Towards Flexible Code Clone Detection, Management, and Refactoring in IDE

Minhaz F. Zibran  
Chanchal K. Roy  
Department of Computer Science, University of Saskatchewan, Saskatoon, SK, Canada S7N 5C9  
{minhaz.zibran, chanchal.roy}@usask.ca

ABSTRACT

In this paper, we propose an IDE-based clone management system to flexibly detect, manage, and refactor both exact and near-miss code clones. Using a k-difference hybrid suffix tree algorithm we can efficiently detect both exact and near-miss clones. We have implemented the algorithm as a plugin to the Eclipse IDE, and have been extending this for real-time code clone management with semi-automated refactoring support during the actual development process.

Categories and Subject Descriptors
D.2.7 [Software Engineering]: Distribution, Maintenance, and Enhancement—restructuring, and reverse engineering

General Terms
Algorithm, Design, Management

Keywords
Clone analysis, detection, refactoring, maintenance

1. INTRODUCTION

Over the past decade several techniques and tools for detecting code clones have been proposed having their own strengths and weaknesses [5]. While most of them are capable of detecting Type-1 (exactly similar code fragments except for white-spaces and formatting) and Type-2 (syntactically similar code snippets, where identifiers/variables can be renamed) clones, only a few of them are reported to detect Type-3 (where one or more lines of code can be added/modified/removed) clones. However, it is not enough to only detect code clones. Code clones are required to be tracked, managed, and possibly should be removed through refactoring wherever feasible. And support for such activities should be integrated with the IDEs for blending clone management with actual development effort. However, most clone detectors are developed as separate tools. Those few tools that are integrated with IDEs are mostly focussed in detecting Type-1 and Type-2 clones, and are yet to offer sufficient support for flexible clone management and refactoring. To address these issues, we propose an IDE-based clone management system for accurate and flexible detection, management, and refactoring of both exact (Type-1) and near-miss (Type-2 and Type-3) code clones.

2. OUR APPROACH

Accurate detection of code clones is the fundamental and vital step towards clone management and refactoring. We have developed a language independent matching engine (LIME), a tool for fast localization of all k-difference (edit distance) occurrences of one code fragment inside another. On top of LIME, we have developed a near-miss clone detection tool as a plugin to the Eclipse IDE. Figure 1 presents a schematic diagram of the major algorithmic modules and the process of clone detection used in our approach.

2.1 Clone Detection

We detect code clones applying a multiphase approach. At the very first (code preprocessing) phase, we generate AST (Abstract Syntax Tree) for the source code, filter out the comments, and extract code snippets of desired granularity such as functions and/or blocks. Then we normalize the snippets by uniformly formatting them and consistently renaming the identifiers.

In the next (matching) phase, the normalized code snippets are passed to LIME, which first fingerprints the snippets by applying Rabin’s linear time fingerprinting algorithm [4] on each match unit. A match unit may be a token/word or a line of code. Thus, a ‘fingerprinted’ code snippet consists of a sequence fingerprints, which are essentially numeric values.

LIME then concatenates all fingerprint sequences and generates a generalized suffix tree (GST) using Ukkonen’s linear algorithm [7]. Concatenation of the edge-labels on the paths from the root to the non-leaf nodes of the GST yields the sets of all sequences common in the fragments. Tracing back to the original source code of the fingerprint sequences identifies the Type-1 and Type-2 clones. To the best of our knowledge, CCFinder, CloneDigger, Dup, and the rest of the suffix-tree-based clone detectors exploit suffix trees up to this level with or without fingerprinting the source code [5]. However, LIME goes beyond this, and further processes the GST in linear time, to enable finding LCE (Longest Common Extension) in constant time. Given a pair of sequences
$S_1$ and $S_2$, and an index pair $(i, j)$ where $i$ and $j$ refer to
positions in $S_1$ and $S_2$ respectively, the $LCE$ between the
sequences is the longest subsequence of $S_1$ starting at position
$i$ that matches a subsequence of $S_2$ starting at the $j^{th}$
position.

Having the GST preprocessed, LIME then applies a $k$-
difference hybrid dynamic programming algorithm [3] to de-
tect Type-3 clones. Given two sequences $T$ and $P$ of lengths
$m$ and $n$ respectively ($n \leq m$), the algorithm finds all end
locations in $T$ where $P$ matches with at most $k$ differences (edit
distance) in $O(km)$ time and $O(m + n)$ space complex-
ities. Here, $k = \left\lfloor (n \times \theta)/100 \right\rfloor$, $0 \leq \theta \leq 100$, and $\theta$ is the
user-defined dissimilarity threshold.

As the clone pairs are identified, they are clustered into
groups based on their similarities, and the results are re-
ported to the user through Eclipse’s interactive Tree View.
Our tool also augments Eclipse’s search engine by introduc-
ing the facility to find all exact and near-miss cloned copies
within a chosen boundary (selected files, packages/directo-
ries, projects, or the entire workspace) for any code fragment
selected in the editor.

2.2 Clone Management and Refactoring

To facilitate cost-effective semi-automatic management and
refactoring of exact and near-miss clones, we have been
working on the following areas.

Incremental Detection. Clone detection in a large code
base can consume significant amount of time and resource.
On the other hand, clone management during the devel-
opment process demands quick response. Hence, the IDE-
integrated clone management tools should preserve the ini-
tial clone detection results, track changes in the code corpus,
and incrementally update the clone detection results by com-
paring the modified and newly added code fragments to the
existing results. Moreover, the clone detection results should
also be carefully updated to remove references to any deleted
source code.

Clone Refactoring. To support consistent modification of
code clone groups, a number of tools support simultaneous edit-
ing for Type-1 and Type-2 clones [5] including the work of
Hou et al. [2]. For near-miss clones (specially, Type-3), sup-
port for edit propagation is also necessary, where the edit
operations on a code snippet can be semi-automatically ap-
p lied to all its cloned fragments as well. In addition to the
support for rename refactoring, earlier research [1] identi-
fied that extract function and pull-up method refactoring
patterns could be promising towards code clone refactoring.
In this regard, we propose a two-phase approach for object-
oriented code base.

In the first phase, extract method refactoring pattern is
applied. For each class, the cloned fragments that do not
constitute the entire method bodies, are identified as refac-
toring candidates. Then those fragments can be replaced
by calls to a newly introduced method that unifies all those
cloned fragments in one place.

The second phase applies pull-up method. To find the
refactoring candidates, all method level clones across all
classes are identified. If classes containing such methods
possess a common superclass, those methods are removed
from all those classes, and a generalized method is intro-
duced in the common superclass. If, in case, those classes
do not share a common superclass, an abstract class can be
introduced as a common superclass, to which the methods
can be pulled up. This two-phase refactoring approach, with
minor tuning, can also be applied to procedural code.

Refactoring Schedule. Effective application of the refac-
toring candidates is likely to be a cumbersome task. Under-
lying activities such as the identifier renaming, redefinition
of method signature, and parameter reordering are likely
to introduce interdependencies and conflicts. There may
also be certain restrictions and priorities from the organiza-
tion’s side due to limited time and resource. Given the re-
strictions and limited resources, only a subset of refactoring
candidates may be required to have chosen for application,
where the target remains maximizing the code/design quality
while minimizing the efforts. However, different choices of
refactorings may incur distinguishable impact on the quali-
ity. Thus, a flexible way to plan for the refactoring schedule
is also necessary. We plan to model such a scheduling as a
constraint satisfaction optimization problem, and incorpo-
rate a smart refactoring scheduler with the clone manage-
ment system.

Refactoring Verification. The purpose of code clone
refactoring is mainly to restructure the source code for en-
hancing maintainability without altering its functionality.
Therefore, we believe, as the refactoring patterns are ap-
p lied, test cases should automatically be generated to ver-
ify that those refactorings do not change the program be-

3. CONCLUSION

This paper presents our ongoing work towards an IDE-
based clone management and refactoring tool. We have al-
ready implemented the clone detection part of this system
and conducted an empirical study on identifying both exact
and near-miss clones in Weltlab and PostGreSQL, and com-
pared the results with NiCad [6]. We experienced that our
algorithm reported almost no false positives, and detected
all the clones that NiCad detected. We believe, once com-
pleted, our clone management system will significantly help
the clone community and industry practitioners in dealing
with both exact and near-miss clones.

Acknowledgments: This work is supported in part by
the Natural Sciences and Engineering Research Council of
Canada (NSERC).

4. REFERENCES

Refactoring support based on code clone analysis. In
environment for the proactive management of
approximate string matching. J. Algorithms, 10(2):
and evaluation of code clone detection techniques and
tools: a qualitative approach. Sc. of Com. Prog., 74:
detection of near-miss intentional clones using flexible
pretty-printing and code normalization. In ICPC’08,