An AST-based Tool, Spector, for Plagiarism Detection: The Approach, Functionality, and Implementation

Vítor T. Martins, Pedro Rangel Henriques(✉), and Daniela da Cruz

Departamento de Informática/Centro Algoritmi,
Universidade do Minho, 4710-057 Braga, Portugal
{vtiagovm,pedrorangelhenriques,danieladacruz}@gmail.com
http://www.di.uminho.pt/eng/

Abstract. We propose a methodology using abstract syntax trees for the detection of plagiarism in source code, within an academic environment.

We show the architecture and decisions that came before we produce our own solution (Spector), after conducting a study of the methods and tools in existence. An example is then shown, which goes through and explains each of the algorithms steps.

Finally, conclusions are drawn noting that such a system, while not the most efficient, produces accurate results.

Keywords: Software · Plagiarism · Detection · Comparison · Test

1 Introduction

In our previous work [3], we discuss the results of our search for source code plagiarism detection methodologies and tools. We found several candidates but, after testing their accuracy by using files modified to hide plagiarism, we saw that most solutions had trouble with some cases.

Given the desire to make detections upon programs and our experience in language engineering, we were motivated to use a compilation aproach. After some research and discussion, we chose to use Abstract Syntax Trees (ASTs) since, by abstracting source code, we can analyze its functionality. Such tools exist and were said to be accurate [1,2] but were not available for download or use. To counteract that lack, our contribution will be the development of one such tool and make it openly available.

2 Approach

Figure 1 shows the representation of an AST generated from a source code. We can see that the original source code defines a function (sum) that adds two integers and uses it to print an equation that adds 4 to 7.

© Springer International Publishing Switzerland 2015
J.-L. Sierra-Rodríguez et al. (Eds.): SLATE 2015, CCIS 563, pp. 153–159, 2015.
DOI: 10.1007/978-3-319-27653-3_15

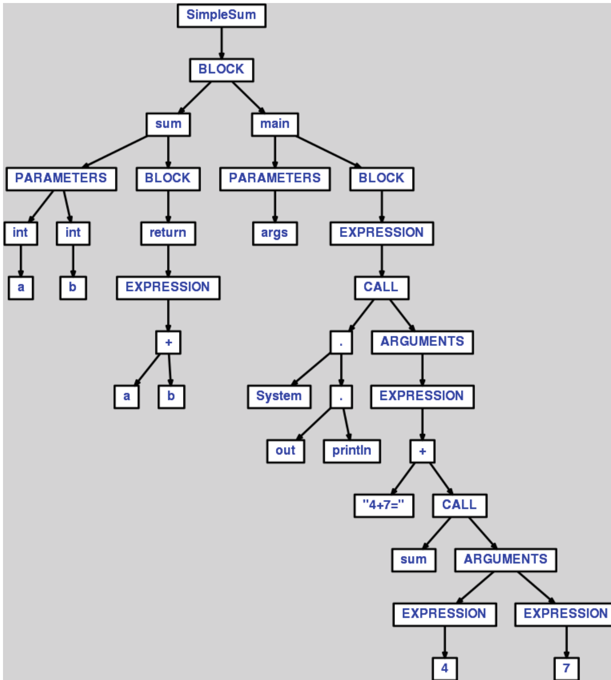

Fig. 1. An *AST* generated from a source code

Knowing that this representation gives us an ordered network of nodes, while retaining the information about their type and contents, it is easy to see that it provides enough information for source code comparison.

In order to facilitate the generation of *ASTs*, we have chosen to use *ANTLR*[1] [4] which allows us to generate parsers from grammar files, which can translate source code into an *AST* automatically.

2.1 Target Characteristics

As previously said, by using an intermediate abstraction, we can easily ignore specific characteristics. So we will focus on those that can be modified without altering the source code functionality. Namely Identifiers (the names of classes, variables, etc.), Expression elements (for ex.: $x, >, 1$), Conditionals (like if and while) and Blocks (a group of statements enclosed between $\{\}$).

This means we will ignore other characteristics like comments, as we are focusing on the source codes functionality. For example, if we wanted to compare Identifiers we do not need to check their scope or type, we can simply see if they are used in the same places and have similar behaviors.

[1] ANother Tool for Language Recognition.

2.2 Architecture

After making our decisions on how our system would work, we gave it the name of *Spector* (Source insPECTOR). We also produced a diagram (in Fig. 2) that shows the various parts and how they relate.

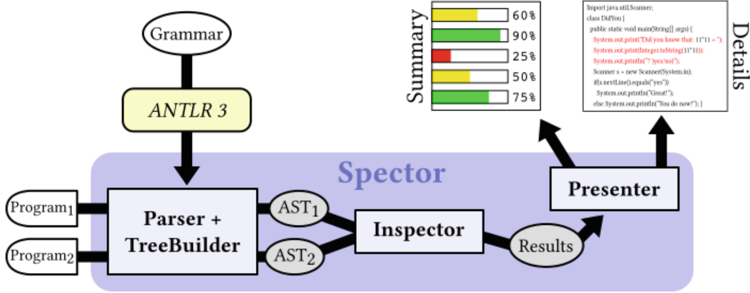

Fig. 2. A diagram of the interaction between the systems parts

In Fig. 2 we can observe that given a Grammar G, we can produce a Parser+TreeBuilder TB, using *ANTLR*. This TB can then be used to translate a Program P into an *AST*. Given a pair of Programs (let us say P_1 and P_2), a TB will be used to produce AST_1 and AST_2. These *ASTs* will then be delivered to an Inspector which will produce measures and send them to a Presenter which will output the results. To support a new language, we would simply replace G and generate a new TB.

3 Functionality

The functionality is split into 5 algorithms that each produce a measure, along with a final one that calculates the global measure.

Given source codes S_1 and S_2, we first check if they are not exact copies by comparing: the number of nodes, the number of nodes by category[2] and finally, the contents of every node. When all those comparisons match, we return a measure of 100 %, which avoids using the other algorithms.

For other cases, we have algorithms for each target (listed in Sect. 2.1). While these algorithms work in different ways, they follow a base functionality:

1. Build a map per source, associating elements to their occurrences (M_1, M_2),
2. Calculate the highest number of items between both Ms (A),
3. For each pair of items between M_1 and M_2,
   (a) If their number of occurrences is similar,

[2] In grammatical terms, these categories are the terminals that were assigned to integers by ANTLR (in a tokens file).

   i. Add the pair to a map (Candidates),
4. Calculate the size of the Candidates map (B),
5. Measure += $(B/A) * W_1$,
6. For each pair in the Candidates map,
   (a) If their number of occurrences by category is similar,
      i. Add the pair to a map (Suspects),
7. Calculate the size of the Suspects map (C),
8. Measure += $(C/B) * W_2$,
9. For each pair in the Suspects map,
   (a) If they have a similar behavior (specific to each algorithm),
      i. Add the pair to a map (Equivalences),
10. Calculate the size of the Equivalences map (D),
11. Measure += $(D/C) * W_3$,
12. For each pair of identifier names in the Equivalences map,
    (a) If their contents are the same,
       i. Add the pair to a map (Copies),
13. Calculate the size of the Copies map (E),
14. Measure += $(E/D) * W_4$,
15. Return Measure.

The W_i variables indicate weight constants and were given a value of 0.18, 0.42, 0.38 and 0.02, respectively. We chose these values based on a few tests. However, they must be adjusted through the use of further tests.

Of course, each algorithm is targeting something different, so they have the following differences:

Algorithm that detects Identifiers: The map (M) associates an Identifier name to its Occurrence nodes (IOM), which is similar if their parent nodes have the same category and the (non-identifier) neighbors[3] have the same contents.

Algorithm that detects Expression elements: The map (M) associates an Expression element to its Occurrence nodes (EOM), which is similar to those whose (non-identifier) neighbors contents are equal.

Algorithm that detects Conditionals: The map (M) associates a Conditional to its Condition node (CCM), which is similar to those whose condition has (non-identifier) nodes with the same contents.

Algorithm that detects Blocks: In this case, two maps are created: one associates a Block node to a Name[4] (BNM) and the other associates each Block node to all of its Children (BCM). The children are: every node inside the block along with the nodes from called blocks. This is done by checking if nodes are calls to internal methods, in which case the contents of the called block are added.

[3] The other children of this nodes parent.

[4] This name is the identifier of the parent block, in other words, "$main_{block}$" has a block with "main" as its name.

Main Algorithm: This algorithm calculates a final measure from those produced by the previous algorithms. That final measure is a similarity measure of a pair of Suspects.

Let us consider that each algorithm was implemented in a method $Method_i$, where i is a number from 1 to 5. With $Method_1$ being the one which determines if two source codes are exact copies. The algorithm works as follows:

1. X = Array with 5 elements,
2. for each method i,
   (a) X[i] = $Method_i(S_A, S_B)$,
3. If X[1] is different from 0,
   (a) Return X[1].
4. Otherwise
   (a) Calculate the number of Xs from 2 to 5 that are not 0 (A),
   (b) Measure = $\left(\frac{X[2]+X[3]+X[4]+X[5]}{A}\right) * 100$,
   (c) Return Measure.

As we can see, the algorithm either returns the X[1] measure (which is either 0 % or 100 %) or the average computation done using the other results (X[2] to X[5]) that were not 0 %.

Threshold. Since the algorithms match the number of occurrences when checking if the elements should be added to a Candidates map, the comparisons will be limited to cases with an equal number of occurrences. Which led us to the addition of a similarity threshold, which specifies the strictness of the comparisons. As an example: If we were comparing the number of nodes within two blocks and the first had 10 nodes, a threshold of 20 % means that the second *AST* must have between 8 and 12 nodes to be considered similar.

4 Implementation

To keep *Spector* modular, we split its functionality into two packages. A **lang** package which contains a Suspect class that will have the input generated from a source code and, for each language: The Parser+TreeBuilder classes and a Nexus class which interfaces with them. Along with a **spector** package that contains the main classes (Spector, Inspector and Presenter) along with their auxiliary classes (FileHandler and Comparison).

4.1 Features

We list below the main features provided by our tool:

1. Can output summary and/or detailed results,
2. Accepts submissions as groups of files,
3. Works offline,
4. Available as Open Source.

It is important to note that, the first two are relevant for any plagiarism detector and the other two are crucial since we want the tool to be available for integration into other systems.

5 Example

A complete example comparing 2 similar files (AllIn1 and AllIn2), from a semantic point of view, can be found with a detailed description at the following website: http://www.di.uminho.pt/~gepl/Spector/paper/slate15/examples/algorithms.pdf

Notice that the example finishes with the following computation:

$$\left(\frac{0.941 + 0.941 + 0.98 + 0}{3}\right) * 100 = 95.4\,\%$$

This gives us a similarity measure of 95.4 %, which indicates that the source codes are very similar. If we use the tool, produced following our methodology, to compare this test, we would read a different result in the HTML summary outputed and reproduced in Fig. 3.

Fig. 3. The HTML produced, showing the similarity measure.

As we can see, instead of being close to the expected 95.4 %, the result is instead 72.242 %. Detailed results show that the calculation was instead:

$$\left(\frac{92.45 + 94.017 + 98 + 4.5}{4}\right) = 72.242\,\%$$

This was due to the last algorithm returning a 4.5 as its similarity measure. The algorithms details show that the tool mistakingly associated the **AllIn1** to the **main** block in the Candidates map, due to them having the same number of child nodes. This shows us the importance of the weights in the algorithms since we want to avoid such false negatives as well as false positives.

6 Conclusion

In this paper, we have discussed our approach on building a tool that will detect plagiarism in source code named Spector. We have defined its structure and the decisions that were behind it. We have also seen the algorithms that will drive the comparisons and how they are used together to produce similarity measures. Seeing as we focused on detecting similarity between source code structures, the resulting tool is a *source code similarity detector* and is likely to report false-positives when faced with smaller code.

As future work we have the improvement of the results in terms of information about the associations established. Along with the upgrade of the Java grammar, to support the latest version of the Java language and the extension with grammars to cope with other languages.

The next step will be to perfect the tools implementation and test it against bigger test cases so that an optimal threshold and weights may be determined.

Acknowledgments. This work is co-funded by the North Portugal Regional Operational Programme, under the National Strategic Reference Framework (NSFR), through the European Regional Development Fund (ERDF), within project GreenSSCM - NORTE-07-02-FEDER-038973.

References

1. Bahtiyar, M.Y.: JClone: syntax tree based clone detection for Java. Master's thesis, Linnæus University (2010)
2. Cui, B., Li, J., Guo, T., Wang, J., Ma, D.: Code comparison system based on abstract syntax tree. In: 2010 3rd IEEE International Conference on Broadband Network and Multimedia Technology (IC-BNMT), pp. 668–673 (2010)
3. Martins, V.T., Fonte, D., Henriques, P.R., da Cruz, D.: Plagiarism detection: a tool survey and comparison. In: Pereira, M.J.V., Leal, J.P., Simões, A. (eds.) 3rd SLATE. OASIcs, vol. 38, pp. 143–158. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany (2014). http://drops.dagstuhl.de/opus/volltexte/2014/4566
4. Parr, T.J., Quong, R.W.: ANTLR: a predicated-LL(k) parser generator. Softw.-Pract. Experience **25**(7), 789–810 (1995). http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.15.70