Duplicate code detection using anti-unification

Peter Bulychev
Lomonosov Moscow State University, Russian Federation
Email: peter.bulychev@gmail.com

Marius Minea
Institute e-Austria Timisoara, Romania
Email: marius@cs.utt.ro

***Abstract*—This paper describes a new algorithm for finding software clones. It is conceptually independent of the source language of the analyzed programs, working at the level of abstract syntax trees. The algorithm considers that two sequences of statements form a clone if one of them can be obtained from the other by replacing some subtrees. To our knowledge this notion was not previously employed in the literature. It allows to take into account all information on the syntactic structure of a program. We have implemented this algorithm in the tool Clone Digger. It currently supports the Python and Java languages. Clone Digger is free and provided under the GPL license.**

I. Introduction

Different researchers report that the amount of duplicate code in software systems varies from 6.4% - 7.5% to 13% - 20% [1]. Duplicate code can occur as a result of approaches to development and maintenance, due to language or programmer limitations, or simply by accident [1]. Code duplication can be a significant drawback, leading to bad design, and increased probability of bug occurrence and propagation. As a result, it can significantly increase maintenance cost (for instance, any bug in the original has to be fixed in all duplicates), and form a barrier for software evolution. Consequently, duplicate code detectors are a useful class of software analysis tools. Such tools can aid in measuring the quality of software systems and in the process of refactoring. Techniques for detecting duplicate code can be classified according to several criteria. Code can be viewed as similar based on syntactic criteria or at a semantic level (from the point of view of execution effects). In this paper we consider only syntactic similarity. Within this category, duplicate clone detection can be performed at different levels of granularity: strings, tokens, abstract syntax trees, feature vectors [1]. The first two are quite rigid and low-level, therefore we use an approach based on abstract syntax trees.

Two sequences of statements form duplicate code if they are similar enough according to a selected measure of similarity. Such measures can be defined using a set of allowed editing operations and their cost. According to [1] there are three different types of syntactic changes: adding/removing of whitespaces and comments, changing names of variables, and more complex modifications. We aim to detect a wide range of clones, including the third type: e.g., expressions with similar structure.

In essence, we wish to characterize the *structural similarity* of two code fragments in order to determine whether they should be classified as code duplicates. We can formalize this by using the concept of *anti-unifier*, which denotes the most specific generalization of two terms. Anti-unification was first described by Plotkin [2] and Reynolds [3]. In the current paper we use anti-unification to calculate the distance between two abstract syntax trees and group similar trees into equivalence classes called clusters. Anti-unification catches the structural differences between two trees: it allows for instance the replacement of a variable with a more complex expression, but distinguishes between functions of different arities.

Our algorithm of finding duplicates consists of several phases. In the beginning we partition all statements into clusters using anti-unification distance; as a result, the code is abstractly viewed as sequence of cluster identifiers. Next, we find all pairs of identical sequences of cluster IDs. The matching pairs of sequences, which have similar statements in corresponding positions, are now globally checked for similarity. This check is again performed using anti-unification distance, and duplicates are reported if the distance is below a certain threshold.

A fully syntactic abstraction in duplicate clone detection is first reported in [4]. Their algorithm detects a similarity between, e.g., `a[1]` and `a[x+1]` by reducing them to the pattern `a[?]`. This pattern can be seen as anti-unifier of the two expressions. Our work continues this approach based on patterns, extending it to cover more complex programming constructs such as sequences of statements. We use a more natural and flexible way of building patterns; moreover we provide metrics to assess similarity, whose quality increases if the occurrences of the same variable (in the same scope) refers to the same leaf in the abstract syntax tree. Our algorithm is also conceptually independent of the programming language, working at the level of abstract syntax trees.

II. Preliminaries

A. Anti-unification

Anti-unification was first studied in [2], [3]. As the name suggests, given two terms, it produces a more general one that covers both rather than a more specific one as in unification.

Let E_1 and E_2 be two terms. Term E is a generalization of E_1 and E_2 if there exist two substitutions σ_1 and σ_2 such that $\sigma_1(E) = E_1$ and $\sigma_2(E) = E_2$. The most specific generalization of E_1 and E_2 is called anti-unifier. The process of finding an anti-unifier is called anti-unification.

In this paper we use the anti-unification algorithm described in [5].

Anti-unification is originally described for trees. Strictly speaking, the abstract syntax trees we use are not always trees, since leaves containing the same variable references may be merged, but anti-unification can be extended in a straightforward way to directed acyclic graphs.

The anti-unifier tree of two trees T_1 and T_2 is obtained by replacing some subtrees in T_1 and T_2 by special nodes, containing term placeholders which are marked with integers. We will represent such nodes as $?_n$. For example, the anti-unifier of $\mathit{Add}(\mathit{Name}(\mathtt{i}), \mathit{Name}(\mathtt{j}))$ and $\mathit{Add}(\mathit{Name}(\mathtt{n}), \mathit{Const}(1))$ will be $\mathit{Add}(\mathit{Name}(?_1), ?_2)$. In some abstract syntax tree representations occurrences of the same variable refer to the same leaf in a tree. In this case the anti-unifier of $\mathit{Add}(\mathit{Name}(\mathtt{i}), \mathit{Name}(\mathtt{i}))$ and $\mathit{Add}(\mathit{Name}(\mathtt{j}), \mathit{Name}(\mathtt{j}))$ will be $\mathit{Add}(\mathit{Name}(?_1), \mathit{Name}(?_1))$.

B. Anti-unification features

The anti-unifier of two trees represents their "skeleton", inserting placeholders for subtrees which differ. The anti-unifier of a set of trees can be seen as the most specific pattern which matches each tree in the set. Therefore it can be used to store the "average value" of a set of trees. This anti-unification feature was used in [6] to discover widespread patterns of formulas in scientific articles.

An anti-unifier stores only the common top-level tree structure. For example, the anti-unifier of the two trees $\mathit{Add}(\mathit{Add}(\mathit{Name}(\mathtt{a}), \mathit{Name}(\mathtt{b})), \mathit{Name}(\mathtt{c}))$ and $\mathit{Add}(\mathit{Name}(\mathtt{a}), \mathit{Add}(\mathit{Name}(\mathtt{b}), \mathit{Name}(\mathtt{c})))$ will be $\mathit{Add}(?_1, ?_2)$ and therefore it represents the first-level similarity but lacks details on the second level, where the structure of the subtrees does not match.

Let us define the anti-unification distance between two trees. Let U be the anti-unifier of two trees T_1 and T_2 with substitutions σ_1 and σ_2. Let n be the number of placeholders in U. Then σ_1 and σ_2 are mappings from the set $\{?_1, ?_2, \ldots, ?_n\}$ to substituting trees. We define the size of a tree as a number of leaves in it. This notion of size is robust to the particularities of representing abstract syntax trees because it is equal to the number of all name and constant occurrences in the program. Define the anti-unification distance between T_1 and T_2 as a sum of sizes of all substituting trees in σ_1 and σ_2.

For example, consider two trees $\mathit{Add}(\mathit{Name}(\mathtt{i}), \mathit{Name}(\mathtt{j}))$ and $\mathit{Add}(\mathit{Name}(\mathtt{n}), \mathit{Const}(1))$. The anti-unification substitutions are $\sigma_1=\{\mathtt{i}/?_1,\ \mathit{Name}(\mathtt{j})/?_2\}$ and $\sigma_2=\{\mathtt{n}/?_1,\ \mathit{Const}(1)/?_2\}$, the sizes of trees in the substitutions are $|\mathtt{i}|=|\mathtt{n}|=|\mathit{Name}(\mathtt{j})|=|\mathit{Const}(1)|=1$. Therefore the anti-unification distance for this example is 4.

Anti-unification distance can be seen as tree editing distance [7] with a restricted set of operations. It catches the structural differences between two trees and doesn't allow the permutation of siblings or changing the number of child nodes.

III. Duplicate code detection algorithm

Our goal is to find duplicate fragments of code by discovering similarities between sequences of subtrees in the program's abstract syntax tree. The focus on abstract syntax trees allows a flexible level of granularity for our analysis: we can identify, for example, renamed identifiers or modified subexpressions. However, the smallest unit of duplicate code that we report is a statement. We also work with definitions of classes and functions, but since the latter two are very similar cases (their bodies being essentially compound statements), we focus the presentation on statements, for simplicity.

The abstract syntax tree of program is first linearized, i.e., all sequences of statements and definitions are presented in the abstract tree as sibling subtrees. The same approach is used in [8], [9].

Our method of finding duplicate code consists of three phases:

1) Identify similar statements using anti-unification and partition them into clusters. After the first phase each statement is marked with its cluster ID – thus, two statements with the same cluster ID are considered similar in this preliminary view. For example, we can obtain the cluster represented by the anti-unifier $?_1$+=$?_2$, which includes the statements `i+=j` and `m+=2*n` and the cluster $?_1$++, which includes `i++` and `j++`.
2) Find identical sequences of cluster IDs (corresponding to statement sequences within a compound statement). These are candidates to be reported as duplicate code fragments.
3) Refine by examining the identified code sequences for overall similarity. In this phase, every pair of candidate sequences is checked for overall similarity at the statement level, again using anti-unification.

In the very beginning of the whole algorithm an abstract syntax tree for the program is built. Every statement in the program is a root of some subtree. Thus we can compute a similarity measure between two statements by computing the anti-unification distance between two corresponding subtrees.

A. Partitioning similar statements into clusters

```
foreach $tree$ in $statement\_trees$
  $bestcluster$ = argmax($cluster$.add_cost($tree$))
    if $bestcluster$.add_cost($tree$) $<$ $threshold$
      $bestcluster$.append($tree$)
    else
      $clusters$.append(new Cluster($tree$))
```

Fig. 1. Clustering Algorithm

We use a two-pass clustering algorithm. During the first pass of it the most frequent patterns in the source code are discovered. Than we mark each statement by its corresponding pattern.

During this first pass, a preliminary clustering is performed. As discussed above, the anti-unifier of a set of statements can be viewed as its "average value". This can be exploited in the clustering algorithm to avoid comparison with every tree of a set by comparing only with their anti-unifier.

The first pass of the algorithm is shown in Figure III-A. We go over all statements and each new statement is compared with the anti-unifiers of all existing clusters. If an appropriate cluster is found, then the new tree is placed in this cluster, otherwise the tree forms the new cluster.

Let us discuss the required characteristics of the `add_cost` function. Its value should be high in the following cases:

- the cluster is large and its anti-unifier will change significantly after joining the new tree,
- the cluster's anti-unifier is far away from the statement.

Suppose we want to compute the cost of adding a tree `T` to the cluster consisting of `n` trees with anti-unifier `au`. Let `au′` be the result of anti-unification of `T` and `au` with substitutions σ_1 and σ_2: σ_1`(au′) = au`, σ_2`(au′) = T`.

We use the function `add_cost = n*|`σ_1`| + |`σ_2`|`, which satisfies the given requirements.

The clusters grow during the clustering process, therefore their anti-unifiers and the value of the `add_cost` function change. This can lead to the situation when two matching statements can be put into different clusters (although this situation is uncommon, it is very bad). Nevertheless the anti-unifiers of clusters are useful because they can be viewed as the set of widespread patterns. Two statements which are similar to the same pattern are supposed to be similar to one another.

During the second pass all the statements are traversed again and for each statement we search for the most similar pattern from the set produced in the previous pass (again using the anti-unification distance). All the statements marked with the same pattern form the same cluster.

Hashing is used to speed up the clustering process. We follow the approach of [4] using the notion of d-cap. The d-cap of a tree is obtained by replacing all subtrees of the level d and all leaves by placeholders. For example, the 2-cap of $Add(Add(Name(\mathtt{i}), Name(\mathtt{j})), Name(\mathtt{k}))$ is $Add(Add(?_1), Name(?_2))$. Only trees with the same hash value are compared. The depth of d-caps is a parameter of our algorithm.

B. Finding pairs of identical cluster sequences

After the first phase of our algorithm each statement is marked with its cluster ID. We estimate the size of each node in the suffix tree by the size of the anti-unifier of the corresponding cluster. In the second phase we search for all pairs of sequences of statements, which are identically labeled (have the same labels on the same position). Only sequences large enough are considered (according to some selected threshold). This search is performed using a suffix tree approach [10]. Detected pairs are clone candidates and have to be checked for similarity on the statement level in the next phase.

C. Examining code sequences for overall similarity

After the second phase of algorithm we have a set of clone candidates. These candidates are checked as a whole during the third phase. Assume that we have a candidate pair consisting of the following sequences of statements: $\{\mathtt{s}_1, \mathtt{s}_2, \ldots, \mathtt{s}_n\}$ and $\{\mathtt{t}_1, \mathtt{t}_2, \ldots, \mathtt{t}_n\}$. To check this pair for similarity two new trees $\mathtt{Block}(\mathtt{s}_1, \mathtt{s}_2, \ldots, \mathtt{s}_n)$ and $\mathtt{Block}(\mathtt{t}_1, \mathtt{t}_2, \ldots, \mathtt{t}_n)$ are constructed and compared using anti-unification distance. If the distance between them is below a certain threshold then this pair is reported as a clone.

Let us discuss why the third phase is important and why all clone candidates from the second phase can't be reported as real clones. Consider the two sequences from the beginning of the paragraph. It is possible that they differ in each position, therefore they can have n differences and can't be reported as clones. Therefore the third phase is meaningful.

The overall distance between two sequences can't be obtained by summing distances between corresponding statements. Consider two sequences of statements: {`i=0;i+=1;f(i);`} and {`j=0;j+=1;f(j);`}. The distance between corresponding statements is 2, the sum of these values is 6. However, this value doesn't represent a distance because it doesn't use information about shared variables. A more sensible answer is 2 which can be computed by calculating the overall anti-unification distance between the two sequences.

The last example shows that the quality of the algorithm increases if the occurrences of the same variable (in the same scope) refers to one leaf in the abstract syntax tree.

IV. Comparison with existing approaches

There is a large body of prior work in the duplicate code detection field. A comprhensive survey can be found in [1]. Our current work uses the abstract syntax tree approach.

The paper [4] is pioneering by performing fully structural abstraction rather than lexical one. For example, structural abstraction allows to catch the similarity between `a[x]` and `a[y+1]` using the tree pattern `a[?]`. Their algorithm searches for large common patterns in the abstract syntax tree. It is based on heuristics and works in bottom-up manner, specifying and increasing the patterns step-by-step. Anti-unifiers can also be viewed as patterns, the difference is that anti-unifiers can catch the sameness of names (but patterns can be enriched to do it too). In the present paper anti-unifiers are built in top-down manner by enlarging clusters and generalizing their anti-unifiers. It is difficult to compare the method of finding patterns, proposed in [4] and the method of building clusters from the current paper. But the anti-unification based approach is more flexible, because it is based on general notions such as distance between two statements and an "average value" of a set of statements. Therefore another clustering algorithm can be used instead of the chosen one. Though the anti-unification distance between most pairs of statements will be large (because the anti-unifier will be trivial) and not all clustering algorithms fit.

The main advantage of anti-unification over a pattern-based approach is that our algorithm is able to find duplicate sequences of statements and a pattern-based algorithm can find only duplicate statements (we remind that functions and classes are also treated like statements). Suppose that a fragment of duplicate code occupies only a part of two functions. In this case our algorithm will detect this fragment as a whole, while a pattern-based approach will be only able to find a statement-to-statement correspondence.

V. Duplicate code detection tool

Our duplicate code detection tool is called Clone Digger. It is available under GNU General Public License and can be downloaded from the site `http://clonedigger.sourceforge.net`.

Clone Digger is written in Python and thus platform-independent. We use adapters which convert source files into an XML representation of their abstract syntax trees. Currently there are adapters for two languages: Python and Java 1.5. Python abstract syntax trees are built using the standard CPython module ”compiler”. Java trees are built using ANTLR [11]. Adapters for other languages can be created, e.g. by using parser generators or using internal compiler representations.

Clone Digger takes source file names and threshold values as parameters. It produces a HTML file with a list of clones. Each pair is reported statement by statement with a highlighting of differences.

A comparative evaluation of existing clone detection software is performed in the paper [12]. The authors have developed comparison methods and benchmarks and have tested collection of tools in the same conditions. To our knowledge there are two tools that work on the abstract syntax tree level, CloneDR[8] and Asta [4]. Unfortunately, these tools are not publicly available and therefore we cannot compare all tools on the same program text corpora. Anti-unification has many commonalities with the pattern-based approach used in Asta (a more detailed comparison was made in the previous section). The main difference is that our approach handles duplicates consisting of sequences of statements. Therefore the quality of duplicate search is expected to be better than in Asta. The creators of Asta report that over a corpus of Java programs they were able to save 20% of source code size by refactoring found duplicates.

We have tested Clone Digger on sources of some open-source projects. Results can be seen on the tool site (`http://clonedigger.sourceforge.net`).

This work has been supported by INTAS grant (project 05-1000008-8144).

References

[1] C. K. Roy, J. R. Cordy. A Survey on Software Clone Detection Research, 2007.
[2] G. D. Plotkin. A note on inductive generalization. Machine Intelligence, pages 153163, 1970.
[3] J. C. Reynolds. Transformational systems and the algebraic structure of atomic formulas. Machine Intelligence, 5(1):135151, 1970.
[4] W. Evans, C. Fraser, F. Ma. Clone Detection via Structural Abstraction, 2007.
[5] M.H. Sorensen, R. Gluck. An algorithm of generalization in positive supercompilation, In Logic Programming: Proceedings of the International Symposium, MIT Press, 1995.
[6] C. Oancea, C. So, and S. M. Watt. Generalization in Maple. In Ilias S. Kotsireas, editor, Maple Conference 2005, pages 377-382, Waterloo, Ontario, 2005.
[7] P. Bille. A Survey on Tree Edit Distance and Related Problems, 2005.
[8] Ira Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant'Anna, Lorraine Bier. Clone Detection Using Abstract Syntax Trees. International Conference on Software Maintenance (ICSM), 1998
[9] W. Yang. Identifying Syntactic Differences Between Two Programs, 1991.
[10] D. Gusfield. Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology. Cambridge University Press, New York, NY, 1997.
[11] T.J. Parr, R.W. Quong. ANTLR: A Predicated- LL(k) Parser Generator, Software - Practice and Experience, 1995.
[12] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, E. Merlo. Comparison and Evaluation of Clone Detection Tools, Transactions on Software Engineering, 2007.