Mapping Code Clones Using Incremental Clone Detection

Nils Göde
University of Bremen, Germany
http://www.informatik.uni-bremen.de/st/
nils@informatik.uni-bremen.de

Abstract
Understanding the evolution of duplicated source code is necessary to investigate the phenomenon of cloning. To trace individual code clones across multiple program versions, clones of consecutive versions need to be mapped. Current methods detect clones of different versions first and then map detected clones retroactively.

I present an approach to map individual clones between consecutive program versions during clone detection. The method is integrated into a token-based incremental clone detection algorithm. Clones are mapped according to the changes made to the source files of the program between versions.

1 Introduction
Studying the evolution of code clones requires detecting duplicated source code in multiple versions of a program. Clones of consecutive versions need to be mapped to trace individual clones. We presented the first incremental clone detection algorithm designed for detecting clones in consecutive versions of a program [3, 4]. Apart from the benefit in run-time compared to conventional approaches, the algorithm provides the foundation to map clones during clone detection, making a retroactive mapping superfluous.

This paper presents an extension to the incremental algorithm that allows to trace clones across program versions during clone detection. The approach overcomes the problems of a retrospective mapping, which are the computational complexity and the imprecision of heuristics being used.

For the remainder of this paper, the term fragment refers to a section of source code that has a well-defined location and is similar or identical to at least one other fragment. Two or more similar fragments are grouped in a clone class.

2 Related Research
Different approaches to map clones between program versions have been presented. Kim et al. first relate fragments based on the location overlapping and their textual similarity [6]. Clone classes are then mapped according to the fragment relation. Considering extensive changes, a drawback is that clones might not be mapped correctly due to the large difference between the old and the modified versions of fragments. Aversano et al. detect clones in a single version and deduce the relation to clones of following versions from the changes made to the source code [1]. A similar method has been applied by Krinke [7]. Both approaches lack the ability to detect new clones in later versions. Bakota et al. map fragments based on information from the abstract syntax tree [2]. A common problem of all approaches is their high computational cost. A more detailed discussion of problems related to mapping clones can be found in [5].

3 Mapping Clones
Existing approaches map clones either on a low level by matching fragments or on a high level by matching clone classes. The drawback of matching classes is that a class can potentially have an arbitrary number of ancestors or descendants. Choosing one of these to be the “true” ancestor or descendant restricts possible interpretations of the mapping. To map individual fragments instead of clone classes allows a one-to-one mapping between fragments instead. A mapping between classes can then be derived from the fragment mapping according to the respective usage scenario.

Let $x$ and $y$ ($x < y$) be two versions of source code. Then $\text{mod}^y_x(f_x) = f_y$ denotes the changes applied to fragment $f_x$ in version $x$ resulting in $f_y$ in version $y$. The problem of mapping fragments from version $i - 1$ to $i$ is finding $f_{i-1}$ such that $\text{mod}^y_x(f_{i-1}) = f_i$ for each $f$ in version $i$. $f_{i-1}$ is called the ancestor of $f_i$. It may happen that there exists no $f_{i-1}$ meaning $f_{i}$ has been newly introduced in version $i$.

A problem arises from fragments that disappear temporarily because the respective clone class is changed inconsistently. When the change is made consistent in a later version, the reappearing fragment is regarded as a new fragment. To detect temporarily disappearing fragments, fragments that have been part of a clone class in an earlier version and are currently not part of any class are tracked further on. Any such fragment is called a ghost fragment. Note that ghost fragments are just an internal representa-
tion and are not reported as clones to the user.

To detect reappearing fragments, we allow the an-
cestor of a fragment to be in any of the earlier versions,
not just the previous. For that, the notion of the an-
cestor for fragment $f_i$ is extended to $f_{i-1}$ such that
$\text{mod}^{-1}_i(f_{i-1}) = f_i$. Because the ancestor refers to
the most recent occurrence only, it is required that each
fragment $f_{i-k}$ with $0 < k < l$ and $\text{mod}^{-k}_i(f_{i-k}) = f_i$
is a ghost fragment.

My extension to the incremental clone detection
algorithm creates the mapping among fragments by
calculating the ancestor for each fragment $f_i$ for each
version $i$.

4 Tracing Procedure

Although the mapping is based on fragments, the
difficulty is that fragments are defined only in relation
to each other. That means for any version $i$, $\text{mod}^{-1}_i(f_{i-1}) = f_i$ is known. Still there is no guar-
antee that $f_i$ exists as a normal fragment, because
changes made to other fragments of the clone class
might result in $f_i$ remaining the only fragment of the
class and the class disappears in case of which $f_i$ be-
comes a ghost fragment. To solve this, the basic idea
of my approach is to calculate $f_i$ for every $f_{i-1}$ be-
longing to a class that has at least one fragment in a
changed file and assume $f_i$ to be a ghost fragment by
default. It is assured that any of these fragments that
has wrongly been set to be a ghost is redetected in a
later phase and its state set to normal again.

For each version $i$ that is to be analyzed, our in-
cremental algorithm performs the following phases:
(1) Process changed files, (2) process existing clone
classes, (3) detect new clone classes, (4) merge new
and existing fragments. My extensions are as follows.

Prior to processing changed files, the ancestor of
each fragment is set to the fragment’s occurrence in
the previous version unless the fragment is a ghost
fragment.

(1) When processing changed files, the lines that
have changed for a particular file are given as input
to the algorithm. The line information is transformed
to token information giving a higher precision. The
source code locations and bounds of all fragments (in-
cluding ghost fragments) contained in the file are ad-
justed according to the previously calculated token changes.

(2) After all changed files have been processed, the
fragments of existing clone classes are updated. As
soon as a single fragment of a class is contained in a
changed file, all fragments of the class are set to be
ghost fragments, because at this point it is not known
whether the class and its fragments continue to exist.

(3) New clone classes are retrieved. The set of
newly detected classes contains all classes of which
at least one fragment is contained in a changed file.

(4) Within the last phase, the fragments of the new-
ly detected classes are processed. For each frag-
ment it is checked whether the fragment already ex-
ists. The existence of a fragment can be checked in
constant time, because every fragment can clearly be
identified by its first and last token and the file it
is contained in. If it does not exist, the fragment is
truly a new one and has therefore no ancestor. If the
fragment exists and is a ghost fragment, the existing
fragment’s state is set to normal. Note that the exist-
ing fragment’s ancestor has already been set correctly
prior to the first phase.

When all newly detected fragments have been pro-
cessed, every fragment has been correctly mapped to
its previous occurrence. The mapping from the previ-
ous version $i − 1$ to the current version $i$ has therefore
been completely established.

5 Conclusion

Analyzing the evolution of code clones is essential to
study the phenomenon of cloning. This requires map-
ning clones of different program versions. I presented
an extension to our incremental clone detection algo-
gram that traces clones during the clone detection
process. The algorithm identifies the previous occur-
rence of each fragment by considering the changes
made to the files’ token sequences. Furthermore, it
keeps track of fragments that are temporarily not part
of a clone class. The low-level mapping of fragments
provides the foundation for deriving high-level evolu-
tion patterns among clone classes.

References

are maintained: An empirical study. In Proceedings
of the 11th European Conference on Software Main-
tenance and Reengineering, pages 81–90. IEEE Com-

in software evolution. In Proceedings of the 23rd Inter-
national Conference on Software Maintenance, pages

University of Bremen, 2008.

In Proceedings of the 13th European Conference on
Software Maintenance and Reengineering, pages 219–

In Workshop Proceedings of the 13th European Con-
ference on Software Maintenance and Reengineering,

empirical study of code clone genealogies. In Proceed-
ings of the Joint 10th European Software Engineer-
ing Conference and the 13th ACM SIGSOFT Sym-
posium on the Foundations of Software Engineering,

changes to code clones. In Proceedings of the 14th
Working Conference on Reverse Engineering, pages