Tracking Code Clones in Evolving Software

Ekwa Duala-Ekoko and Martin P. Robillard
School of Computer Science
McGill University
Montréal, QC, Canada
{ekwa,martin}@cs.mcgill.ca

Abstract

Code clones are generally considered harmful in software development, and the predominant approach is to try to eliminate them through refactoring. However, recent research has provided evidence that it may not always be practical, feasible, or cost-effective to eliminate certain clone groups. We propose a technique for tracking clones in evolving software. Our technique relies on the concept of abstract clone region descriptors (CRD), which describe clone regions within methods in a robust way that is independent from the exact text of the clone region or its location in a file. We present our definition of CRDs, and describe a complete clone tracking system capable of producing CRDs from the output of a clone detection tool, notify developers of modifications to clone regions, and support the simultaneous editing of clone regions. We report on two experiments and a case study conducted to assess the performance and usefulness of our approach.

1. Introduction

Software systems often contain numerous *code clones*, or groups of source code regions that match each other with varying degrees of exactness. Code clones surface in software systems for a number of reasons, including the difficulty of factoring out functionality using programming language constructs, the requirement to avoid dependencies between modules, the practice of writing code by example [15, 19], and the use of idioms for framework extensions.

Whatever the cause, the presence of code clones in a system means that code that realizes identical or similar logic is not co-located. This duplication of implementation logic often leads to a necessity to modify multiple sections of code consistently [5]. Oversights in that respect often lead to regression faults. For these reasons, much effort has been spent on the detection and removal of code clones from software systems [22]. Technology to scan the source code of a system and identify clones of varying similarity is now readily available. Once identified, clones can be removed through source code refactoring [1, 4, 9].

In recent years, the traditional notion that code clones should be eliminated as a general rule has met with resistance. In particular, Kim et al. challenged the belief that code clones necessarily represent a clear and immediate negative quality factor for a software system. In a study of programming practices in an industry setting [11], Kim et al. found that "skilled programmers often created and managed code clones with clear intent" [13, p. 187]. A later study of code clone genealogies also provided evidence of code clones that are difficult or impossible to refactor using standard techniques, and of code clones that evolve into distinct code [13]. These observations indicate that, in certain situations, it might be beneficial to maintain clone groups as such. Unfortunately, merely accepting the presence of code clones does not by itself mitigate the problems they cause; consequently, strategies must be sought to deal effectively with code clones during software development.

We propose a technique for documenting and monitoring clones in evolving software. Our technique relies on a heuristic representation for clone regions that identifies source code locations within methods using a combination of syntactic, structural, and lexical information. Our abstract representation, called *clone region descriptor* (CRD), goes beyond simple line of code-based clone descriptions, and supports the tracking of clone regions in different versions of a software system.

We developed a complete clone tracking system called *CloneTracker*. Our system takes as input the output of a clone detection tool and automatically produces CRDs to represent the clone regions for different clone groups. Using CRDs, CloneTracker can automatically track clones as the code evolves, notify developers of modifications to clone regions, and support simultaneous editing of clone regions. This way, software developers can specify clone groups they wish to track once and carry on with all their future modification tasks with the knowledge that modifications to clone regions will be detected and supported. Alternatively, clone regions can be inspected at any time to reason about their properties (e.g., to plan a refactoring). In brief, CRDs provide a lightweight way to monitor clones without having to run a clone detection tool every time the code of the system changes.

The contributions of this paper include the description of the first realization of a complete clone tracking technique, an empirical evaluation of its accuracy, and a number of case studies of clone evolution that provide evidence of the usefulness of this approach.

The rest of the paper is organized as follows. In Section 2, we present a real example of clone evolution and explain the difficulties associated with tracking clones. In Sections 3–5, we describe our complete clone tracking system, the details of our clone region representation, and our simultaneous editing algorithm. We report on the quantitative evaluation of our approach in Section 6 and on our case studies in Section 7. We discuss related work in Section 8 and conclude in Section 9.

2. Motivation

We illustrate the present state of the practice for clone detection and the need for advanced clone tracking techniques with a small case study of the jEdit system.[1]

A developer working on a modification to release 4.0-final (63 kLOC) informally observes a number of code clones and decides to run a clone detection tool on the system. Using SimScan,[2] the developer sets a number of search options (Volume="medium", Similarity="fairly similar", Quality/speed="fast") and runs the tool. After approximately 32 minutes (WindowsXP, Pentium4-3GHz, 512MB), the detection completes and returns a list of 251 *clone groups* comprising between 2 and 137 *clone regions* (or individual clones). Each clone region is represented in terms of a file name and a line range. Browsing the results, the developer notices a clone group of potential interest: in class `bsh.Reflect`, a large `for` block in method `findExtendedConstructor` is a clone of a similar region in method `findExtendMethod`.

A detailed study of this clone group provides evidence in direct support of all three main results of the study of Kim et al.

> ...clones impose obstacles during software development because they often change similarly with their counterparts in the same group... [13, p. 187].

Both regions changed consistently for version 4.2-pre2 (change of an exception type), and for version 4.2-pre4 (major cleanup of the code that preserved the clone relation).

> ...popular refactoring techniques [...] cannot easily remove many long-lived clones... [13, p. 188].

Both regions are in methods that have a different return type, which makes them non-locally-refactorable using standard refactorings [4].

> ...we found that many clones were volatile... [13, p. 188]

Our clone group disappeared in release 4.2-pre10. As claimed by Kim et al., and further illustrated by this example, there are many situations in which it may not prove cost-effective or even possible to refactor clones. In such cases, developers must manage clone groups as they evolve. This is no small task in a code base that is in constant evolution. Specifically, without dedicated support, developers who wish to maintain and evolve code clones are faced with the following challenges:

- Current clone detection technology produces descriptions of clone regions in terms of ranges of lines of code. Such descriptions are invalidated as soon as any of the code changes (whether or not the changes are in a clone region).
- Clone regions must be modified individually. Existing clone relations might be overlooked.
- Inconsistent changes to clone groups cannot be detected without re-running the clone detection tool, an alternative that is too computationally costly to be used interactively during software development. As illustrated earlier in this section, a 63kLOC project took 32 minutes to analyze using a popular tool.
- Information about clone groups of interest cannot be reused in future tasks unless code detection techniques are re-applied.

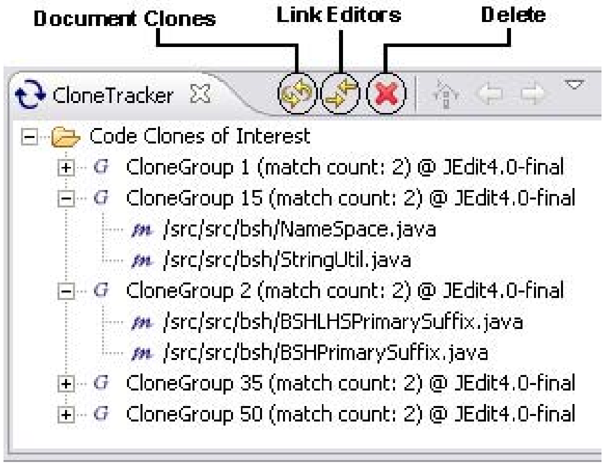
3. Clone Tacking Approach

In this section, we present an overview of CloneTracker from the perspective of a user of the system. We postpone the presentation of the heuristics and algorithms enabling this technology until Sections 4 and 5.

Our clone tracking approach complements existing clone detection technology. Our current version of CloneTracker relies on the SimScan clone detection tool, but can easily be adapted to use other tools. CloneTracker is fully integrated with the Eclipse Platform [18]. Eclipse is an integrated development environment with an architecture that supports the addition of components, called plug-ins, that add to the environment's functionality. The standard distribution of Eclipse includes a set of plug-ins that provide extensive support for development in Java. CloneTracker is implemented as an Eclipse plug-in.

[1] www.jedit.org

[2] blue-edge.bg/simscan



Code Clones of Interest
CloneGroup 1 (match count: 2) @ JEdit4.0-final
CloneGroup 15 (match count: 2) @ JEdit4.0-final
/src/src/bsh/NameSpace.java
/src/src/bsh/StringUtil.java
CloneGroup 2 (match count: 2) @ JEdit4.0-final
/src/src/bsh/BSHLHSPrimarySuffix.java
/src/src/bsh/BSHPrimarySuffix.java
CloneGroup 35 (match count: 2) @ JEdit4.0-final
CloneGroup 50 (match count: 2) @ JEdit4.0-final

Figure 1. Clone Documentation View

With CloneTracker, a developer concerned about the presence of code clones in a system first triggers the execution of SimScan. SimScan produces a tree view describing, as roots, all of the clone groups detected in the target system and, for each group, a list of clone regions containing the similar code. At this point, clone regions are represented in terms of file name and line range. The developer can inspect the code clones by selecting them in a view. With CloneTracker activated, a *clone documentation* view also becomes available in Eclipse (Figure 1).

Once the developer identifies clone groups of interest, the groups can be transfered from the SimScan results view to the clone documentation view through a drag-and-drop operation. To produce a long-term description of a clone group, the developer clicks the *Document Clones* button (see Figure 1). CloneTracker then automatically translates the location of all clone regions into clone region descriptors (see Section 4), which form an active clone model. The clone model now describes clones in a way that is resilient to changes to file names, changes in code blocks preceding the documented clone regions, or changes within the clone regions themselves.

The developer can then start working on the system. If, at any point, the developer edits code in a clone region represented in the clone model, CloneTracker produces a notification that clone regions are being edited. CloneTracker also opens and highlights a sibling of the clone region that is about to be modified (Figure 2). In Figure 2, the blue margin makers show the range of the two clone regions within their respective files.

At that point the developer can decide to link the two editors (by clicking on the *Link Editors* button, see Figure 1) and proceed with the modification of the current clone region. Modifications made within common sub-regions of both clone regions are echoed from the active clone region to the sibling clone region. Section 5 describes the mapping algorithms used for the simultaneous editing feature of CloneTracker. After this task is completed, other developers (with or without access to CloneTracker) can check out, modify, and commit the files containing the clone regions. The next time the initial developer accesses the code, the clone regions are again automatically detected.

4. Clone Modeling Technique

One of the main requirements for our clone tracking approach is to be able to track clone regions independently of their location in a source file. Although this can be achieved easily for clone regions that align with method boundaries, it is much more difficult to do so for regions *within* methods since such regions are typically not labeled or uniquely identified. One possibility is to store the text of code clones and to track their location in different versions of the code using code matching techniques [12]. However, for our purpose (rapid, interactive tracking of clone regions), we needed to investigate more lightweight alternatives.

To gather insights about potential ways to describe clone regions, we manually inspected about 600 clone regions from four different projects involving tens of different developers (ConcernMapper [21], jEdit, JBossAOP,[3] and FreeMind[4]). While looking at these clone regions, we tried to determine what unique characteristics of the clones regions could help us define and locate them in a way that would resist a certain degree of change (textual modifications before, after, and within the clone region, changes to the name of the file in which the region is located, etc.). To this effect, we made a number of observations.

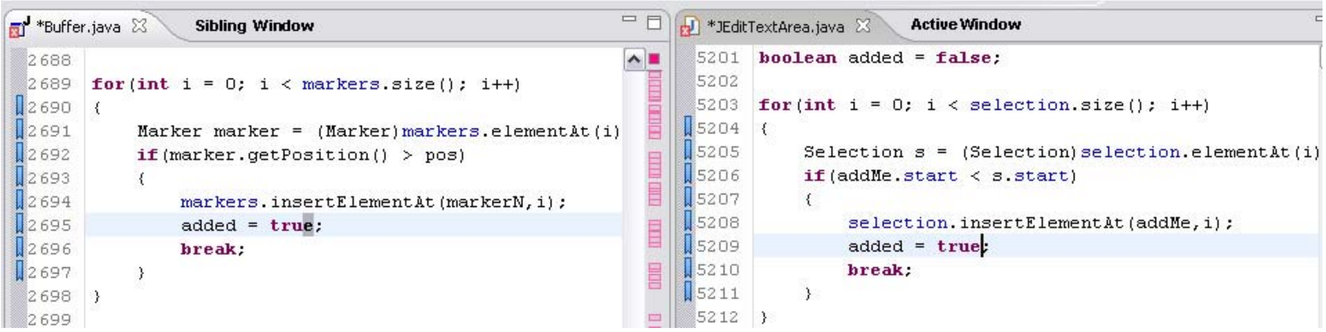
1. *Clone regions are generally constrained within the boundaries of major code blocks (e.g., method boundaries, conditional branches, looping blocks).*
2. *Some structural elements (e.g., loop-termination predicates, branching predicates, exception lists) tend to be unique at a given level of nesting.*

Based on these insights and on our general experience inspecting clone regions, we designed a technique for locating clone regions that uses a combination of the structural properties, lexical layout, and similarities of the clone regions. In the rest of this section, we describe our clone region description model and our algorithm for locating clone regions based on the model. Section 6.1 reports on how we evaluated that the abovementioned observations and the heuristics we derived from them held, and to which extent.

4.1. Clone Region Descriptors (CRDs)

A *Clone Region Descriptor* (CRD) is a lightweight and abstract description of the location of a clone region in a code base. The idea is to provide an approximate location

[3] labs.jboss.com/portal/jbossaop/
[4] freemind.sourceforge.net



```
for(int i = 0; i < markers.size(); i++)
{
    Marker marker = (Marker)markers.elementAt(i)
    if(marker.getPosition() > pos)
    {
        markers.insertElementAt(markerN,i);
        added = true;
        break;
    }
}
```

```
boolean added = false;

for(int i = 0; i < selection.size(); i++)
{
    Selection s = (Selection)selection.elementAt(i)
    if(addMe.start < s.start)
    {
        selection.insertElementAt(addMe,i);
        added = true;
        break;
    }
}
```

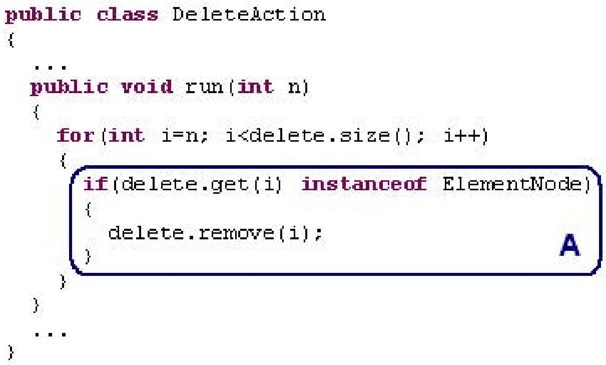
Figure 2. Editing Clone Regions

```
<CRD> ::= <file> <class> <CM> [<method>]
<method> ::= <signature> <CM> <block>*
<block> ::= <btype> <anchor> <CM>
<btype> ::= 'for' | 'while' | 'do' | 'if' |
            'switch' | 'try' | 'catch'
```

Figure 3. Definition of CRDs in Extended BNF

that is independent from specifications based on lines of source code, annotations, or other similarly fragile markers. Figure 3 shows our current definition of a CRD in extended Backus-Naur form (terminal symbols are in italics).

Essentially, a CRD represents the characteristics of each block in which a clone region is nested. With CRDs, clone regions always align with blocks. At the top level, a CRD consists of the name of the enclosing file (`<file>`), the name of the enclosing class (`<class>`), a *corroboration metric* (`<CM>`, explained below), and an optional method descriptor (`<method>`). When entire classes are clones of each other, the method descriptor is not used. The method descriptor consists of a canonical representation of the method's signature (`<signature>`), the corroboration metric (`<CM>`), and zero or more block descriptors (`<block>`). When the clone region aligns with method boundaries, there is no block descriptor. In other cases, block descriptors describe blocks in which the region is nested (in the nesting order). Finally, a block descriptor consists of a description of the block type (`<btype>`), a string describing a distinguishing identifier for the block (`<anchor>`), and the corroboration metric (`<CM>`). The different block types currently supported are listed as part of the `<btype>` non-terminal symbol in Figure 3. For the anchor, different schemes are possible. As an initial investigation, we are currently using the textual representation of a distinctive statement associated with the block. For loops, we use the termination statement. For `if` statements, we use the branching predicate.[5] For `switch` statements, we use the switch expression. For `try` blocks, we use the list of exception types caught in catch clauses associated with the block. For `catch` blocks, we use the type of the exception caught.



```
public class DeleteAction
{
  ...
  public void run(int n)
  {
    for(int i=n; i<delete.size(); i++)
    {
      if(delete.get(i) instanceof ElementNode)
      {
        delete.remove(i);
      }
    }
  }
  ...
}
```

Figure 4. Block Represented by a CRD

For example, the CRD for code block A in Figure 4 is:

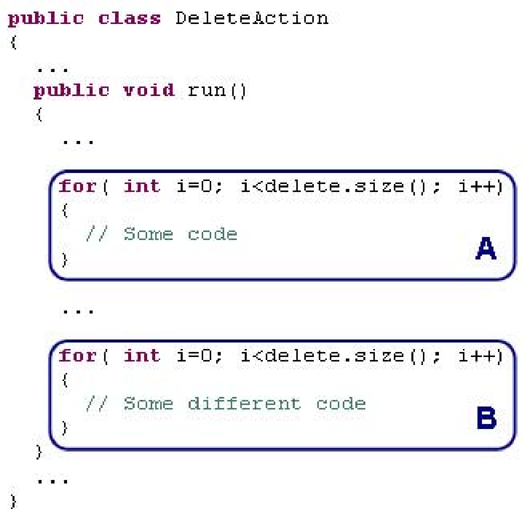
```
...actions/DeleteAction.java,DeleteAction,5
run(int),5
for,delete.size(),4
if,delete.get(i) instanceof ElementNode,2
```

In other words, this CRD points to the block corresponding to the `if` statement with the "...instance of..." predicate, nested within the `for` statement with the "...delete.size..." termination predicate, within the scope of the `run` method, etc. The numbers represent the corroboration metric for each block.

The corroboration metrics are an important element of CRDs because, although CRDs effectively describe code blocks, the blocks thus described are not always unique. Figure 5 shows an example of such a conflict.

Let us assume that we are interested in representing block B using a CRD. If our CRD only includes the fact that we refer to the `for` block with condition "i<delete.size()" in method `run()`, etc., the reference is ambiguous since there exists two such blocks (A and B). We thus need to further distinguish each block within a nesting level.

[5] `Else` branches are currently associated with the last `if` branch.



```
public class DeleteAction
{
  ...
  public void run()
  {
    ...
    for( int i=0; i<delete.size(); i++)
    {
      // Some code
    }
    ...
    for( int i=0; i<delete.size(); i++)
    {
      // Some different code
    }
  }
  ...
}
```

Figure 5. Example of conflict in block

To do so, we use a simple heuristic derived from our initial inspection of code clones. Our observation was that *when two or more code blocks at the same nesting level have identical CRDs, there are usually non-trivial differences in the logic implemented by each block.* This is not surprising if we assume that trivial differences can easily be parameterized and properly factored. We thus leverage off this observation and, for each block, generate a number that reflects the overall structure of the block. This number is our *corroboration metric*.

For our initial investigation of CRDs, we simply add the *cyclomatic complexity* of a block (number of linearly-independent paths) with the *fan-out* of the block (number of method invocations). In the event of a conflict (ambiguity) between two blocks at the same nesting level, we use the corroboration metric to select the block with the metric value that is the closest to the one recorded in the CRD. Section 6.1 reports on the effectiveness of our current corroboration metric for identifying the correct block when there exists multiple potential targets.

When automatically generating CRDs from clone regions produced with clone detection tools, our system finds the most deeply nested code block that fully encloses the code region, and produces a CRD for the block with the information obtained by parsing the source code.

4.2. Clone Region Lookup Algorithm

Given a CRD and a code base (that is not necessarily the one on which the CRD was defined), we identify the corresponding clone region through a series of automatic searches. The searches rely on an abstract syntax tree (AST) representation of the code.[6]

Type and method identification. The first step is to identify the AST node for the type declaration enclosing the clone region. First we assume that the `<class>` is in the `<file>`. When this is not the case (e.g., renamed file), we search the entire code base for type declarations with a name matching `<class>`. If none are found and a `<method>` is specified in the CRD, then we retrieve all method declarations in the system with a signature matching `<signature>`. The result of this search is a list of potential targets (type declarations for CRDs without a `<method>` specification, method declarations otherwise).

When more than one potential target is identified, a *conflict resolution* algorithm is applied. This algorithm computes the difference in the *corroboration metric* between each of the potential targets and the value stored in the CRD, and returns the target with the minimum absolute difference.

Block identification. Once an AST node corresponding to `<method>` is obtained, we recursively traverse its subtrees to look for the leaf block. Blocks are selected through a string comparison of their `<anchor>` condition as specified in the CRD (e.g., termination condition for `for` blocks). Conflicts at this level are also resolved using the *conflict resolution* algorithm.

Limitations. In our definition of CRDs, we made a number of design decisions to simplify the approach at the cost of decreased robustness. First, our reliance on nesting levels implies that changes that simply remove a nesting level while otherwise preserving a clone relation will invalidate a CRD. Second, associating `else` branches with the closest `if` prevents us from discriminating between the two types of blocks. Finally, storing anchors as strings implies that even small changes to the code in an anchor will invalidate the CRD. Our initial assumptions were that the cases impacted by such decisions would be rare enough to have a minimal impact on the overall usability of the technique. Section 6.1 details the current accuracy of the clone lookup algorithm with the above simplifications. In our future work, we plan to study the cost/benefit tradeoffs associated with more sophisticated CRDs.

5. Simultaneous Editing

Our clone tracking system provides support for simultaneously modifying groups of two clone regions. The simultaneous editing feature of our system relies on the clone region lookup algorithm described in the previous section, but requires additional computation to map an individual line *within* a clone region to the corresponding line in another clone region.

[6] The functionality to parse Java files and produce ASTs is provided with the standard Eclipse distribution.

Our line mapping technique is based on the *Levenshtein distance (LD)* [14]—a measure of the similarity between two strings based on the number of deletions, insertions, or substitutions required to transform one string into the other. The LD has been used in identifying similar patterns between strings for applications such as spell checking [2] and web page similarity analysis [3]. We felt the LD might be adequate for identifying regions for simultaneous editing because the copy, paste, and modification process through which code clones are formed is analogous to transforming one string into another through deletions, insertions, and substitutions.

Our *line mapping* algorithm identifies the line L_s in a clone region CR_1 that corresponds to a line L_t in a clone region CR_2. This way, when L_t is about to be modified, the changes can be echoed to L_s if they are applied to an exact match. We defined the similarity α between L_s and L_t as:

$$\alpha = 1 - \frac{LD(L_t, L_s)}{max(|L_t|, |L_s|)}$$

$LD(L_t, L_s)$ is the *Levenshtein distance* between L_s and L_t, and $max(|L_t|, |L_s|)$ is the length of the longest of the two strings. The value of α is in the interval $(0, 1]$.

The *line mapping* algorithm looks for all the lines of code in CR_1 with a similarity α of at least a threshold sim_{th} when compared to L_t. When more than one target line is possible, the algorithm uses lines L_{t-1} and L_{t+1} to resolve the conflict. It returns the line in the list of targets whose L_{s-1} and L_{s+1} have the minimum LD when compared to L_{t-1} and L_{t+1}, respectively.

The modifications in CR_2 are not echoed to CR_1 if a corresponding line is not found. Our simultaneous editing algorithm determines the appropriate character column and applies the modifications if the corresponding line is found. These modifications are made under the supervision of the developer, with the option to link/unlink editors and undo the modifications if need be.

Currently, our simultaneous editing feature is only supported for clone groups of two regions, where the regions are located in different files. Because clones are rarely exact, developer supervision is always required. For this reason, in the context of the Eclipse platform, it is a difficult challenge to support simultaneous editing of large clone groups or of clone regions in the same file. The development of a custom code editor might help address this limitation. In addition, one inherent weakness of LD for matching lines of code within clone regions is that it does not take semantic information into account. For example, reordered method parameters in one region will result in a low similarity despite the maintained semantic association. In the future we plan to explore other pattern matching techniques, including the name-Similarity technique proposed by Xing and Stroulia [24].

6. Quantitative Evaluation

The techniques we use to represent clone regions and support simultaneous editing rely on a number of heuristics. We report on our empirical assessment of the precision of CRDs in representing clone regions and on the precision of our line mapping algorithm.

6.1. Precision of CRDs

The basic tradeoff realized by CRDs is one of increased abstraction and robustness in the description of clone regions at the cost of decreased flexibility and precision in the representation of the boundaries of the region. Specifically, although clones regions can be technically arbitrary, in our system they must align with certain types of code blocks. This difference in representation can introduce discrepancies between the *actual* clone regions (as identified by clone detection tools) and the *documented* clone regions (as represented through CRDs).

We evaluated the accuracy of CRDs by conducting an empirical study using the clone regions identified by SimScan. In this study, we identified a number of clone regions, generated CRDs from them automatically, mapped the CRDs back to the source code, and analyzed the overlap between the initial regions and the regions documented with the CRDs.

To find clones for this study, we selected five Java-based open-source subject systems (JBossAOP, jEdit, FreeMind, Ant,[7] and JCommander.[8] See Table 1). These systems all have a recorded change history, exhibit a non-trivial number of clone groups, and were developed by different developers. For these reasons, we consider that, taken as a whole, these systems represent a reasonable diversity of Java programming styles.

Table 1. Subject Systems

System	Version	kLOC	# Dev.	# Groups
JBossAOP	4.0	35	9	279
jEdit	4.0-final	63	<130	251
FreeMind	0.8.0	14	9	84
Ant	1.6.5	86	20	403
JCommander	0.6.4	35	9	167

We ran the SimScan clone detection tool on each of our subject system (with the settings: Volume="medium", Similarity= "fairly similar", Speed/Quality = "fast"). This phase of the study resulted in a total of 1184 clone groups consisting of between 2 and 9 clone regions (inclusively), for a total of 3275 clone regions. For each clone region, we:

[7] ant.apache.org

[8] jcommander.sourceforge.net

1. Recorded the line range of the region;
2. Used CloneTracker to generate a CRD for the region;
3. Used CloneTracker to find the code represented by the CRD, recording whether this required resolving a conflict (at any nesting level);
4. Recorded the line range for the block represented by the CRD.

Table 2 summarizes our results for each system analyzed (the last column aggregates the results over all systems). The second row (# of CR) presents the number of clone regions identified for the system. The third row (# of Overlap) presents the number of clone regions for which the region mapped from the CRD overlapped with the original region (i.e., shared at least one line). Non-overlapping regions results from unresolved conflicts or from the limitations of CRDs as described in Section 4.2. Nevertheless, even with our initial heuristics, a large majority of clone regions (96%) were correctly tracked by our CRDs. This result is also consistent between systems, with a difference span of 5%.

The following three rows report on how closely overlapping clone regions matched. The fourth row (Avg. length of CR) presents the average length of a clone region (in lines of source code). The overall value of 24.6 shows that most clones regions are non-trivial sections of code. The fifth row (Avg. ML per CR) presents the average number of *missed lines* (lines in the original clone region but not mapped by the CRD). These lines are typically caused by artifacts of the clone detection technique. For example, SimScan does not systematically include or skip method signatures and/or Javadoc comments from the clone region, whereas CRDs systematically skip the method header, and start the clone region at the first curly brace. When the method signature is on a separate line, it will become a missed line in our experiment. The sixth row (Avg. EL per CR) presents the average number of *extra lines* (lines not in the original clone region but mapped by our CRD). These lines result from the fact that clone regions have to be expanded to the closest enclosing block to be described by a CRD. Overall, we see that not only do clone regions generally overlap with the regions represented by CRDs, but they also align acceptably well, as the average number of missing or extra lines is below 4.

The last two rows report on the extent with which conflicts must be resolved within a nesting level in order to find the correct block, and the extent with which our corroboration metric helps in this process. The seventh line (# of conflicts) presents the number of conflicts detected between CRD blocks at the same nesting level, and the last row (# of Resolved Conflicts) presents the number of conflicts that were correctly resolved. We see that these numbers exhibit non-negligible variations between systems, which may be caused by variations in programming style. We expect that experimentation with alternative corroboration metrics should help us lower the number of unresolved conflicts. Overall, however, these initial results show that even a simple metric can help disambiguate a majority (on average 81%) of blocks with an otherwise equivalent representation.

6.2. Simultaneous Editing

The simultaneous editing algorithm we presented in Section 5 is parameterized with a threshold sim_{th} that determines how closely two lines must match to be considered equivalent lines. We determined an appropriate level for this threshold by empirically assessing the success of simultaneous edits for different threshold values.

We selected benchmarks for our evaluation of simultaneous editing by randomly choosing 15 different clone groups of size two from each subject system (see Table 1). For each of the 15 × 5 = 75 clone groups, we linked the editors and made three modifications within common sub-regions of clone groups. The three modifications consisted of an insertion, a deletion, and a replacement (selecting a region and pasting something into it). For each region, the modification were chosen to reflect a likely software modification (e.g., removing an argument from a method call). For each modification, we recorded whether the result of the simultaneous modification was correct (based on a simple manual inspection of the corresponding regions). We repeated this experiment for seven different values of sim_{th} between 0.35 and 0.95 (values below 0.35 had no further effect). For a given region, we used identical sets of modifications for all threshold values.

Figure 6 illustrates the impact of the threshold on the results. With a high similarity threshold (0.95), simultaneous editing was not very successful (30%). This phenomenon is not surprising since the very stringent similarity requirement, combined with the fact that very few clones are exact, means that equivalent lines will not be found. Lowering the threshold increases the success level since a greater number of lines are matched. This trends flattens with a threshold of 0.55 (and a success level of 80%). This flattening is a strength of our algorithm and is the consequence of our decision to look at lines above and below target lines in cases where multiple targets lines meet the threshold. In other words, our algorithm is robust to a decrease in performance caused by collisions in potential target lines. We did not evaluate the success level for lower threshold values since the results would not be useful, and since the experiment was very labor-intensive. However, we expect that the success level will eventually degrade as the number of collisions reaches extreme levels.

Overall, for our subject systems, we could parameterize the algorithm to be successful up to 4 out of 5 times in supporting simultaneous editing. Based on this experiment, we chose 0.55 as a working threshold.

Table 2. Precision and Accuracy of CRD for Describing Clone Regions

	jEdit		JBossAOP		FreeMind		Ant		JCommander		ALL	
# of CR	536		1142		195		966		436		3275	
# of Overlap	505	(94%)	1066	(93%)	188	(96%)	951	(98%)	426	(98%)	3136	(96%)
Avg. length of CR	27.1		23.3		18.0		25.0		27.0		24.6	
Avg. ML per CR	2.1		2.0		1.3		1.4		1.8		1.8	
Avg. EL per CR	1.5		4.0		2.9		3.5		5.0		3.5	
# of Conflicts	29		34		4		53		11		131	
# of Resolved Conflicts	28	(97%)	28	(82%)	4	(100%)	39	(74%)	7	(64%)	106	(81%)

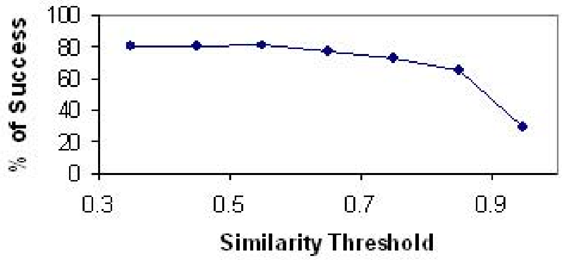


Figure 6. Impact of Threshold on Success

7. Case Study

To provide initial evidence that CRDs succeed in describing clones in evolving software, we used CloneTracker to document clones in base versions of both jEdit and Ant, and attempted to track these clones across subsequent versions of the subject systems using the documented clone models. We selected these systems because of their long version history, in which we could find clones that were not aligned with method boundaries, and which survived through multiple versions of the systems.

We selected five clone groups for this study. We describe these groups in Table 3 so that others can independently assess our observations. The table identifies both clone regions of the five clone groups using a summary of the information in their CRD. The clone groups were initially detected in release 1.5.4 for Ant and in version 4.0-final for jEdit.

All clone groups studied evolved as part of changes performed to different versions of the system. The evolution of Group 1 is described in detail in Section 2. For Group 2, we found changes made to non-identical sub-regions of the clones at releases 4-2-pre1 and 4-2-final. We found that Group 3 was changed inconsistently in Ant1.6.0 (where a variable access was changed to method call to make the clone regions *more* similar). Group 4 was changed inconsistently in Ant1.6.0. Finally, Group 5 was also changed inconsistently in Ant1.6.3 (lines were added in the body of one sub-region). For all the clone regions studied, the corresponding line ranges changed across every version of the system.

After having studied the evolution of each group, we used CloneTracker on the base versions to automatically generate CRDs for all the clone regions. Then, using the generated clone models, we attempted to locate these clone regions for the remaining subsequent versions of the system (releases 1.6.0 to 1.6.5 for Ant, 6 in total, and releases 1.4-pre1 to 4.3-pre6 for jEdit, 27 in total).

We successfully located Group 1 in all versions of jEdit until version 4-2-pre10, where the group disappeared from the system (the class still exists but the clone regions were removed).

Group 2 was successfully tracked across the first 26 versions of jEdit. We successfully located only region A in the final version (4.3-pre6). The class of clone region B and the corresponding block could not be found in the system.

Clone groups 3, 4 and 5 were successfully tracked across all six versions of Ant.

In brief, using the CRDs and our clone tracking system allowed us to track the clones throughout different versions of the system. In each case where a clone was modified, our system automatically would have warned the developer about the clone, supporting developers in their efforts to find, understand, and modify the cloned regions.

8. Related Work

The ideas and techniques investigated in our work on clone tracking intersect with a broad spectrum of research projects on clone detection and analysis, linked editing, and source code representations.

Clone Detection. A vast body of work exists on techniques to efficiently detect and analyze clones in source code. In the classic form, a clone detection tool takes as input the source code text of a software system, pre-processes the text (e.g., to break lines into tokens and to remove nonessential differences), and then performs a similarity analysis on the transformed input. In their presentation of the CCFinder tool, Kamiya et al. provide a clear and thorough description of this type of clone detection technology [10]. Other clone detection approaches have also been proposed that use inputs such as the abstract syntax tree of a program,

Table 3. Clone Groups Studied

Group	Class	Method	Block

1 (jEdit)	A) bsh.Reflect
	B) bsh.Reflect
		findExtendedConstructor(...)
		findExtendedMethod(...)
			for(i<constructors.length)
			for(i<methods.length)

2 (jEdit)	A) org.gjt.sp.jedit.jEdit
	B) org.gjt.sp.jedit.Abbrevs
		initUserProperties()
		load()
			if(settingsDir!=null)
			if(settings!=null)

3 (Ant)	A) ...ant.taskdefs.Expand
	B) ...ant.taskdefs.optional.XMLValidateTask
		execute(...)
		execute(...)
			if(fileset.size()>0)
			for(i<fileset.size())

4 (Ant)	A) ...ant.taskdefs.ManifestTask
	B) ...ant.taskdefs.Jar
		execute(...)
		getManifest(..)
			try(ManifestException,IOException)
			try(UnsupportedEncodingException,IOException)

5 (Ant)	A) ...ant.taskdefs.optional.net.FTP B) ...ant.DirectoryScanner	checkIncludePatterns() checkIncludePatterns()	for(icounter<includes.length) for(icounter<includes.length)

the topology of a program dependency graph, or code metrics. We refer the interested reader to one of a number of annotated bibliographies of the code clone literature [22].

Clone Genealogy Anaysis. Kim et al.'s empirical study of code clone genealogies [13] provided an important part of the motivation for this research. For their study, Kim et al. built a clone genealogy extraction tool. This tool integrates the CCFinder clone detector and reports, for a sequence of program versions, how each clone region has evolved (changed, disappeared, etc.) with respect to the other clone regions in the group. The mapping of clone regions between versions is computed from an analysis of textual similarity using a module that extends the *diff* utility program. Using their clone genealogy extraction tool, Kim et al. tracked the evolution of code clones in two Java programs. Their study led to the conclusions quoted and discussed in Section 2.

Linked Editing. A number of previous projects focused on the investigation of linked editing techniques from the perspective of the user interface. Miller and Myers [17] proposed to use *simultaneous editing* to simplify repetitive text editing tasks. Their technique is implemented in LAPIS, a text editor with a knowledge of Java, C++, and HTML syntax. With LAPIS, a developer has to manually enter the regions to link, either through selection or by specifying a text pattern. Regions in LAPIS are expressed in terms of character regions. A similar technique, called *linked editing*, has been proposed by Toomin et al. [23]. The technique is implemented in a tool called Codelink. Codelink allows a user to manually select clone regions and to link them. Codelink's algorithm for tracking exact sub-regions within a clone region is different than ours, and uses a tokenized version of the input text. Codelink also allows users to save a description of clone regions as meta-data. The exact description of the meta-data is not provided, but it is not resilient to file modifications as the authors express the wish to "make [their] link meta-data resilient to file modifications made by third-party tools" [23].

Some of the user interface aspects of CloneTracker's simultaneous editing feature were inspired by this previous work on linked editing. However, in contrast to LAPIS and Codelink, CloneTracker's main purpose is to support the long-term tracking of clones. For this reason, it uses an abstract model to represent clones, whereas the two systems described above manage clone regions in terms of regions of text. Our linked editing feature also benefits from the high level of automation provided by our system, which saves users the task of manually specifying clone regions.

Source Code Representation. A number of approaches have been proposed that allow developers to specify a subset of the source code of a program using abstract models that are resilient to a certain amount of changes in the source code (e.g., Concern Graphs [20], Aspect Browser [6], Intentional Views [16], CME [8]). Typically, such frameworks allow developers to specify code of interest in terms of properties of the program (e.g., all the callers of method *m*). Although they could be used to track clones that align with the boundaries of coarse-grained elements (e.g., methods), they do not provide the flexibility to tag specific blocks in the source code. In the context of aspect-oriented programming (AOP), attempts have been made to identify low-level constructs in programs, such as `for` loops [7]. Because the underlying goal of AOP is to impact *crosscutting* code, such techniques focus on constructs that can describe *classes* of constructs (e.g., all loops with a specific predicate), as opposed to individual regions.

9. Conclusion

The presence of code clones in software systems creates additional work for developers and can increase the risk of introducing regression faults during software maintenance. Unfortunately, it is not always possible or practical to eliminate certain clone groups from a software system.

We propose to mitigate the negative effects of code clones by tracking clone regions of interest as a system evolves. We implemented a system, called CloneTracker, that can automatically generate abstract representations for clone regions from the output of a clone detection tool, detect any modification to tracked clone regions, and support the simultaneous editing of clone regions. Our system relies on the concept of clone region descriptors (CRDs), which identify clone regions at the granularity of code blocks using heuristics based on the structural properties, lexical layout, and similarities of the clone region. Our initial investigation of this idea showed that, even using relatively simple heuristics, we could track the vast majority of the 3275 clone regions we investigated. Our experience also pointed to a number of technical aspects that could be improved upon to further increase the accuracy of the approach. However, we expect that further adjustments are bound to provide mostly incremental improvements, as the evidence we have collected so far indicates that CRDs are a lightweight, practical, and robust representation for tracking code clones in evolving software.

Acknowledgments

The authors are grateful to Miryung Kim, Barthélémy Dagenais, and the anonymous reviewers for their valuable comments on this paper. This work was supported by NSERC and by IBM.

References

[1] M. Balazinska, E. Merlo, M. Dagenais, B. Lagüe, and K. Kontogiannis. Partial redesign of Java software systems based on clone analysis. In *Proceedings of the 6th Working Conference on Reverse Engineering*, pages 326–336, 1999.

[2] F. J. Damerau. A technique for computer detection and correction of spelling errors. *Communications of the ACM*, 7(3):171–176, 1964.

[3] A. De Lucia, R. Francese, G. Scanniello, and G. Tortora. Identifying cloned navigational patterns in web applications. *Journal of Web Engineering*, 5(2):150–174, 2006.

[4] M. Fowler. *Refactoring—Improving the Design of Existing Code*. Addison-Wesley, 2000.

[5] R. Geiger, B. Fluri, H. C. Gall, and M. Pinzger. Relation of code clones and change couplings. In *Proceedings of the 9th International Conference of Funtamental Approaches to Software Engineering*, volume 3922 of *Lecture Notes in Computer Science*, pages 411–425. Springer, 2006.

[6] W. G. Griswold, J. J. Yuan, and Y. Kato. Exploiting the map metaphor in a tool for software evolution. In *Proceedings of the 23rd International Conference on Software Engineering*, pages 265–274, 2001.

[7] B. Harbulot and J. R. Gurd. A join point for loops in AspectJ. In *Proceedings of the International Conference on Aspect-Oriented Software Development*, pages 63–74, 2006.

[8] W. Harrison, H. Ossher, S. Sutton Jr., and P. Tarr. Concern modeling in the concern manipulation environment. Technical Report RC23344, IBM Research, 2004.

[9] Y. Higo, T. Kamiya, S. Kusumoto, and K. Inoue. Refactoring support based on code clone analysis. In *Proceedings of the 5th International Conference on Product Focused Software Process Improvement*, volume 3009 of *Lecture Notes in Computer Science*, pages 220–233. Springer, 2004.

[10] T. Kamiya, S. Kusumoto, and K. Inoue. CCFinder: A multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, 28(7):654–670, 2002.

[11] M. Kim, L. Bergman, T. Lau, and D. Notkin. An ethnographic study of copy and paste programming practices in OOPL. In *Proceedings of the International Symposium on Empirical Software Engineering*, pages 83–92, 2004.

[12] M. Kim and D. Notkin. Program element matching for multi-version program analyses. In *Proceedings of the 3rd International Workshop on Mining Software Repositories*, 2006.

[13] M. Kim, V. Sazawal, D. Notkin, and G. C. Murphy. An empirical study of code clone genealogies. In *Proceedings of the Joint European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 187–196, 2005.

[14] J. B. Kruskal. An overview of sequence comparison. In David Sankoff and Joseph B. Kruskal, editors, Time warps, string edits, and macromolecules: the theory and practice of sequence comparison, 1983.

[15] B. M. Lange and T. G. Moher. Some strategies of reuse in an object-oriented programming environment. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 69–73, 1989.

[16] K. Mens, T. Mens, and M. Wermelinger. Maintaining software through intentional source-code views. In *Proceedings of the 14th International Conference on Software Engineering and Knowledge Engineering*, pages 289–296, 2002.

[17] R. C. Miller and B. A. Myers. Interactive simultaneous editing of multiple text regions. In *Proceedings of the USENIX 2001 Annual Technical Conference*, pages 161–174, 2001.

[18] Object Technology International, Inc. Eclipse platform technical overview. White Paper, 2001.

[19] D. F. Redmiles. Reducing the variability of programmers' performance through explained examples. In *Proceedings of the Conference on Human Factors in Computing Systems*, pages 67–73, 1993.

[20] M. P. Robillard and G. C. Murphy. Representing concerns in source code. *ACM Transactions on Software Engineering and Methodology*, 2006. Accepted for publication.

[21] M. P. Robillard and F. Weigand-Warr. ConcernMapper: simple view-based separation of scattered concerns. In *Proceedings of the 2005 OOPSLA Workshop on Eclipse technology eXchange*, pages 65–69, 2005.

[22] R. Tairas. Bibliography of code detection literature. http://www.cis.uab.edu/tairasr/clones/literature/.

[23] M. Toomim, A. Begel, and S. L. Graham. Managing duplicated code with linked editing. In *Proceedings of the IEEE Symposium on Visual Languages and Human Centric Computing*, pages 173–180, 2004.

[24] Z. Xing and E. Stroulia. UMLDiff: an algorithm for object-oriented design differencing. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, pages 54–65, 2005.