ABSTRACT
This paper presents a tree-pattern-based method of automatically and accurately finding code clones in program files. Duplicate tree-patterns are first collected by anti-unification algorithm and redundancy-free exhaustive comparisons, and then finally clustered. The algorithm is designed in such a way that the same comparison is not repeated for speed, while thoroughly examining every possible pairs of tree patterns for accuracy. Our method maintains the syntax structure of code in tree-pattern clusters, which gives the flexibility of finding different types of clones while keeping the precision.

Categories and Subject Descriptors
D.2 [Software Engineering]: D.2.7 Distribution, Maintenance, and Enhancement -- Restructuring, reverse engineering, and reengineering, Version control; D.3 [Programming Language]: D.3.3 Language Constructs and Features -- Patterns

General Terms
Algorithms, Management, Languages

Keywords
clone detection, tree-pattern, reverse engineering, software maintenance

1. INTRODUCTION
A duplicate code or a code clone is the set of code fragments that are identical or similar to one another. The software spread with clones is difficult to understand, to uncover bugs, and to make an improvement by modification, eventually having a negative impact on the quality, extensibility and maintainability of the software.

This work was supported by the Engineering Research Center of Excellence Program of Korea Ministry of Education, Science and Technology(MEST) / Korea Science and Engineering Foundation(KOSEF). (R11-2008-007-01003-0)
detection. However, some information may still be lost. Figure 2 shows an example of giving a false positive in such a method. The characteristic vectors of ASTs are identical, but the two code fragments are actually different.

Figure 1. Example: false-negative clones

Figure 2. Example: false-positive clones

We need to find a way to make the best use of structural information of trees in order to keep the precision as high as possible. Thus the comparison unit of our approach is a tree pattern. We exhaustively select all subtree pairs of the two ASTs and collect every duplicate tree-pattern comprising code fragment. Non-identical subtrees are replaced by a hole in a pattern. Thus the tree pattern can be thought as a tree with zero or more hole leaves. Figure 3 shows the duplicate tree-patterns detected by our method, where identical code fragments are all identified without missing anything: five tree-patterns.

Figure 3. Duplicate tree-patterns found by our method

3. DUPLICATE PATTERN-CLUSTERS

In this section, we define a tree pattern that represents the common duplicate code fragments and a pattern cluster that is the collection of identical tree-patterns. We then present an algorithm to find all the duplicate tree-patterns from two given trees and to organize them into clusters of identical tree-patterns.

3.1 Preliminaries

A tree is represented as a term in a ranked alphabet defined as follows.

Definition 1 (ranked alphabet) A ranked alphabet is a tuple \((S, \alpha)\) where \(S\) is a finite set of symbols, \(\alpha\) is a mapping from each symbol in \(S\) to its arity represented by a natural number greater than or equal to 0. That is, the arity of a symbol \(s \in S\) is \(\alpha(s)\). The set of symbols of arity \(k\) is denoted by \(S_k\). That is, \(S_0\) is the set of constant symbols (representing terminal tokens in AST), \(S_1\) is the set of unary symbols, \(\ldots\), \(S_k\) is the set of \(k\)-ary symbols. We assume that \(S\) contains at least one constant.

Definition 2 (trees) The set \(T(S)\) of trees over the ranked alphabet \(S\) is the smallest set inductively defined by:

1. \(S_0 \subseteq T(S)\)
2. if \(k \geq 1, s \in S_k, \text{ and } t_1, \ldots, t_k \in T(S), \text{ then } s(t_1, \ldots, t_k) \in T(S)\).

If a tree \(t\) has the form of \(s(t_1, \ldots, t_k)\), we say that the function symbol \(s\) is a root node of \(t\), and \(t_1, \ldots, t_k\) are immediate subtrees of \(t\). In order for a tree to represent an AST of a program, each node in the tree is decorated with a unique label that represents a program point consisting of its line number and its column number. For example, say \(S = S_2 \cup S_0\) where \(S_2 = \{\text{cons}\}\) and \(S_0 = \{\text{nil}, a\}\). Then a labeled tree, \(\text{cons}(a', \text{cons}(a, \text{cons}(b, \text{nil}))\), where each superscript \(l\) over symbols represents a unique program point in its source program, has five nodes and five subtrees including the tree itself. The graphical view of the tree would be like the right:

Our goal is to find the largest connected common part of two trees. For example, suppose we have two trees \(=(a, +(a, b))\) and
We call this a node in the tree. The largest connected common part of the two trees would be \((a \cdot \neg (a \cdot \#))\) where \(\#\) represents a hole. We call this a tree pattern or just a pattern.

**Definition 3 (tree patterns or patterns)** A tree pattern is a tree with an additional special leaf node called a hole, \(\#\), which is not in the original set of symbols. Every node is decorated with a pair of program-point labels, each of which pinpoints the exact location of patterns in the source code.

In order to find all possible duplicate tree-patterns from two trees \(t_1\) and \(t_2\), every combination of all subtree pairs from the two given trees should be compared. The number of subtree pairs to compare would be \(n_1 \times n_2\) where \(n_1\) is the number of nodes in \(t_1\) and \(n_2\) is the number of nodes in \(t_2\).

For each pair of subtrees, the algorithm compares nodes in each subtree one by one starting from the root node, left-to-right, in a top-down fashion. If two nodes are equivalent, the algorithm builds a tree pattern with the node symbol. It keeps building the tree pattern as long as the nodes are identical. If two nodes, \(n_1\) and \(n_2\), are not identical, then it ignores the entire tree whose root node is \(n_1\) or \(n_2\), and replaces the node with a hole, \(\#\), in the tree pattern. The algorithm repeats the same process until there are no more nodes to compare. For example, suppose the algorithm compares two trees as in Figure 4(1), then it builds a new tree pattern that has 4 nodes and 1 hole as shown in Figure 4(2). Notice that in the pattern, the labels representing program points from two trees are kept as a pair. This process of finding a tree pattern is traditionally known as **anti-unification** [10, 11].

![Figure 4. Example: detection of a pattern](image)

The identical patterns are clustered. Figure 5 shows how two patterns with the same shape are merged into one pattern cluster.

![Figure 5. Example: pattern clustering](image)

**Definition 4 (pattern clusters)** A pattern cluster is the collection of identical patterns. Since a pattern cluster represents more than 2 pairs of duplicate patterns, each node of pattern clusters annotated with a pair of program-point label set.

For instance, if \(\{l_1, l_2\}, \{l_3, l_4\}\) is attached to a node in a pattern cluster, then we have four duplicate patterns whose locations are \((l_1, l_3), (l_1, l_4), (l_2, l_3),\) and \((l_2, l_4)\), respectively.

### 3.2 The algorithm

Finding all the largest common duplicate tree-patterns by exhaustively comparing a pair of trees is an expensive task. However, since it is the only way to thoroughly find exactly identical patterns, we design the algorithm so that the repetition of the same comparison is avoided.

In order to detect the largest possible duplicate patterns and to avoid any repetitive comparisons in the process, the algorithm keeps the record of pairs of nodes already compared. Since the algorithm builds duplicate patterns in a top-down fashion, the pair of subtrees whose root nodes are already in the list can be safely ignored since the duplicate pattern of the pair will be the proper subtree of an already found pattern. The algorithm is divided into two parts: duplicate tree-pattern detection and clustering.

First, the algorithm needs to find all possible subtree pairs of two input trees, and compares each pair. In order to avoid the repetitive comparisons, a two-dimensional array of type **work-table** is maintained. The array remembers the pairs that have been

![Algorithm Duplicate Tree-Pattern Detection](image)
When finding a duplicate pattern from two given trees, the anti-unification is applied. Our anti-unification algorithm is described as a recursive function anti-unify. While it builds up a tree pattern, it annotates the pair of program-point labels to every node in the pattern. Identified duplicate tree-patterns are collected into tree-patterns indexed by its root node.

Once all the duplicate tree-patterns are identified, identical duplicate patterns are merged into a pattern cluster. Function clustering clusters a set of identically shaped patterns. First, one pattern is picked from pattern set and transformed into a singleton pattern cluster. Until there is no more pattern to cluster, we repeatedly pick one pattern at a time and put it into the right pattern cluster.

4. IMPLEMENTATION

We implemented our tree-pattern-based duplicate code detection algorithm in Objective Caml 3.09, targeting for finding clones in C programs. We used CIL 1.3.6 (C Intermediate Language) parser for parsing C programs. Our experiment has been performed on a Workstation at Xeon 1.60 GHz Dual Core processor and 2GB RAM running Linux 2.6.22.

In order to see how effective our identical duplicate pattern detection method, the three well-known applications with long history of maintenance, Tar, Apache Server, and MySQL, were selected for experiment. Among many different versions, we selected the latest version of each application program: Tar 1.9, Apache Web Server 2.2.8 and MySQL 6.0.3 1. Details of our experimental results will be included in the later version of this paper.

5. CODE-CLONE DETECTION

Once we have duplicate tree-pattern clusters, we can identify code clones of type 1 through type 3. When we build pattern clusters, we can also collect subtrees replaced by holes. The hole size (the size of a subtree replaced by the hole) and other characteristics of the subtrees are used to determine clones of different types. We show how the pattern clusters are used to detect code clones by examining clone types by Koschke’s categorization [7].

• Exact Clone (type 1) : Pattern clusters with no hole are exact clones.

• Parameter-substituted clone (type 2) : Pattern clusters with holes whose size is less than equal to 2 are either consistent or inconsistent parameter-substituted clones. If a tree pattern is α-convergent, then it is consistent. Otherwise, it is inconsistent.

• Structure-substituted clone : Pattern clusters with holes whose size is greater than 2 are either consistent or inconsistent structure-substituted clones. In order to find consistent structure-substituted clones, code fragments replaced by holes must be examined. A simple α-convergent checking in addition to the consistency checking of code fragments will do the job.

• Non-contiguous clone (type 3) : In order to detect clones of this kind, the strategy employed by Bulychev et al. [3] has to be employed. We leave this as a future work.

6. RELATED WORK

In this section, we survey some tree-based clone-detection techniques.


---

1 We have also done experiment for the following program versions: Tar 1.15, 1.16, 1.17, 1.18, Apache web server 2.2.0, 2.2.2, 2.2.4, and MySQL 4.1.22, 5.0.51, 5.1.22. Due to space limitation, we do not report the detailed data here.
program are computed to approximate the structural information in the Euclidean space. Its clever characterization of a tree makes it to be scalable, but may result in false positives due to the lost of some structural information.

Evans et al.’s Asta [4] proposes to compare ASTs by finding clones in the form of trees with gaps, which are essentially the same as tree-patterns in our paper. It uses dynamic programming (bottom-up approach) to avoid repeated comparison. In order to save the computation time further, it employs some clever heuristics. Bulychev et al. proposed a method of detecting duplicate codes using anti-unification [3]. The fundamental difference is the unit of detected tree pattern. Its clone unit is restricted to a statement or statements sequence, but ours has no restriction.

Wahler et al. [14] uses the frequent itemset technique for clone detection to detect exact and parameterized clones in the XML representation of AST. Koschke et al. [8] and Tairas et al. [13] introduced linear-time clone-detection algorithms based on suffix tree. However, these approaches do not keep the syntax structure intact.

7. CONCLUSION
In this paper, we propose a method of finding code clones using duplicate tree-pattern clusters collected by anti-unification algorithm, redundancy-free exhaustive comparisons, and clustering. Our method maintains the syntax structure of code in tree-pattern clusters, which gives the flexibility of finding different types of clones while keeping the precision.

We expect the method to be effectively applied to the version control in software configuration management and to find refactorable clones. Moreover, with some appropriate abstraction of ASTs, the method could be used to find copy-and-paste bugs and code plagiarism detection.

8. REFERENCES