4,9)">med_i</VARpat>
          <FNexp location="(4,54)">
            <Match>
              <MRules>
                <Pats>
                  <VARpat location="(10,11)">x</VARpat>
                </Pats>
                <APPexp location="(14,54)">
                  <APPexp location="(14,48)">
                    <VIDPATHexp location=".>foldr</VIDPATHexp>
                    <PARexp location="(20,48)">
                      <FNexp location="(21,47)">
                        <Match>
                          <MRules>
                            <Pats>
                              <RECpat location="(24,29)">
                                <INTlab>1</INTlab>
                                <VARpat location=".>a</VARpat>
                                <INTlab>2</INTlab>
                                <VARpat location=".>b</VARpat>
                              </RECpat>
                            </Pats>
                            <ORELSEexp location="(33,47)">
                              <VIDPATHexp loc=".>b</VIDPATHexp>
                              <APPexp location="(42,47)">
                                <VIDPATHexp loc=".></VIDPATHexp>
                                <RECexp location=".>
                                  <INTlab>1</INTlab>
                                  <VIDPATHexp loc=".>a</VIDPATHexp>
                                  <INTlab>2</INTlab>
                                  <VIDPATHexp loc=".>x</VIDPATHexp>
                                </RECexp>
                              </APPexp>
                            </ORELSEexp>
                          </MRules>
                        </Match>
                      </FNexp>
                    </PARexp>
                  </APPexp>
                <VIDPATHexp location="(49,54)">>false</VIDPATHexp>
              </APPexp>
            </MRules>
          </Match>
        </FNexp>
      </ValBind>
    </RECURSIVE>
  </VALdec>
</startprog>
</MosML-program>

```

B SML files

B.1 Tuning

The SML files used for tuning. This file is available in three versions. In the other two version all declaration have been transposed. The order of the declaration is different in all three files.

```
(* four statements should be reported *)
(* the list should not be reported *)
(* fun tilcifre is OK to be reported as well *)

(* should not be reported *)
fun tilciffer n = chr (ord #"0" + n);

(* should not be reported *)
fun divmod (a,b) = (a div b, a mod b);

(* should possibly be reported *)
fun tilcifre n = if n = 0 then []
                else let val (q, r) = divmod (n, 10)
                    in tilcifre q @ [tilciffer r]
                    end

(* should not be reported *)
val n = 1;
val cifre = tilcifre n;

(* should not be reported *)
val ekstra = 1 - length cifre;

(* should be reported *)
fun roman (_, 0) = ""
  | roman ((n as (value, letter))::ns, i) =
    if i < value then roman(ns, i)
    else letter ^ roman(n::ns, i - value);

(* should not be reported *)
val numerals2 = [(1000, "M"), (900, "CM"), (500, "D"), (400, "CD"),
                (100, "C"), (90, "XC"), (50, "L"), (40, "XL"),
                (10, "X"), (9, "IX"), (5, "V"), (4, "IV"), (1, "I")];

(* non of these function should be reported *)
fun f1(x) = x ^ ".";
fun f2(x) = if x>0 then 1 else 2;
fun f3(x,y) = if x then y else 0;

(* should be reported *)
fun mindstePrimFaktor n =
  let
    fun mDivisor (n,d) = if n mod d = 0
                        then d
                        else mDivisor(n,d+1)
  in
    mDivisor (n,2)
  end;

(* should not be reported *)
datatype solution = SNONE | SONE of real | STWO of real*real;

(* should be reported *)
fun minlist []=NONE
|minlist (x::xr)=
```

```

let
  fun minlist' ([],min)=min
    |minlist' (x::xr,min)=minlist'(xr,Int.min(x,min))
in
  SOME (minlist'(xr,x))
end;

```

B.2 testdepth.sml

```

fun testdepth t =
  let
    val n = ord(t)
  in
    if n <= 31 then
      #"}" :: [chr(n+33)]
    else
      if n >= 125 andalso n <= 184 then
        #"}" :: [chr(n-60)]
      else
        if n >= 185 andalso n <= 255 then
          #"~" :: [chr(n-152)]
        else
          raise Fail "Try to escape above 255"
    end;

```

B.3 Applying counter measures to tun1.sml

B.3.1 Renaming of identifiers

```

(* should not be reported *)
fun tilc y = chr (ord #"0" + y);

(* should not be reported *)
fun divm (x,y) = (x div y, x mod y);

(* should possibly be reported *)
fun tilci x = if x = 0 then []
              else let val (y, z) = divm (x, 10)
                    in tilci y @ [tilc y]
                    end

(* should not be reported *)
val x = 1;
val c = tilci x;

(* should not be reported *)
val eks = 1 - length c;

(* should be reported *)
fun ro (_, 0) = ""
  | ro ((x as (v, l))::xs, i) =
    if i < v then ro(xs, i)
    else l ^ ro(x::xs, i - v);

(* should not be reported *)
val numerals2 = [(1000, "M"), (900, "CM"), (500, "D"), (400, "CD"),
                 (100, "C"), (90, "XC"), (50, "L"), (40, "XL"),
                 (10, "X"), (9, "IX"), (5, "V"), (4, "IV"), (1, "I")];

(* non of these function should be reported *)
fun g1(z) = z ^ ".";
fun g2(z) = if z>0 then 1 else 2;

```

```

fun g3(z,v) = if z then v else 0;

(* should be reported *)
fun mindstePrimFaktor x =
  let
    fun mDivisor (x,y) = if x mod y = 0
                        then y
                        else mDivisor(x,y+1)
  in
    mDivisor (x,2)
  end;

(* should not be reported *)
datatype solution = SNON | SNE of real | SWO of real*real;

(* should be reported *)
fun minl []=NONE
|minl (y::yr)=
  let
    fun minl' ([],m)=m
    |minl' (y::yr,m)=minl'(yr,Int.min(x,m))
  in
    SOME (minl'(yr,y))
  end;

```

B.3.2 Transposed code

```

(* boer fanges *)
fun minlist []=NONE
|minlist (x::xr)=
  let
    fun minlist' ([],min)=min
    |minlist' (x::xr,min)=minlist'(xr,Int.min(x,min))
  in
    SOME (minlist'(xr,x))
  end;

fun mindstePrimFaktor n =
  let
    fun mDivisor (n,d) = if n mod d = 0
                        then d
                        else mDivisor(n,d+1)
  in
    mDivisor (n,2)
  end;

fun roman (_, 0) = ""
| roman ((n as (value, letter)):ns, i) =
  if i < value then roman(ns, i)
  else letter ^ roman(n:ns, i - value);
val numerals2 = [(1000, "M"), (900, "CM"), (500, "D"), (400, "CD"),
                 (100, "C"), (90, "XC"), (50, "L"), (40, "XL"),

                 (10, "X"), (9, "IX"), (5, "V"), (4, "IV"), (1, "I)];

(* boer ikke fanges *)
datatype solution = SNONE | SONE of real | STWO of real*real;

fun f2(x) = if x>0 then 1 else 2;
fun f3(x,y) = if x then y else 0;
fun f1(x) = x ^ ".";

```

```

(* boer ikke fanges *)
fun divmod (a,b) = (a div b, a mod b);

(* boer ikke fanges *)
fun tilciffer n = chr (ord #"0" + n);

(* boer måske fanges *)
fun tilcifre n = if n = 0 then []
                 else let val (q, r) = divmod (n, 10)
                       in tilcifre q @ [tilciffer r]
                       end

(* boer ikke fanges *)
val n = 1;
val cifre = tilcifre n;

(* boer ikke fanges *)
val ekstra = 1 - length cifre;

```

B.3.3 Adding redundant statements

```

(* should not be reported *)
fun tilciffer n = chr (ord #"0" + n);

fun helle n = n = "Hello";
(* should not be reported *)
fun divmod (a,b) = (a div b, a mod b);

(* should possibly be reported *)
fun tilcifre n = if n = 0 then []
                 else let val (q, r) = divmod (n, 10)
                       val a = 5; (* redundant here *)
                       in tilcifre q @ [tilciffer r]
                       end

(* should not be reported *)
val n = 1;
val x = 4; (* redundant here *)
val cifre = tilcifre n;

(* should not be reported *)
val ekstra = 1 - length cifre;

(* should be reported *)
fun roman (_, 0) = ""
  | roman ((n as (value, letter))::ns, i) =
    if i < value then roman(ns, i)
    else letter ^ roman(n::ns, i - value);

(* should not be reported *)
val numerals2 = [(1000, "M"), (900, "CM"), (500, "D"), (400, "CD"),
                (100, "C"), (90, "XC"), (50, "L"), (40, "XL"),
                (10, "X"), (9, "IX"), (5, "V"), (4, "IV"), (1, "I")];

(* non of these function should be reported *)
fun f1(x) = x ^ "." ^ ""; (* redundant here *)
fun f2(x) = if x>0 then 1 else 2;
fun f3(x,y) = if x then y else 0;

(* should be reported *)
fun mindstePrimFaktor n =
  let

```

```

    fun mDivisor (n,d) = if n mod d = 0
                        then d
                        else mDivisor(n,d+1)
    val n = 15 * 85 (* redundant here *)
in
    mDivisor (n,2)
end;

(* should not be reported *)
datatype solution = SNONE | SONE of real | STWO of real*real;

(* should be reported *)
fun minlist []=NONE
|minlist (x::xr)=
    let
        val n = #"B"
        fun minlist' ([],min)=min
          | minlist' (x::xr,min)=minlist'(xr,Int.min(x,min))
    in
        SOME (minlist'(xr,x))
    end;

```

B.3.4 Changing fixtivity

```

(* should not be reported *)
nonfix +;
nonfix ::;
nonfix -;
nonfix <;
nonfix >;
nonfix ^;
nonfix @;
fun tilciffer n = chr (+ (ord #"0",n));

(* should not be reported *)
fun divmod (a,b) = (a div b, a mod b);

infix divmod;

(* should possibly be reported *)
fun tilcifre n = if n = 0 then []
                else let val (q, r) = (n divmod 10)
                    in (@(tilcifre q, [tilciffer r]))
                end

(* should not be reported *)
val n = 1;
val cifre = tilcifre n;

(* should not be reported *)
val ekstra = -(1,(length cifre));

(* should be reported *)
fun roman (_, 0) = ""
| roman (:(n as (value, letter)),ns, i) =
    if <(i,value) then roman(ns, i)
    else ^(letter,roman(:(n,ns), -(i, value)));

(* should not be reported *)
val numerals2 = [(1000, "M"), (900, "CM"), (500, "D"), (400, "CD"),
                 (100, "C"), (90, "XC"), (50, "L"), (40, "XL"),
                 (10, "X"), (9, "IX"), (5, "V"), (4, "IV"), (1, "I")];

```

```

(* non of these function should be reported *)
fun f1(x) = ^ (x, ".");
fun f2(x) = if >(x,0) then 1 else 2;
fun f3(x,y) = if x then y else 0;

(* should be reported *)
fun mindstePrimFaktor n =
  let
    fun mDivisor (n,d) = if n mod d = 0
                        then d
                        else mDivisor(n,+(d,1))
  in
    mDivisor (n,2)
  end;

(* should not be reported *)
datatype solution = SNONE | SONE of real | STWO of real*real;

(* should be reported *)
fun minlist []=NONE
|minlist (::(x,xr))=
  let
    fun minlist' ([],min)=min
    |minlist' (::(x,xr),min)=minlist'(xr,Int.min(x,min))
  in
    SOME (minlist'(xr,x))
  end;

```

B.3.5 Changing the structur of selection statements

```

(* should not be reported *)
val ekstra = 1 - length cifre;

(* should be reported *)
fun roman (_, 0) = ""
| roman ((n as (value, letter))::ns, i) =
  (* Selection changed *)
  if i > value then letter ^ roman(n::ns, i - value)
  else roman(ns, i);

(* should not be reported *)
val numerals2 = [(1000, "M"), (900, "CM"), (500, "D"), (400, "CD"),
                (100, "C"), (90, "XC"), (50, "L"), (40, "XL"),
                (10, "X"), (9, "IX"), (5, "V"), (4, "IV"), (1, "I")];

(* non of these function should be reported *)
fun f1(x) = x ^ ".";
fun f2(x) = if x<0 then 2 else 1;
fun f3(x,y) = if not(x) then 0 else y;

(* should be reported *)
fun mindstePrimFaktor n =
  let
    fun mDivisor (n,d) = if n mod d <> 0
                        then mDivisor(n,d+1)
                        else d
  in
    mDivisor (n,2)
  end;

(* should not be reported *)

```

```

datatype solution = SNONE | SONE of real | STWO of real*real;

(* should be reported *)
fun minlist []=NONE
|minlist (x::xr)=
  let
    fun minlist' ([],min)=min
    |minlist' (x::xr,min)=minlist'(xr,Int.min(x,min))
  in
    SOME (minlist'(xr,x))
  end;

```

B.4 Changing datatypes

```

(* should not be reported *)
fun tilciffer n = chr (ord #"0" + ceil(n));

(* should not be reported *)
fun divmod (a,b) = (real(ceil(a) div ceil(b)), real(ceil(a) mod ceil(b)));

(* should possibly be reported *)
fun tilcifre n = if n = 0.0 then []
  else let val (q : real, r) = divmod (n, 10.0)
        in tilcifre q @ [tilciffer r]
        end

(* should not be reported *)
val n = 1.0;
val cifre = tilcifre n;

(* should not be reported *)
val ekstra = 1 - length cifre;

(* should be reported *)
fun roman (_, 0.0) = ""
| roman ((n as (value, letter))::ns, i) =
  if i < value then roman(ns, i)
  else letter ^ roman(n::ns, i - value);

(* should not be reported *)
val numerals2 = [(1000.0, "M"), (900.0, "CM"), (500.0, "D"), (400.0, "CD"),
  (100.0, "C"), (90.0, "XC"), (50.0, "L"), (40.0, "XL"),
  (10.0, "X"), (9.0, "IX"), (5.0, "V"), (4.0, "IV"), (1.0, "I")];

(* non of these function should be reported *)
fun f1(x) = x ^ ".";
fun f2(x) = if x>0.0 then 1.0 else 2.0;
fun f3(x,y) = if x then y else 0.0;

(* should be reported *)
fun mindstePrimFaktor n =
  let
    fun mDivisor (n,d) = if real(round(n) mod round(d)) = 0.0
      then d
      else mDivisor(n,d+1.0)
  in
    mDivisor (n,2.0)
  end;

(* should not be reported *)
datatype solution = SNONE | SONE of real | STWO of real*real;

```

```
(* should be reported *)
fun minlist []=NONE
|minlist (x::xr)=
  let
    fun minlist' ([],min)=min
    |minlist' (x::xr,min)=minlist'(xr,Int.min(x,min))
  in
    SOME (minlist'(xr,x))
  end;
```

C Source code

C.1 Font-end - the XML dumping code

C.1.1 Changes made in the SML compiler

From line 430 to 436 in Compiler.sml:

```
fun compileImplPhrase xos os elab dec =
  let val (iBas,resdec) = resolveToplevelDec dec in
    (* KFL *)
    XMLDump.dump xos resdec
    ; compResolvedDecPhrase os elab (iBas,resdec)
  end
;
```

From line 453 to 497 in compiler.sml

```
(* KFL *)
val filename_xml = "../XMLdocs/" ^ filename ^ ".xml"
val xos = TextIO.openOut filename_xml

in
  (* Added by LL to produced parsable XML *)
  XMLDump.text xos "<?xml version=\"1.0\" ?>\n" ;
  XMLDump.text xos ("<MosML-program name=\"\"^filename^.sml\">\n") ;
  XMLDump.text xos "<startprog>\n" ;

  ( start_emit_phrase os;
    app (compileImplPhrase xos os elab) decs;
    (case umode of
      STRmode =>
        (Hasht.clear (iBasOfSig(!currentSig));
         Hasht.clear (sigEnvOfSig(!currentSig)))
      .....)

        (getOption (!uStamp)) (#uMentions (!currentSig))
        os
      end);
    (* Added by LL to produced parsable XML *)
    XMLDump.text xos "</startprog>\n" ;
    XMLDump.text xos "</MosML-program>" ;

    close_out os; TextIO.closeOut xos;
    restorePrState()
  end
```

C.1.2 The XML dumping file

```
structure XMLDump :> XMLDump =
struct

  (** XML utilities **)
  (** We use the Msp module for constructing XML efficiently *)
  local open Msp in
    infix &&

    (* A function that really belongs to the Msp module *)
    fun writeseq os Empty      = ()
      | writeseq os Nil        = TextIO.output(os, "\n")
      | writeseq os ($ s)     = TextIO.output(os, s)
      | writeseq os ($$ ss)   = List.app (fn s => TextIO.output(os, s)) ss
      | writeseq os (s1 && s2) = (writeseq os s1; writeseq os s2)
  end
```

```

val $ = Msp.$
val $$ = Msp.$$
val op && = Msp.&&
val empty = Msp.Empty
(* tag inserts extra newlines for prettyfication *)
fun tag t ws = Msp.markl t ws && Msp.Nl
(* tag with attributes *)
fun taga t at ws = Msp.markla at t ws && Msp.Nl
val mark = Msp.mark0
fun tagl t elem list =
  tag t (List.foldr op && empty (List.map elem list))
(* tag list with attributes *)
fun tagla t at elem list =
  taga t at (List.foldr op && empty (List.map elem list))

(** The dumper code **)
(** First tranlate to XML represented as a Msp.wseq and then print that *)
local open Asynt Location
  val FIXME = mark "FIXME"
  val out = TextIO.stdOut;
  val output = TextIO.print "Do you wish to use alpha conversion?\n"
  val input = TextIO.stdIn;
  val alpha = TextIO.inputLine(input)
  fun FIXME2 s = mark ("FIXME " ^ s)
in
  (* if the user have choosen to use alpha coversion all          *)
  (* variable names, function names and strings will be replaced *)
  val isAlpha = (Char.contains alpha #"y")

  fun writest os s = writeseq os ($ s)

  fun reloc (Loc(i1, i2)) = "location=\"(^^(Int.toString i1)^\",\"
    ^^(Int.toString i2)^\")\"
  fun resString s = $ (Msp.htmlencode s)

  (* simplified, not to dump all info *)
  fun showIDInfo {info, qualid} = $(Msp.htmlencode(Const.showQualId qualid))

  (* Replaces variable and function names - alpha conversion *)
  fun showIDInfoAlpha {info, qualid} = $("V")

  fun longmodidinfo (idinf, _) = tag "LongModIdInfo" (showIDInfo idinf)

  fun locstring (loc, s) = taga (reloc(loc)) "LocString" ($ s)

  fun tyvar tvar = tag "TyVar" (showIDInfo tvar)

  fun tyvarseq [] = empty
    | tyvarseq seq = tagl "TyVarSeq" tyvar seq

  local datatype Lab = datatype Mixture.Lab in
  fun lab (INTlab i) = tag "INTlab" ($(Msp.htmlencode(Int.toString i)))
    | lab (STRINGlab s) = tag "STRINGlab" ($(Msp.htmlencode s))
  end

  (* The row function as it ought to be *)
  fun row loc t elem xs = tagla (reloc(loc)) t (fn (l, x) => lab l && elem x) xs

  fun tyconpath (location, tcp) =
    case tcp of
      LONGtyconpath (ltc) => taga (reloc(location)) "LONGtyconpath" (showIDInfo ltc)
    | WHEREtyconpath (ltc, mid, mexp) => taga (reloc(location)) "WHEREtyconpath"
      ((showIDInfo ltc) && locstring mid && modexp mexp)

  and ty (location, t) =
    case t of
      TYVARTy(tvar) => taga (reloc(location)) "TYVARTy" (tyvar tvar)
    | RECTy tr => row location "RECTy" ty tr
    | CONty (t1, tcp) => taga (reloc(location)) "CONty" (tagl "TYlist" ty t1 && tyconpath tcp)
    | FNty(t1, t2) => taga (reloc(location)) "FNty" (ty t1 && ty t1)
    | PACKty (se) => taga (reloc(location)) "PACKty" (sigexp se)
    | PARTy (t1) => taga (reloc(location)) "PARTy" (ty t1)

```

```

and pat (loc, p) =
  case p of
    VARpat longVid => if isAlpha then taga (reloc(loc)) "VARpat"
                      (showIDInfoAlpha longVid) else taga
                      (reloc(loc)) "VARpat" (showIDInfo longVid)
  | SCONpat (sc, _) => taga (reloc(loc)) "SCONpat" (scon sc)
  | WILDCARDpat => $ "<WILDCARDpat></WILDCARDpat>"
  | NILpat longVid => taga (reloc(loc)) "NILpat" (showIDInfo longVid)
  | CONSPat (lv, pa) => taga (reloc(loc)) "CONSPat" (showIDInfo lv && (pat pa))
  | EXNILpat longVid => taga (reloc(loc)) "EXNILpat" (showIDInfo longVid)
  | EXCONSPat (lv, pa) => taga (reloc(loc)) "EXCONSPat" (showIDInfo lv && (pat pa))
  (* | EXNAMEpat of Lambda.Lambda *)
  | EXNAMEpat l => FIXME2 "EXNAMEpat"
  | REFpat (pa) => taga (reloc(loc)) "REFpat" (pat pa)
  | INFIXpat (ref (RESinfixpat p)) => pat p
  | INFIXpat (ref (UNRESinfixpat p)) => tagla (reloc(loc)) "INFIXpat" pat p
  | RECpat (ref (TUPLerp p)) => tagla (reloc(loc)) "RECpat" pat p
  | RECpat (ref (RECrp (pl, _))) => row loc "RECpat" pat pl
  | VECpat pats => tagla (reloc(loc)) "VECpat" pat pats
  | PARpat (pa) => taga (reloc(loc)) "PARpat" (pat pa)
  | TYPEDpat (pa, t) => taga (reloc(loc)) "TAYPEDpat" (pat pa && ty t)
  | LAYEREDpat (pl, p2) => taga (reloc(loc)) "LAYEREDpat" (pat pl && pat p2)

and scon sc =
  case sc of
    Const.INTscon i => if isAlpha then tag "INTscon" ($(Int.toString l))
                      else tag "INTscon" ($(Int.toString i))
  | Const.WORDscon i => tag "WORDScon" ($(Word.toString i))
  | Const.CHARscon i => if isAlpha then tag "CHARscon" $(Msp.htmlencode
                                           (Char.toString #"C"))
                      else tag "CHARscon" $(Msp.htmlencode(Char.toString i))
  | Const.REALscon i => if isAlpha then tag "REALscon" $(Real.toString l.0)
                      else tag "REALscon" $(Real.toString i)
  | Const.STRINGscon i => if isAlpha then tag "STRINGscon" ($ "StringReplaced")
                      else tag "STRINGscon" ($ (Msp.htmlencode (String.toString(i))))

and exp (location, e) =
  case e of
    SCONexp(sc, _) => taga (reloc(location)) "SCONexp" (scon sc)
  | VIDPATHexp (ref (RESvidpath ex)) => if isAlpha
    then taga (reloc(location)) "VIDPATHexp" (showIDInfoAlpha ex)
    else taga (reloc(location)) "VIDPATHexp" (showIDInfo ex)
  | VIDPATHexp (ref (OVLvidpath (lv, _, _))) => if isAlpha
    then taga (reloc(location)) "VIDPATHexp" (showIDInfoAlpha lv)
    else taga (reloc(location)) "VIDPATHexp" (showIDInfo lv)
  | RECexp (ref (TUPLere ex)) => tagla (reloc(location)) "TUPLExp" exp ex
  | RECexp (ref (RECre ex)) => row location "RECexp" exp ex
  | VECexp (ex) => tagla (reloc(location)) "VECexp" exp ex
  | LETexp (d, ex) => taga (reloc(location)) "LETexp" (dec d && exp ex)
  | PARExp (ex) => taga (reloc(location)) "PARExp" (exp ex)
  | APPexp (e1, e2) => taga (reloc(location)) "APPexp" (exp e1 && exp e2)
  | INFIXexp (ref (RESinfixexp e)) => exp e
  | INFIXexp (ref (UNRESinfixexp exps)) => tagla (reloc(location)) "INFIXexp" exp exps
  | TYPEDexp (ex, t) => taga (reloc(location)) "TYPEDexp" (exp ex && ty t)
  | ANDALSOexp (ex1, ex2) => taga (reloc(location)) "ANDALSOexp" (exp ex1 && exp ex2)
  | ORELSEexp (ex1, ex2) => taga (reloc(location)) "ORELSEexp" (exp ex1 && exp ex2)
  | HANDLEexp (ex, mat) => taga (reloc(location)) "HANDLEexp" (exp ex && match mat)
  | RAISEexp ex => taga (reloc(location)) "RAISEexp" (exp ex)
  | IFexp (e1, e2, e3) => taga (reloc(location)) "IFexp" (exp e1 && exp e2
    && exp e3)
  | WHILEexp (e1, e2) => taga (reloc(location)) "WHILEexp" (exp e1 && exp e2)
  | FNexp mat => taga (reloc(location)) "FNexp" (match mat)
  | SEQexp (e1, e2) => taga (reloc(location)) "SEQexp" (exp e1 && exp e2)
  (* Note: Skal option med? - t&ank lidt over det *)
  | STRUCTUREexp (mexp, sexp, _) => taga (reloc(location)) "STRUCTUREexp"
    (modexp mexp && sigexp sexp)
  | FUNCTORexp (mexp, sexp, _) => taga (reloc(location)) "FUNCTORexp"
    (modexp mexp && sigexp sexp)

and modexp (loc, (mexp, _) =
  case mexp of
    DECmodexp (d) => taga (reloc(loc)) "DECmodexp" (dec d)
  | LONGmodexp (longmodid) => taga (reloc(loc)) "LONGmodexp" (showIDInfo longmodid)
  | LETmodexp (d, me) => taga (reloc(loc)) "LETmodexp" (dec d && modexp me)

```

```

| PARmodexp (me) => taga (reloc(loc)) "PARmodexp" (modexp me)
| CONmodexp (me, se) => taga (reloc(loc)) "CONmodexp" (modexp me && sigexp se)
| ABSmodexp (me, se) => taga (reloc(loc)) "ABSmodexp" (modexp me && sigexp se)
(* | FUNCTORmodexp of FunctorSort * ModId * (IdKindDesc ref) * SigExp * ModExp *)
| FUNCTORmodexp (fs, mi, ik, se, me) => FIXME2 "FUNCTORmodexp"
| APPmodexp (m1, m2) => taga (reloc(loc)) "APPmodexp" (modexp m1 && modexp m2)
| RECmodexp (mi, ref (SOME(Globals.RECrec rs)), se, me) => FIXME
| RECmodexp (mi, ref (SOME(Globals.NONrec rs)), se, me) => FIXME
| RECmodexp (mi, (ref (NONE)), se, me) => taga (reloc(loc)) "RECmodexp"
  (locstring mi && sigexp se && modexp me)

and sigexp (loc, se) =
  case se of
  | SPECsigexp (sp) => taga (reloc(loc)) "SPECsigexp" (spec sp)
  | SIGIDsigexp (sid) => taga (reloc(loc)) "SIGIDsigexp" (locstring sid)
  | WHEREsigexp (se, tvarseq, longtycon, t) => taga (reloc(loc)) "WHEREsigexp"
    (sigexp se && tvarseq tvarseq && (showIDInfo longtycon) && ty t)
  | FUNSIGsigexp (fs, mi, sel, se2) => FIXME
  | RECSigexp (mi, sel, se2) => taga (reloc(loc)) "RECSigexp"
    (locstring mi && sigexp sel && sigexp se2)

and dec (location, d) =
  case d of
  | VALdec(tvseq, (nonrecursive, recursive)) => taga (reloc(location)) "VALdec"
    (tyvarseq tvseq && tagl "NONRECURSIVE" valbind nonrecursive
    && tagl "RECURSIVE" valbind recursive)
  | PRIM_VALdec(tvseq, primvals) => taga (reloc(location)) "PRIM_VALdec"
    (tyvarseq tvseq && tagl "PrimVals" primvalbind primvals)
  | FUNdec(ref fdec) => fundec fdec
  | TYPEdec tybinds => tagla (reloc(location)) "TYPEdec" tybind tybinds
  | PRIM_TYPEdec (tnameequ, tydecs) => taga (reloc(location)) "PRIM_TYPEdec"
    ($ "TyNameEqu" && tagl "typeDecs" tydecs tydecs)
  | DATATYPEdec (datbinds, _) => taga (reloc(location)) "DATATYPEdec"
    (tagl "DatBindList" datbind datbinds)
  | DATATYPEerepdec (tycon, tcp) => taga (reloc(location)) "DATATYPEerepdec"
    (locstring tycon && tyconpath tcp)
  | ABSTYPEdec(datbinds, SOME(tybinds), d) => tag "ABSTYPEdec"
    (tagl "DatBindList" datbind datbinds && tagl "TypeBindList" tybind tybinds && dec d)
  | ABSTYPEdec(datbinds, NONE, d) => tag "ABSTYPEdec"
    (tagl "DatBindList" datbind datbinds && dec d)
  | EXCEPTIONdec exbinds => tagla (reloc(location)) "EXCEPTIONdec" exbind exbinds
  | LOCALdec (d1, d2) => taga (reloc(location)) "LOCALdec" (tag "LOCAL" (dec d1) &&
    tag "IN" (dec d2))
  | OPENdec (longModIdInfos) => tagla (reloc(location)) "OPENdec"
    longmodidinfo longModIdInfos
  | STRUCTUREdec modbinds => taga (reloc(location)) "STRUCTUREdec"
    (tagl "ModBinds" modbind modbinds)
  | FUNCTORdec funbinds => taga (reloc(location)) "FUNCTORdec"
    (tagl "FunBinds" funbind funbinds)
  | SIGNATUREdec sigbinds => tag "SIGNATUREdec" (tagl "SigBinds" sigbind sigbinds)
  | EMPTYdec => $ "<EMPTYdec></EMPTYdec>"
  | SEQdec(d1, d2) => taga (reloc(location)) "SEQdec" (dec d1 && dec d2)
  | FIXITYdec (Globals.NONFIXst, names) => tagla (reloc(location)) "FIXITYdec" resString names
  | FIXITYdec (Globals.INFIXst i, names) => taga (reloc(location)) "FIXITYdec"
    ($ (Int.toString i) && tagl "StringList" resString names)
  | FIXITYdec (Globals.INFIXrst i, names) => taga (reloc(location)) "FIXITYdec"
    ($ (Int.toString i) && tagl "StringList" resString names)

and spec (loc, s) =
  case s of
  | VALspec(tvseq, valdescs) => taga (reloc(loc)) "VALspec"
    (tyvarseq tvseq && tagl "ValSpecList" valdesc valdescs)
  | PRIM_VALspec (tvseq, primvals) => taga (reloc(loc)) "PRIM_VALspec"
    (tyvarseq tvseq && tagl "PRIM_VALbindList" primvalbind primvals)
  | TYPEDESCspec (tne, td) => taga (reloc(loc)) "TYPEDESCspec" ($ "TyNameEqu"
    && tagl "TypDescs" typdecs td)
  | TYPEspec (tybinds) => taga (reloc(loc)) "TYPEspec"
    (tagl "TypBinds" tybind tybinds)
  | DATATYPEspec (datbinds, SOME(tybinds)) => taga (reloc(loc)) "DATATYPEspec"
    (tagl "DatBinds" datbind datbinds && tagl "TypBinds" tybind tybinds)
  | DATATYPEspec (datbinds, NONE) => taga (reloc(loc)) "DATATYPEspec"
    (tagl "DatBinds" datbind datbinds)
  | DATATYPEerepspec (tcon, tconpath) => taga (reloc(loc)) "DATATYPEerepspec"

```

```

      (locstring tcon && tyconpath tconpath)
| EXCEPTIONspec (exdescs) => taga (reloc(loc)) "EXCEPTIONspec"
      (tagl "ExDescs" exdesc exdescs)
| LOCALspec (spl, sp2) => taga (reloc(loc)) "LOCALspec" (spec spl && spec sp2)
| OPENspec (longModIdInfos) => tagla (reloc(loc)) "OPENspec"
      longmodidinfo longModIdInfos
| EMPTYspec => $ "<EMPTYspec></EMPTYspec>"
| SEQspec (spl, sp2) => taga (reloc(loc)) "SEQspec" (spec spl && spec sp2)
| INCLUDEspec (sexp) => taga (reloc(loc)) "INCLUDEspec" (sigexp sexp)
| STRUCTUREspec (moddescs) => taga (reloc(loc)) "STRUCTUREspec"
      (tagl "ModDescs" moddesc moddescs)
| FUNCTORspec (fundescs) => taga (reloc(loc)) "FUNCTORspec"
      (tagl "FunDescs" fundesc fundescs)
| SHARINGTYPEspec(sp, longtycons) => taga (reloc(loc)) "SHARINGTYPEspec" (spec sp
      && tagl "LongTyCons" showIDInfo longtycons)
| SHARINGspec (sp, (loc, longmodids)) => taga (reloc(loc)) "SHARINGspec" (spec sp
| FIXITYspec (Globals.NONFIXst, names) => tagla (reloc(loc)) "FIXITYspec" resString names
| FIXITYspec (Globals.INFIXst i, names) => taga (reloc(loc)) "FIXITYspec"
      ($(Int.toString i) && tagl "StringList" resString names)
| FIXITYspec (Globals.INFIXrst i, names) => taga (reloc(loc)) "FIXITYspec"
      ($(Int.toString i) && tagl "StringList" resString names)
| SIGNATUREspec (sigbinds) => taga (reloc(loc)) "SIGNATUREspec"
      (tagl "SigBinds" sigbind sigbinds)

(* shouldn't be like this *)
and signa (loc, s) =
  case s of
    (* shouldn't be like this *)
    (* NamedSig of {locsigid : SigId, sigexp: SigExp} *)
    NamedSig _ => $ "<NamedSig></NamedSig>"
  | AnonSig specs => tagla (reloc(loc)) "AnonSig" spec specs
  | TopSpecs specs => tagla (reloc(loc)) "TopSpecs" spec specs

(* shouldn't be like this *)
and structur (loc, s) =
  case s of
    (* NamedStruct of {locstrid : ModId, locsigid : SigId option, decs : Dec list} *)
    NamedStruct _ => $ "<NamedStruct></NamedStruct>"
    (* | Abstraction of {locstrid : ModId, locsigid : SigId, decs : Dec list} *)
    | Abstraction _ => $ "<Abstraction></Abstraction>"
    | AnonStruct decs => tagla (reloc(loc)) "AnonStruct" dec decs
    | TopDecs decs => tagla (reloc(loc)) "TopDecs" dec decs

and valbind (ValBind(ref p, e)) =
  tag "ValBind" (pat p && exp e)

and fclause (FClause(ref pats, e)) = tag "FClause" (tagl "Pats" pat pats && exp e)

and valdesc(idinf, (loc, t)) = taga (reloc(loc)) "ValDesc" ((showIDInfo idinf) && (ty (loc, t)))

and exdesc (idinf, t) =
  case t of
    SOME(t) => tag "ExDesc" ((showIDInfo idinf) && ty t)
  | NONE => tag "ExDesc" (showIDInfo idinf)

and primvalbind (inf, (loc, t), i, s) =
  tag "PrimValBind" ((showIDInfo inf) && (ty (loc, t)) && (Int.toString i) && (s))

and typbind(tvseq, tcon, t) =
  tag "TypBind" (tyvarseq tvseq && locstring tcon && ty t)

and conbind (ConBind(idinf, t)) =
  case t of
    SOME t => tag "ConBind" ((showIDInfo idinf) && ty t)
  | NONE => tag "ConBind" (showIDInfo idinf)

and exbind e =
  case e of
    EXDECexbind(idinf, _) => tag "EXDECexbind" (showIDInfo idinf)
  | EXEQUALexbind(i1, i2) => tag "EXEQUALexbind" ((showIDInfo i1) && (showIDInfo i2))

and fvalbind (loc, fclauses) = taga (reloc(loc)) "FValBind" (tagl "FClauses" fclause fclauses)

```

```

and datbind (ts, tc, conbinds) = tag "DatBind" (tyvarseq ts && locstring tc
&& tagl "CondBindList" conbind conbinds)

and modbind m =
  case m of
    MODBINDmodbind (mid, mexp) => tag "MODBINDmodbind" (locstring mid && modexp mexp)
  | ASmodbind (mid, sexp, e) => tag "ASmodbind" (locstring mid && sigexp sexp && exp e)

and funbind f =
  case f of
    FUNBINDfunbind (fid, mexp) => tag "MODBINDmodbind" (locstring fid && modexp mexp)
  | ASfunbind (fid, sexp, e) => tag "ASmodbind" (locstring fid && sigexp sexp && exp e)

and sigbind(SIGBINDsigbind(sigid, sexp)) = tag "SIGBINDsigbind" (locstring sigid && sigexp sexp)

and typdecs (tvseq, tcon) = tag "TypDecs" (tyvarseq tvseq && locstring tcon)

and mrule (MRule(ref pats, e)) = (tagl "Pats" pat pats) && exp e

and match(mrules) = tag "Match" (tagl "MRules" mrule mrules)

and moddesc(MODDESCmoddesc(modid, sexp)) = tag "MODDESC" (locstring modid && sigexp sexp)

and fundec fdec =
  case fdec of
    UNRESfundec(tvseq, fbinds) => Fnlib.fatalError "XMLDump"
  | RESfundec d => dec d

and fundesc(FUNDESCfundesc(fid, sexp)) = tag "FUNDESCfundesc" (locstring fid && sigexp sexp)

fun text os t =
  writeseq os ($ t)

fun dump os d =
  let val xml = dec d
  in
    writeseq os xml
  end

end

end

```

C.2 The detection tool

C.2.1 The main class

This takes care of getting everything done in the right order. Reading the user parameters, storing the document, traversing the trees, finding the copied nodes and writing the output.

```

public class Plagiarism {

  // for the attributebuilding
  private static SetStandardMethodsAttr smfattr;
  private static DFTraversal dft;

  private static FindPlagiarism fp;
  private static MakeOutput output;

  // help objects for the computation of attributes and finding plagiarism
  private static SearchHashtable sht = new SearchHashtable();
  private static AttributeComp attrcomp;

  // list of all the nodes in xmlstore
  private static Hashtable nodelist;

  // list of nodes that are found to be plagiarism
  private static ArrayList plagodelist = null;

  // for finding plagiarism - set by the user
  private static int indegree;
  private static int maxdepth;
  private static int mindepth;

```

```

private static int maxnodes;
private static int minnodes;

public static void main(String[] args)
    throws XMLStoreException, XMLStoreXMLException, UnknownValueReferenceException, IOException
{
    // Makes use of weakattributes in XML Store
    GlobalSettings.globalWeakAttributes = true;

    System.setProperty("org.xml.sax.driver", "org.apache.crimson.parser.XMLReaderImpl");

    if(args.length < 1){
        System.err.println("Usage: java myTest <directoryname>");
        return;
    }

    // Prompting for the next number from command-line via ReadPlagParams class
    ReadPlagParams read = new ReadPlagParams(); // from Barnes pp360-362

    // Indegree
    System.out.println("Please give the maximum number of documents that must be shared");
    indegree = read.getNumber();

    // Maxdepth
    System.out.println("Please give the maximum depth of subtrees");
    maxdepth = read.getNumber();

    // Mindepth
    System.out.println("Please give the minimum depth of the subtrees");
    mindepth = read.getNumber();

    // Maxnodes
    System.out.println("Please give the maximum number of nodes in subtrees");
    maxnodes = read.getNumber();

    // Minnpdes
    System.out.println("Please give the minimum number of nodes in subtrees");
    minnodes = read.getNumber();

    // parameters for plagiarism detection has now been given
    read.finished(); // close the inputstream after end use

    System.out.println("Indegree: " + indegree + " , maxdepth: " + maxdepth + " , mindepth: " + mindepth
        + " , maxnodes: " + maxnodes + " , minnodes: " + minnodes);

    // Create XML Store
    XMLStore xmlstore = new WriteBufferXMLStore("plagiarismstore.disk");

    String publicdocumentfilename = "publicdocument.xml";

    // get publicdocument file
    File publicdocfile = new File(publicdocumentfilename);

    boolean existpublicdocfile = publicdocfile.exists();

    ValueReference pdfvref = null;
    // save public document file if it exists
    if(existpublicdocfile){
        Node pdfnode = SAXBuilder.build(publicdocumentfilename);
        pdfvref = xmlstore.save(pdfnode);
    } // ELSE do nothing: no public document

    // Save files in directory given as argument.
    File dir = new File(args[0]);

    if(!dir.exists()){
        System.err.println(args[0] + " does not exist! Try another directory.");
        return;
    }

    if(dir.isFile()){
        System.err.println(args[0] + " is a file \n" + "must be a directory");
    }
}

```

```

        return;
    }

    String[] filesindir = null;

    if(dir.isDirectory()){
        System.out.println(args[0] + " is a directory");
        filesindir = dir.list();
    }

    ArrayList doclist = new ArrayList(filesindir.length);

    // files in directory is stored - hidden files are skipped!
    for(int i = 0 ; i < filesindir.length ; i++){
        System.out.println(i + " : " + filesindir[i]);
        File file = new File(filesindir[i]);
        if(!file.isHidden()){ // file should be stored
            Node n = SAXBuilder.build(args[0]+"/"+filesindir[i]);
            ValueReference vref = xmlstore.save(n);
            doclist.add(vref);
        } // ELSE do nothing as file is hidden and should not be stored
    }

    if(doclist.size() != 0){

        // traverse the public document file if it exists and set it's attributes first of all
        if(existspublicdocfile && pdfvref != null){
            SetStandardMethodsAttr smfattr = new SetStandardMethodsAttr(xmlstore);
            nodelist = smfattr.setSMAttributes(pdfvref);
        } // ELSE initialize nodelist as it will not have been initialized by setSMAttributes.
        else{
            nodelist = new Hashtable(1000);
        }

        //DFTraversal dft = new DFTraversal(xmlstore, nodelist);
        DFTraversal dft = new DFTraversal(xmlstore, nodelist);

        nodelist = dft.DFTraverseTree(doclist);

        ArrayList plagnodelist = null;

        if(nodelist.size() != 0){

            FindPlagiarism fp = new FindPlagiarism(maxdepth,mindepth,maxnodes,minnodes,indegree,xmlstore);
            plagnodelist = fp.findPlagiarism(nodelist);
        } // ELSE do nothing

        System.out.println("\n\n");

        MakeOutput output = new MakeOutput(xmlstore);
        output.makeOutput(plagnodelist,nodelist);

    }
    else{
        System.out.println("No documents in directory " + args[0] + " to find plagiarism for!");
    }
}

xmlstore.close();
}
}

```

C.2.2 Reading the user defined parameters

This class reads the user defined parameters from the command line. Highly inspired by Code 12.17 in [Bar00] p. 361.

```

public class ReadPlagParams{

    private BufferedReader reader;
    private StreamTokenizer stream;

    // reads input from command window. Highly inspired by Code 12.17 p. 361 in
    // David J. Barnes: "Object-Oriented Programming with Java" p. 361.
    public int getNumber() throws IOException{
        // ensure that no unused tokens from previous calls are used
    }
}

```

```

        reader = new BufferedReader(new InputStreamReader(System.in));
        stream = new StreamTokenizer(reader);

        if(stream.nextToken() != StreamTokenizer.TT_NUMBER || stream.ttype == StreamTokenizer.TT_EOF){
            // prompt for a new number
            System.out.println("Input not a number or not readable! Please try again.");
            System.out.flush();
            getNumber();
        }

        return (int) stream.nval;
    }

    // close the input stream after end use
    public void finished() throws IOException{
        reader.close();
    }
}

```

C.2.3 Traversal of the public document tree

This class traverses the public document tree to set the `isstandardmethod` attribute of the nodes in the public document to true.

```

class SetStandardMethodsAttr{

    XMLStore xmlstore;
    Hashtable nodelist;
    SearchHashtable sht = new SearchHashtable();

    public SetStandardMethodsAttr(XMLStore xmlstore){
        this.xmlstore = xmlstore;
        nodelist = new Hashtable(100);
    }

    // starts the traversal of the standard document
    public Hashtable setSMAttributes(ValueReference vref)
        throws XMLStoreException, XMLStoreXMLException, UnknownValueReferenceException, IOException
    {
        Node node;
        node = xmlstore.load(vref);
        traverseTree(node);

        return nodelist;
    }

    private int globaltravtree = 1;

    private void traverseTree(Node node){

        int travtree = globaltravtree;

        ValueReference vref = node.getValueReference();
        if(!sht.inNodelist(nodelist, vref)){

            setAttributes(node);

            ChildNodes children = node.getChildNodes();

            if(children != null){ // node is no leaf

                for(int i = 0 ; i < children.getLength();i++){
                    Node child = children.getNode(i);
                    globaltravtree++;
                    traverseTree(child);
                }
            } // ELSE do nothing as node has no children to be set

        } // ELSE do nothing as node is already set.

    }

    private void setAttributes(Node node){

```

```

        ValueReference vref = node.getValueReference();
        PlagAttributes nodeattr = new PlagAttributes(vref,true);
        nodelist.put(new Integer (vref.hashCode()),nodeattr);
    }

    private void writeNode(Node node){
        System.out.print(" "+node.getNodeValue());
        System.out.println(" , vref: " + node.getValueReference());
    }
}

```

C.2.4 Traversal of the document trees

This class traverses the document trees with depth-first traversal and sets the plagiarism attributes of the nodes.

```

public class DFTraversal{

    private XMLStore xmlstore;

    /*Objects for computation and traversal*/
    // for searchin arraylists
    private SearchHashtable sht = new SearchHashtable();
    // for computation of plag attributes
    private AttributeComp attrcomp = new AttributeComp(sht);
    // for traversing subtrees whose depth has already been set
    private PreOrderTraversal potrav = new PreOrderTraversal(xmlstore, sht, attrcomp);

    private Hashtable nodelist; // list of all the nodes' plagiarism attributes

    // ValueReference to and name of the document currently traversed
    private ValueReference currentdocref;
    private String currentdocname;

    private int numberTravTree = 1; // for depugging purposes

    public DFTraversal(XMLStore xmlstore){
        this.xmlstore = xmlstore;
        nodelist = new Hashtable(1000);
    }

    public DFTraversal(XMLStore xmlstore, Hashtable nodelist){
        this.xmlstore = xmlstore;
        this.nodelist = nodelist;
    }

    // start the traversal of the document
    public Hashtable DFTraverseTree(ArrayList doclist)
        throws XMLStoreIOException, UnknownValueReferenceException
    {
        for(int i = 0 ; i < doclist.size() ; i++){
            currentdocref = (ValueReference) doclist.get(i);
            Node currentdoc = xmlstore.load(currentdocref);
            currentdocname = currentdoc.getAttribute("name");
            System.out.println(" current document processed: " + currentdocname + " ");
            traverseTree(currentdoc, null);
        }
        numberTravTree = 1; // for debugging purposes RESET
        return nodelist;
    }

    private void traverseTree(Node node, Node parent){

        int thisTravTree = numberTravTree; // for debugging purposes

        ValueReference vref = node.getValueReference();

        boolean inList = sht.inNodelist(nodelist,vref);
        PlagAttributes nodeattr = null;

        if(!inList){
            nodeattr = new PlagAttributes(node.getValueReference(), false);
        }
        else{
            nodeattr = sht.findAttributes(nodelist,vref);
        }
    }
}

```

```

    }

    ChildNodes children = node.getChildNodes();

    if (children != null){ // node is not a leaf

        for(int i = 0; i < children.getLength(); i++){

            Node child = children.getNode(i);

            /* if nodelist does not contain a PlagAttributes object with the
             * ValueReference of this child or the depth is -1 then it has not yet had its depth
             * computed and can be traversed otherwise depth should not be computed
             * (but the rest of the plagiarism attributes can!)
             */
            boolean inlist = sht.inNodelist(nodelist, child.getValueReference());

            if(!inlist || sht.findAttributes(nodelist,child.getValueReference()).getMaxdepth() == -1){
                numberTravTree++; // for debugging purposes
                traverseTree(child,node);
            }
            else{
                potrav.startPOtraversal(child,node,nodelist,currentdocref, currentdocname);
            }
        }
    }
    else{ // Node is a leaf
        attrcomp.findStore(node,parent, nodeattr, nodelist, currentdocref, currentdocname, true);
    }

    if (children != null){
        attrcomp.findStore(node,parent, nodeattr, nodelist, currentdocref, currentdocname, false);
    }

}

// for debugging purposes
private void writeNode(Node n){
    System.out.print(n.getNodeValue() + ": ");
    System.out.println("vref=" + n.getValueReference() + " ");
}
}
}

```

C.2.5 Traversal of subtrees in Pre-Order

This class traverses the subtrees of the document if the max and mindepth has already been set.

```

class PreOrderTraversal{

    private XMLStore xmlstore;

    private SearchHashtable sht;

    private Hashtable nodelist;
    private ValueReference currentdocref;
    private String currentdocname;

    private AttributeComp attrcomp;

    private int numTrav = 1; // for debugging purposes

    /* constructor
     */
    public PreOrderTraversal(XMLStore xmlstore, SearchHashtable sht,
        AttributeComp attrcomp){
        this.xmlstore = xmlstore;
        this.sht = sht;
        this.attrcomp = attrcomp;
    }

    /* Starts the traversing of the tree. This method initialises some variables that
     * is needed for the setting=updating of the node attributes. Only the doclist needs to be

```

```

    * computed=updated at this point (maxdepth, mindepth and nodes
    * have already been set, and indegree will be computed later)
    */
public void startPOtraversal(Node node, Node parent, Hashtable nodelist,
                             ValueReference docref, String docname){
    this.nodelist = nodelist;
    currentdocref = docref;
    currentdocname = docname;
    numTrav = 1; // for debugging purposes
    traverseTree(node, parent);
}

/* Traverses the documenttree in preorder
*/
private void traverseTree(Node node, Node parent){

    int thisTravTree = numTrav;

    PlagAttributes nodeattr = sht.findAttributes(nodelist,node.getValueReference());
    boolean inDoclist = sht.inDoclist(nodeattr.getDocList(),currentdocref);

    // add parent to parentlist if it's not there already
    if(!sht.inDoclist(nodeattr.getParents(),parent.getValueReference())){
        nodeattr.addParent(parent);
    } // ELSE do nothing

    // add currentdocref to doclist if it's not there already
    if(!sht.inDoclist(nodeattr.getDocList(),currentdocref)){
        nodeattr.addDocument(currentdocref, currentdocname);
    } // ELSE do nothing

    ChildNodes children = node.getChildNodes();
    if(children != null){ // node is not a leaf

        for(int i = 0; i < children.getLength(); i++){
            Node child = children.getNode(i);
            numTrav++; //for debugging purposes
            traverseTree(child, node);
        }
    } // ELSE do nothing: node is a leaf
}

/* Prints the value of the node on screen
* for debugging purposes
*/
private void writeNode(Node n){
    System.out.print(n.getNodeValue()+ " : ");
    System.out.println(" vref: " + n.getValueReference() + " (POtrav writeNode)");
}
}

```

C.2.6 Applying the plagiarism criteria to the nodes

This class finds the nodes which are possible plagiarism by applying the plagiarism criteria.

```

public class FindPlagiarism{

    private XMLStore xmlstore; // for debugging purposes

    private SearchHashtable sht = new SearchHashtable();

    /* criteria set by the user
    */
    // the depth of the subtree
    private int maxdepth;
    private int mindepth;
    // the number of nodes in the subtree
    private int nodesmax;
    private int nodesmin;
    // how many documents that share this node
    private int indegreemax;

    private double boundarymax;
    private double boundarymin;
}

```

```

private ArrayList plagodelist = new ArrayList();

public FindPlagiarism(int maxdepth, int mindepth, int nodesmax,
                    int nodesmin, int indegreemax, XMLStore xmlstore){
    this.xmlstore = xmlstore;
    this.maxdepth = maxdepth;
    this.mindepth = mindepth;
    this.nodesmax = nodesmax;
    this.nodesmin = nodesmin;
    this.indegreemax = indegreemax;
}

// puts all nodes within the criteria in a new nodelist as they
// are possible plagiarisms
public ArrayList findPlagiarism(Hashtable nodelist){

    Object[] attrs = sht.tableToArray(nodelist);

    for(int i = 0; i < attrs.length; i++){

        PlagAttributes tempnode = (PlagAttributes) attrs[i];

        boolean isstandardmethod = tempnode.getIsStandardmethod();
        int tempindegree = tempnode.getParents().size();

        if(!isstandardmethod && tempindegree != 1 && !(tempnode.getMaxdepth() == 0)){

            boolean inIndegree = isDocumentsharing(tempnode,nodelist);

            if(inIndegree){

                boolean insubtreedepth = inSubtreedepth(tempnode);
                boolean insubtreenodes = inSubtreenodes(tempnode);

                if(insubtreedepth && insubtreenodes){
                    plagodelist.add(tempnode);
                } // ELSE do nothing: no plagiarism detected
            } // ELSE do nothing: not shared by different documents
        } // ELSE do nothing: node is part of public document, has 1 or 0 parent(s) or is a leaf
    }

    return plagodelist;
}

// check if the depths of the subtree are within the depth boundary
private boolean inSubtreedepth(PlagAttributes nodeattr){

    if(nodeattr.getMindepth() >= mindepth && nodeattr.getMaxdepth() <= maxdepth){
        return true;
    }
    else{
        return false;
    }
}

// checks if the number of nodes in the subtree is within the node boundary
private boolean inSubtreenodes(PlagAttributes nodeattr){

    int nodesinsubtree = nodeattr.getNodesinSubtree();

    if(nodesmin <= nodesinsubtree && nodesinsubtree <= nodesmax){
        return true;
    }
    else{
        return false;
    }
}

// checks if the sharing is between different documents
public boolean isDocumentsharing(PlagAttributes nodeattr, Hashtable nodelist){

    int docsharingdegree = 0;
    int nlength = nodeattr.getDocList().size(); // length of this nodes docList

```

```

Hashtable parents = nodeattr.getParents();

if(parents.size() == 0){ // node is a document root
    docsharingdegree = nlength;
}
else{

    Object[] parentlist = sht.tableToArray(parents);

    for(int i = 1 ; i < parentlist.length ; i++){

        ValueReference parentvref = (ValueReference) parentlist[i];
        PlagAttributes parentattr = (PlagAttributes) sht.findAttributes(nodelist,parentvref);

        int plength = parentattr.getDocList().size();

        // if the lenght of the doclist of the node is longer than that of its
        // parent it has more ingoing edges and indegree will therefore be
        // nlength - plength + 1. The node is thus shared between diff. documents
        // '+ 1' because otherwise the original topline wont be counted!!

        int tempdocsharingdegree;

        if(nlength > plength){
            tempdocsharingdegree = (nlength - plength) + 1;
        }
        else{
            tempdocsharingdegree = 1;
        }

        if(tempdocsharingdegree > docsharingdegree){
            docsharingdegree = tempdocsharingdegree;
        } // ELSE do nothing
    }
}

if(docsharingdegree > 1 && docsharingdegree < indegreemax){
    return true;
}
else{
    return false;
}
}

// for debugging purosos
private void writeNodeName(PlagAttributes attr){

    ValueReference vref = attr.getValueReference();
    Node node;

    try{
        node = xmlstore.load(vref);
    }catch(Exception e){
        node = null;
    }

    if(node != null){
        System.out.println(node.getNodeValue() + ":");
    }
    else{
        System.out.println("Unknown node:");
    }
}
}
}

```

C.2.7 Storing of the plagiarism attributes

This class is the place the plagiarism attributes for the nodes are stored.

```

public class PlagAttributes{

    private boolean isstandardmethod; // whether the node is part of the public document

```

```
private String firstdoc = null; // the first document that visits this node
private ValueReference vref; // value reference of the node this Object belongs to
private Hashtable parentnodes = new Hashtable(); // list of this nodes parents (ValueReferences)
private int maxdepth = -1; // maximum depth of the subtree
private int mindepth = -1; // minimum depth of the subtree
private Hashtable doclist = new Hashtable(); // list of documents that can reach this node
private int nodes; // number of nodes in subtree + this node

public PlagAttributes(ValueReference vref, boolean isstandardmethod){
    this.vref = vref;
    this.isstandardmethod = isstandardmethod;
}

/* Get and set methods
*/
public ValueReference getValueReference(){
    return vref;
}

public boolean getIsStandardmethod(){
    return isstandardmethod;
}

public void addParent(Node parent){
    if(parent != null){
        ValueReference vref = parent.getValueReference();
        parentnodes.put(new Integer(vref.hashCode()),vref);
    } //ELSE do nothing
}

public Hashtable getParents(){
    return parentnodes;
}

public void setMaxMindepth(int max, int min){
    maxdepth = max;
    mindepth = min;
}

public int getMaxdepth(){
    return maxdepth;
}

public int getMindepth(){
    return mindepth;
}

public void addDocument(ValueReference doc, String docname){
    doclist.put(new Integer(doc.hashCode()),doc);
    // if this node has not been visited before set firstdoc
    if (firstdoc == null){
        firstdoc = docname;
    } // ELSE do nothing
}

public String getFirstdoc(){
    return firstdoc;
}

// for copying of object
private void setDocList(Hashtable doclist){
    this.doclist = doclist;
}

public Hashtable getDocList(){
    return doclist;
}

public void setNodesinSubtree(int i){
    nodes = i;
}

public int getNodesinSubtree(){
```

```

        return nodes;
    }
}

```

C.2.8 Compute the attributes

```

public class AttributeComp{

    SearchHashtable sht;

    public AttributeComp(SearchHashtable sht){
        this.sht = sht;
    }

    /* In charge of computing and setting the node attributes in the right order
    * the right order being only that MaxMindepth must be set first as this
    * saves the nodes in the nodelist so that the latter methods can use them!
    */
    public void findStore(Node node, Node parent, PlagAttributes nodeattr, Hashtable nodelist,
        ValueReference docref, String docname, boolean isleaf){
        // needs to be done first as this saves the nodeattr on the nodelist
        // for use by the latter methods
        findStoreMaxMindepth(node, nodeattr, nodelist, isleaf); // Must ALWAYS be first!
        findStoreNumNodes(node, nodeattr, nodelist, isleaf);
        addParent(nodeattr, parent);
        addDocument(nodeattr, docref, docname);
    }

    /* finds the max and min depth of subtrees for the given node.
    * The subtree of the node has already been computed at this time because it's
    * depth-first traversal, therefore the nodeattr _does_ lie in the nodelist
    */
    private void findStoreMaxMindepth(Node node, PlagAttributes nodeattr,
        Hashtable nodelist, boolean isleaf){

        if(!isleaf){

            ChildNodes children = node.getChildNodes();
            int childrenlength = children.getLength();

            if(childrenlength > 0){
                Node firstchild = children.getNode(0);

                ValueReference firstchildref = firstchild.getValueReference();
                PlagAttributes firstchildattr = sht.findAttributes(nodelist, firstchildref);
                int tempmax = firstchildattr.getMaxdepth();
                int tempmin = firstchildattr.getMindepth();

                for (int i = 1 ; i < children.getLength() ; i++){
                    Node child = children.getNode(i);
                    ValueReference childref = child.getValueReference();
                    PlagAttributes childattr = sht.findAttributes(nodelist, childref);

                    if(childattr.getMaxdepth() > tempmax){
                        tempmax = childattr.getMaxdepth();
                    }
                    if(childattr.getMindepth() < tempmin){
                        tempmin = childattr.getMindepth();
                    }
                }

                tempmax++;
                tempmin++;
                nodeattr.setMaxMindepth(tempmax,tempmin);

            }

            // The node is an ELEMENT with no children, treat the node as a leaf
            else{
                nodeattr.setMaxMindepth(0,0);
            }
        }else{
            nodeattr.setMaxMindepth(0,0);
        }
    }
}

```

```

        Integer attrkey = new Integer(nodeattr.getValueReference().hashCode());
        nodelist.put(new Integer(nodeattr.getValueReference().hashCode()),nodeattr);
    }

    /* Computes and sets the number of nodes in the subtree of the given node.
     * The node is counted too!
     */
    private void findStoreNumNodes(Node node, PlagAttributes nodeattr,
                                   Hashtable nodelist, boolean isleaf){

        int tempnodecount = 1; // initialised to 1 as the current node must be counted too

        if(!isleaf){

            ChildNodes children = node.getChildNodes();
            for (int i = 0 ; i < children.getLength() ; i++){

                Node child = children.getNode(i);
                ValueReference childref = child.getValueReference();
                PlagAttributes childattr = sht.findAttributes(nodelist, childref);

                tempnodecount = tempnodecount + childattr.getNodesinSubtree();
            }
            nodeattr.setNodesinSubtree(tempnodecount);
        }else{
            nodeattr.setNodesinSubtree(tempnodecount); // leaves has no subtree
        }
    }

    // adds the parent of the node to the parent list
    private void addParent(PlagAttributes nodeatt, Node parent){

        if(parent != null){
            nodeatt.addParent(parent);
        }// ELSE do nothing: node has no parent
    }

    /* Stores the vref of the document node that are currently traversed.
     */
    private void addDocument(PlagAttributes nodeattr, ValueReference docref, String docname){

        Hashtable doclist = nodeattr.getDocList();

        if(!sht.inDoclist(doclist,docref)){
            nodeattr.addDocument(docref,docname);
        } // do nothing as doclist already contains docref
    }
}

```

C.2.9 Retrieving objects from a hashtable

This class helps to get retrieve the objects in the hashtables and to convert hashtables to arrays

```

public class SearchHashtable{

    /* Used to find an object in the hashtable ValueReference is the Key to the object
     * Used by findAttributes and inDoclist(?)
     * Returns an Object.
     */
    private Object findValue(Hashtable table, ValueReference vref){

        Integer vrefhash = new Integer(vref.hashCode());
        Object o = table.get(vrefhash);

        return o;
    }

    public PlagAttributes findAttributes(Hashtable table, ValueReference vref){

        PlagAttributes attr = (PlagAttributes) findValue(table,vref);

        return attr;
    }
}

```

```

    }

    public Docpercent findDocpercent(Hashtable table, String name){
        Docpercent doc = (Docpercent) table.get(name);
        return doc;
    }

    public boolean inDocpercents(Hashtable table, String name){
        return table.containsKey(name);
    }

    /* Uses findParameters to tell whether an object with the specified
    * ValueReference exists in the specified ArrayList.
    * Returns true if it does and false if not.
    */

    public boolean inHashtable(Hashtable table, ValueReference vref){

        Integer vrefhash = new Integer(vref.hashCode());
        boolean tcontain = table.containsKey(vrefhash);

        return tcontain;
    }

    public boolean inNodelist(Hashtable nodelist, ValueReference vref){

        Integer vrefhash = new Integer(vref.hashCode());
        boolean tcontain = nodelist.containsKey(vrefhash);

        return tcontain;
    }

    public boolean inDoclist(Hashtable doclist, ValueReference docref){

        Integer docrefhash = new Integer(docref.hashCode());
        boolean tcontain = doclist.containsKey(docrefhash);

        return tcontain;
    }

    // turns the hashtable into an array of Objects (the values of the table)
    public Object[] tableToArray(Hashtable table){
        Collection valuecoll = table.values();
        Object[] valuearray = valuecoll.toArray();
        return valuearray;
    }

    // for debugging purposes
    private Object[] tableToKeyArray(Hashtable table){
        Set kset = table.keySet();
        Object[] karray = kset.toArray();
        return karray;
    }

    // for debugging purposes
    public void writeNodelistKeys(Hashtable table){
        Object[] keys = tableToKeyArray(table);

        for(int i = 0; i < keys.length; i++){
            Integer key = (Integer) keys[i];
            System.out.println("Key #" + i + ": " + key);
        }
    }
}

```

C.2.10 Making the output for the HTML file

This class generate the output that is to be written to the HTML file, and writes the file.

```
public class MakeOutput{
```

```

//this is to round up to 2 decimal places
DecimalFormat myFormat = new DecimalFormat("0.00");

XMLStore xmlstore;
String attr;
String finalFiles = "";
String firstDoc = "";
String plagString = "";
printLocation location = new printLocation();
toHTML htmlFile = new toHTML();

Hashtable docpercents = new Hashtable();

PlagAttributes plagattr;
Hashtable allnodes;

SearchHashtable sht = new SearchHashtable();

public MakeOutput(XMLStore xmlstore){
    this.xmlstore = xmlstore;
}

public void makeOutput(ArrayList nodelist, Hashtable allnodeslist){

    allnodes = allnodeslist;
    printNumberOfPlagiarismParts(nodelist.size());
    String tmpString = "<BR>";

    for(int i = 0; i < nodelist.size(); i++){
        System.out.println("***** Plagiarised part #" + (i+1) + " *****\n");

        plagattr = (PlagAttributes) nodelist.get(i);

        printPlagiarismPart(plagattr.getValueReference());
        printDocuments(plagattr.getDocList());

        System.out.println("\n*****");
    }

    System.out.println("***** Total % copied in documents: *****");
    Object[] docstotalpercent = sht.tableToArray(docpercents);
    for(int i = 0; i < docstotalpercent.length; i++){
        Docpercent docp = (Docpercent) docstotalpercent[i];
        System.out.println(" " + docp.getName() + " " + docp.getPercent() + "%");
        tmpString = tmpString + docp.getName() + " ----> "
            + myFormat.format(docp.getPercent()) + "% " + "<BR>";
    }

    System.out.println();
    htmlFile.firstInFile("<b>Total % copied in documents:<br></b>"+tmpString);
    try{
        htmlFile.writeFile("output");
    }catch(Exception e) {System.out.println("Fejl! "+e); }

    System.out.println("*****");
}

public void printNumberOfPlagiarismParts(int i){
    System.out.println("#####");
    System.out.println("##### PLAGIARISM #####");
    System.out.println("#####\n");
    if (i>0){
        System.out.println(" There are " + i + " part(s) of possible plagiarism.\n");
    }
    else{
        System.out.println(" No plagiarism detected! \n");
    }
}

public void printPlagiarismPart(ValueReference rootnoderef){
    Node plagstartnode;
    try{

```

```

        plagstartnode = xmlstore.load(rootnoderef);
    }catch(Exception e){// this should never happen as node will always be there
        System.out.println(" No plagnode with that ref");
        plagstartnode = null;
    }
    PlagAttributes nattr = (PlagAttributes) allnodes.get(new Integer(rootnoderef.hashCode()));
    System.out.println(" Plagiarised part (rootnode): "+ plagstartnode.getNodeValue()
        + " - plagiarism degree: " + nattr.getNodesinSubtree());
    plagString = "<BR> Plagiarism degree: " + nattr.getNodesinSubtree() + "<BR>";

    try{
        firstDoc = nattr.getFirstdoc();
        attr = plagstartnode.getAttribute("location");
        System.out.println(" "+attr);
    }catch(Exception e) { }

}

// gets the filenames for the documents that shares a node
public void printDocuments(Hashtable doclist){

    String files = "";
    Object[] docarray = sht.tableToArray(doclist);
    ValueReference firstref = (ValueReference) docarray[0];

    System.out.println(" doclist length: " + doclist.size());

    for(int i = 0 ; i < docarray.length; i++){
        ValueReference docref = (ValueReference) docarray[i];
        Node docnode;
        String docname;
        try{
            docnode = xmlstore.load(docref);
            docname = docnode.getAttribute("name");
        }catch(Exception e){
            System.out.println(e);
            docnode = null;
            docname = "Unknown document";
        }

        Docpercent docp;
        if(!sht.inDocpercents(docpercents, docname)){
            docp = new Docpercent(docname);
            docpercents.put(docname, docp);
        }
        else{
            docp = sht.findDocpercent(docpercents, docname);
        }

        System.out.print(" Document " + (i+1) + ": " + docname);
        plagString += " Document " + (i+1) + ": " + docname;
        files = files+docname+", ";
        double plagpercent = percentagePlagiarised(docnode);
        setDocpercentage(docref, docp, plagpercent);
    }
    files = firstDoc+" seems to be copied in files "+files+"." +
        "<BR>Copied part listed further below.<BR>" + plagString;

    String printedLocation = location.pLocation(attr, "SMLdocs/"+firstDoc);
    htmlFile.appendHTML(printedLocation, files, attr);
}

// updates the total percentage of the document if it shares this node
private void setDocpercentage(ValueReference docref, Docpercent docp, double percent){

    Hashtable htparents = plagattr.getParents();
    Object[] parents = sht.tableToArray(htparents);
    if(parents.length != 0){
        boolean percentsat = false;
        for(int i = 0 ; i < parents.length; i++){
            ValueReference parentref = (ValueReference) parents[i];
            PlagAttributes parentattr = sht.findAttributes(allnodes, parentref);
            Hashtable parentdoclist = parentattr.getDocList();

```

```

        boolean indoclist = sht.inDoclist(parentdoclist, docref);
        if(indoclist && parentdoclist.size() <= 1 && !percentsat){
            docp.setPercent(percent);
            percentsat = true;
        }
    } // ELSE do nothing: node has no parents
}

// the percentage of the nodes of the whole doc that has been plagiarised
private double percentagePlagiarised(Node doc){

    double partnodes = (double) plagattr.getNodesinSubtree();
    ValueReference docvref = doc.getValueReference();
    PlagAttributes docattr = (PlagAttributes) allnodes.get(new Integer(docvref.hashCode()));
    double docnodes = (double) docattr.getNodesinSubtree();
    double percent = (double)((int)((partnodes/docnodes)*10000))/100;

    plagString += " ---- Nodes in document: " + docnodes
                + " ---- copied part "+ percent + "% of document <BR>";
    System.out.println("  Nodes in document: " + docnodes
                      + " - copied part "+ percent + "% of document");

    return percent;
}
}

```

C.2.11 Retrieval of locations from SML files

The below class will return what is written between the given location in a text file - here a SML file. The location should be given as a string on the form "(124,856)". 124 is the character in file in which to begin retrieving code and 856 is the last character to be retrieved.

```

import java.io.*;
import java.util.StringTokenizer;

// Class to find and print location in SML programs
class printLocation {

    String loc;
    String fInput;
    char[] cbuf;
    int size;

    // Method for finding and returning the text at the above given location
    // No input needed. Return type is of String
    public String pLocation(String location, String functionName) {

        loc = location;
        fInput = functionName;
        try{
            // Locations are given on the form "(loc1,loc2)"
            // Isolate the actual locations
            StringTokenizer sTokens = new StringTokenizer(loc, "(,)", true);

            // Convert the location to type Integer
            Integer i1 = new Integer(sTokens.nextToken());
            Integer i2 = new Integer(sTokens.nextToken());

            // Get the two locations as type integer
            int loc1 = i1.intValue();
            int loc2 = i2.intValue();
            size = loc2-loc1+1;
            cbuf = new char[size];

            // Open the SML as a BufferedReader
            BufferedReader reader = new BufferedReader(new FileReader(fInput));

            // Read the text in SML file from loc1 to loc2
            reader.skip(loc1);
            reader.read(cbuf, 0, size);

            // Converts the charbuffer to a String

```

```

        String output = new String(cbuf);

        // Return the string now containing the code
        // found between char loc1 to char loc2 in the SML file
        return (output);

    } catch (Exception e) {
        System.out.println(e);
    }
    return " ";
}
}

```

C.2.12 Writing of information to HTML file

This class provides the possibility of appending strings, any number to a HTML file. Each append via the method `append` will be separated by a horizontal line. The method `writeFile` will write the entire page to disk. The layout of the page will be that same as the pages on www.plan-x.org.

```

import java.io.*;
import java.util.*;

// Class to find and print location in SML programs
class toHTML {

    Date newDate = new Date();
    String location;

    // The string which to attach to the HTML document.
    // The string will by a part of an SML program and is provided
    // by the calling method.
    String s = "";

    // Start of the HTML document - The layout of the page will be the
    // same as the layout on the www.plan-x.org
    String beginPage = "<html><head><title>XMLStore - PDT</title></head> " +
        "<body bgcolor=\#f2f0e6\><div id=\"bread\"><p align=\"right\"> " +
        "<span lang=\"en-gb\">Plan-X</span> - <span lang=\"en-gb\"> " +
        "Plagiarism Detection Tool</span></div><div id=\"content\"><h1> " +
        "<span lang=\"en-gb\">Plan-X: Plagiarism Detection Tool</span></h1> " +
        "<hr color=\"#000000\" size=\"3\"><p align=\"right\">&nbsp;</p> " +
        "<p align=\"justify\">";

    // End of the HTML document - also containing the time and date where
    // the file was generated
    String endPage = "<BR><BR><BR><hr color=\"#000000\" size=\"3\">&nbsp;< " +
        "<i><font size=\"2\">Generated by the XMLStore Plagiarism Dectection Tool - " +
        + newDate.toString() + ".<BR>Tool made by Christa Fotel and Lars Langer " +
        "at the Department of Computer Science, Copenhagen University, May 2004</i> " +
        "</font></H5></div></div></body></html>";

    // Variable to contain the entire page
    String page = "";

    // Name of the output file. Initialised in the constructor and provided by the
    // calling method
    String fOut;

    // Method for appending text to the page
    public void appendHTML(String str, String fout, String location) {
        try{
            StringTokenizer sTokens = new StringTokenizer(location, "(.)");

            // The location is inserted so that user will know where to
            // look in the SML-files
            location = "<BR><BR><hr color=\"#000000\" size=\"3\">&nbsp;<i><font size=\"3\">"+
                "<BR>From character " + sTokens.nextToken() + " to charater " +
                + sTokens.nextToken() + " in file " + fout + " :<BR><BR>";

            // A identified part is attaceh to the HTML page
            s = "<BR><b><i> " + location + "</i></b> " + str;
            page += s; }
        catch(Exception e) { }
    }
}

```

```
}
public void firstInFile(String s) {
    // A method enabling the caller to attach a string to the
    // beginning of the page
    page = s+page;
}

// Writing output to the file given to the constructor
public void writeFile(String fout) {

    page = beginPage + page + endPage;

    // Replacing all line breaks with HTML line breaks
    page = page.replaceAll("\n", "<BR>");
    fout = fout + ".htm";

    try {

        // Writes the string to a new HTM file
        BufferedWriter writer = new BufferedWriter(new FileWriter(fout));
        System.out.println(" Writing Output .. .. ");
        writer.write(page, 0, page.length());
        writer.close();

    }
    catch (Exception e) {
        System.out.println(e);
    }
}
}
```

C.2.13 Storage of total percentage

Stores a document's total percentage copied.

```
class Docpercent{

    private String name;
    private double percent = 0;

    // Constructor
    public Docpercent(String name){
        this.name = name;
    }

    public String getName(){
        return name;
    }

    public void setPercent(double percent){
        this.percent += percent;
    }

    public double getPercent(){
        return percent;
    }
}
```