Clone Region Descriptors: Representing and Tracking Duplication in Source Code

EKWA DUALA-EKOKO and MARTIN P. ROBILLARD
McGill University

3

Source code duplication, commonly known as *code cloning*, is considered an obstacle to software maintenance because changes to a cloned region often require consistent changes to other regions of the source code. Research has provided evidence that the elimination of clones may not always be practical, feasible, or cost-effective. We present a clone management approach that describes clone regions in a robust way that is independent from the exact text of clone regions or their location in a file, and that provides support for tracking clones in evolving software. Our technique relies on the concept of abstract *clone region descriptors* (CRDs), which describe clone regions using a combination of their syntactic, structural, and lexical information. We present our definition of CRDs, and describe a clone tracking system capable of producing CRDs from the output of different clone detection tools, notifying developers of modifications to clone regions, and supporting updates to the documented clone relationships. We evaluated the performance and usefulness of our approach across three clone detection tools and five subject systems, and the results indicate that CRDs are a practical and robust representation for tracking code clones in evolving software.

Categories and Subject Descriptors: D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement

General Terms: Design, Experimentation

Additional Key Words and Phrases: Source code duplication, code clones, clone detection, refactoring, clone management

ACM Reference Format:

Duala-Ekoko, E. and Robillard, M. P. 2010. Clone region descriptors: Representing and tracking duplication in source code. ACM Trans. Softw. Eng. Methodol. 20, 1, Article 3 (June 2010), 31 pages.
DOI = 10.1145/1767751.1767754 http://doi.acm.org/10.1145/1767751.1767754

This work was supported by NSERC.
This article is a revised and extended version of an article presented at ICSE 2007 in Minneapolis, MN.
Authors' address: School of Computer Science, McGill University, 3480 University Street, McConnell Engineering Building no. 318, Montreal, P.Q., Canada, H3A 2A7; email: {ekwa,martin}@mcgill.ca.
Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.
© 2010 ACM 0163-5948/2010/06-ART3 $10.00
DOI 10.1145/1767751.1767754 http://doi.acm.org/10.1145/1767751.1767754

1. INTRODUCTION

Software systems often contain numerous groups of source code regions that match each other with varying degrees of exactness. Such source code regions, commonly known as *code clones*, have been reported to account for up to 30% of the source code of large scale software systems [Baker 1995]. 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 [Lange and Moher 1989; Redmiles 1993], and the use of idioms for framework extensions.

Code clones present two major obstacles to software maintenance. First, code cloning may duplicate faulty code regions, resulting in the *recurring-bug* problem, where resolved bugs seems to reappear as cloned siblings get executed [Aversano et al. 2007]. The recurring-bug problem typically increases the software maintenance effort because bugs have to be resolved multiple times. Second, the presence of code clones in a system means that code that realizes identical or similar logic is not colocated. This duplication of implementation logic often leads to a necessity to modify multiple sections of code consistently [Geiger et al. 2006]. Oversights in consistently modifying clone regions may lead to regression faults. For these reasons, much effort has been spent on the detection and removal of code clones from software systems [Tairas 2008; Koschke 2008]. Technology to scan the source code of a system and identify clones of varying similarity is now readily available (see, e.g., Kamiya et al. [2002]; Jiang et al. [2007a]; Basit and Jarzabek [2007]).

What should we do with code clones, once identified? Some researchers have argued that, to achieve a good design, the code should "say everything once and only once," and thus have advocated the removal of clones through source code refactoring [Fowler 2000, p. 56]. However, 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, Kim et al. found that "skilled programmers often created and managed code clones with clear intent" [Kim et al. 2004, p. 187]. A later study of code clone genealogies also provided evidence of clones that are difficult or impossible to refactor using standard techniques, and of clones that evolve into distinct code [Kim et al. 2005]. A different study by Kapser and Godfrey [2006] reported several situations where code duplication appears to be a beneficial design option, such as the use of duplication in exploratory development or experimental changes to core subsystems.

The difficulty to refactor certain code clones, and the potential of cloning as a viable design option, have strengthened the call for clone management tools. A handful of tools to manage clones formed through copy-and-paste operations and to support consistent modification to clone regions have been developed: for instance, CReN [Jablonski and Hou 2007], LAPIS [Miller and Myers 2001], and CodeLink [Toomim et al. 2004]. However, tools such as CReN

and LAPIS represent clone regions using line ranges or character offsets, and rely exclusively on the Integrated Development Environment (IDE) to update the location of the clone regions. Consequently, modifications that alter the line ranges of a clone region, or refactorings, such as pulling up a method to its superclass, may invalidate the clone relationships if performed outside the host environment. Furthermore, techniques that capture copy-and-paste operations for clone management are not beneficial to existing source code. To be effective, clone management techniques require a representation that is robust to evolutionary changes and applicable to both existing and future source code.

To this end, we introduce *clone region descriptors* (CRDs), a heuristic-based representation for identifying cloned source code locations within methods using a combination of syntactic, structural and lexical information, and software metrics about a clone region. CRDs go beyond simple lines of code-based clone descriptions, and support the tracking of clone regions in different versions of a software system. We have developed a clone tracking system called *CloneTracker* to support CRDs. 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 of interest to a developer. Using CRDs, CloneTracker can automatically track clones of interest as the code evolves and notify developers of modifications to clone regions. This way, software developers can specify clone groups they wish to track once and carry on with all of their future modification tasks with the knowledge that modifications to specified 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 representation to manage clone groups of interest without having to reidentify or reclassify them in subsequent versions of a system.

We evaluated the effectiveness and usefulness of CRDs using three clone detection tools and five target open-source systems. Although CRDs are heuristic-based, they were able to represent a large majority of the 16,350 clone regions we analyzed. An additional study also indicated that CRDs are robust and an effective representation for tracking a large majority of the clones across different versions of a system. The contributions of this article include the following:

— a technique for representing and tracking clones using a combination of the syntactic, structural, and software metrics of a clone region, and a tool implementing this technique;
— a comparison of the precision of CRDs with respect to line-based representation of clone regions for three clone detection tools and five subject systems, with insights on how CRDs perform for different clone detection techniques;
— an evaluation of the robustness of CRDs to evolutionary changes for several versions of the subject systems, with clones successfully tracked across 40 versions of one of the systems.

The rest of the article is organized as follows. We present a real example of clone evolution and explain the difficulties associated with tracking clones in

Section 2. In Sections 3–4, we describe our clone tracking system, the details of our clone region representation, and our technique for updating the clone model as the corresponding software system evolves. We report on the evaluation of our approach in Section 5, discuss related work in Section 6, and conclude in Section 7.

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 (e.g., Volume=medium, Similarity=fairly similar, Quality/Speed=fast)[3] and runs the tool. After approximately 32 min (WindowsXP, Pentium4-3 GHz, 512 MB), 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 code region in method `findExtendedMethod` (see Figure 1).

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 ... [Kim et al. 2005, 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 ... [Kim et al. 2005, p. 188].

Both regions are in methods that have a different return type (the method shown in Figure 1(a) has return type `Constructor`, whereas the method shown in Figure 1(b) has return type `Method`), which makes them locally unfactorable using standard refactorings [Fowler 2000]. In addition, our clone group existed for 15 versions of jEdit (from version 4.0-final to version 4.2-pre07 inclusive), making it long-lived.

> ... we found that many clones were volatile ... [Kim et al. 2005, p. 188].

[1] `www.jedit.org`.
[2] `blue-edge.bg/simscan`.
[3] Volume, for the size of the region; Similarity, for the minimum similarity of the reported clone regions; Quality/Speed, a tradeoff between the quality of the results and the speed of the search. Higher speed is achieved by skipping statistically less significant areas of the search space.

```
static Constructor findExtendedConstructor(
Object[] args, Constructor[] constructors){
  ...
  for(int i=0; i¡constructors.length; i++)
  {
    Constructor curr = constructors[i];

    Class[] pms = curr.getParameterTypes();
    If(pms.length != args.length)
       continue;

    try{
         for(int j=0; j¡pms.length; j++)
         tmpArgs[j] = NameSpace.
         getAssignableForm(args[j],pms[j]);

        //if you get here, all the
        //arguments were assignable
        System.arraycopy(
        tmpArgs, 0, args, 0, args.length);
        return curr;
    } catch(EvalError e){
        //do nothing
    }
  }
     ...
}
```

(a)

```
static Method findExtendedMethod(
String name, Object[] args, Method[] ms){
  ...
  for(int i=0; i¡ms.length; i++)
  {
    Method curr = ms[i];
    if(name.equals(curr.getName())){
    Class[] pms = curr.getParameterTypes();
    If(pms.length != args.length)
       continue;

    try{
         for(int j=0; j¡pms.length; j++)
         tmpArgs[j] = NameSpace.
         getAssignableForm(args[j],pms[j]);

        //if you get here, all the
        //arguments were assignable
        System.arraycopy(
        tmpArgs, 0, args, 0, args.length);
        return curr;
    } catch(EvalError e){
        //do nothing
    }
  }}
     ...
}
```

(b)

Fig. 1. Sample locally unfactorable clone group with two clone regions, both located in the `bsh.Reflect.java` class of jEdit release 4.0-final.

Our clone group eventually *disappeared*[4] in release 4.2-pre10. The disappearance in release 4.2-pre10 was caused by a major modification in one of the clone regions independently of the other, thus eliminating their clone relationship. As claimed by Kim et al. [2005], 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 our example, refactoring may not prove cost-effective because the clone group eventually disappear, and not feasible because the return types of the methods are different. 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 tool 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. The information about the locations of clone regions (i.e., the files and the positions of the clone regions within these files) is therefore not always reusable in future tasks because the line ranges and character-based representations are fragile, even to basic code

[4]A clone group identified in version v_1 of a system is said to have disappeared in version v_n, $n > 1$ if it has no clone regions in common with any of the clones groups identified in v_n.

modifications. Clone detection techniques must therefore be reapplied to recover the locations of clone regions.

—A developer must perform clone detection on two versions of the code, and successfully map identical clone groups in both versions in order to identify inconsistently modified clones. This alternative is impractical for interactive software development because, as illustrated earlier in this section, a 63kLOC project required 32 min just to identify the clone groups using SimScan.

—Although clone detection speed could be improved by using other tools, the cost of manual *clone classification*—determining whether knowledge of a reported clone group is actually important to developers—cannot be avoided. Even a conservative clone classification effort of 30 s/clone group would require about 2 h to go through the list of 251 groups identified in jEdit.

We have developed CloneTracker to address these challenges. In the next section, we present the details of our representation, heuristics, and algorithms enabling this technology. We postpone the presentation of CloneTracker until Section 4.

3. REPRESENTING CLONE REGIONS

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.[5] 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 [Kim and Notkin 2006]. However, for our purpose (rapid, interactive tracking of clone regions), we needed to investigate alternative representations.

To gather insights about potential ways to describe clone regions, we manually inspected several clone regions from four different projects involving tens of different developers (ConcernMapper [Robillard and Weigand-Warr 2005], jEdit, JBossAOP,[6] and FreeMind[7]). These clones were identified by SimScan, an AST-based clone detector. While looking at these clone regions, we tried to determine what unique characteristics of the clone 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, and changes from method refactorings). 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). For

[5]To *align with method boundaries* implies the entire body of a method is a clone region, whereas to occur *within a method* implies a clone region is located within a code block such as a loop inside a method.

[6]`labs.jboss.com/portal/jbossaop/`.

[7]`freemind.sourceforge.net`.

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

Fig. 2. Definition of CRDs in extended BNF.

instance, 60% of the clone regions investigated were aligned with method boundaries, and 40% of the clone regions were within methods. Of the clone regions within methods, 23% were within loops, 57% were within conditional branches, and 20% were within the try/catch/finally blocks.

(2) Some structural elements (e.g., loop-termination predicates, branching predicates, exception lists) tend to be unique at a given level of nesting. For instance, less than 5% of the clone regions investigated had nonunique structural elements, and of the nonunique structural elements, 46% involved loop termination predicates and 54% involved branching predicates.

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, our algorithm for locating clone regions based on the model, and the algorithm employed for updating the clone model.

3.1 Clone Region Descriptors (CRDs)

A *Clone Region Descriptor* (CRD) is an abstract description of the location of a clone region in a code base. The idea is to provide an approximate location that is independent from specifications based on lines of source code, annotations, or other similarly fragile markers. Figure 2 shows our 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>—used for *conflict resolution* in cases where a CRD refers to more than one code block—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>), 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 supported by our clone model are listed as part of the <btype> nonterminal symbol in Figure 2.

```
public class DeleteManager
{
  ...
  public void delete(int n)
  {
    ...
    for(int i = n; i < delete.size(); i++)
    {
      If(delete.get(i) instanceof ElementNode)
      {
        //some code
      }                                      A
    }
    ...
  }
}
```

Fig. 3. Sample clone region within a `for` block.

Different schemes are possible for the `<anchor>`. Our empirical data indicates that the use of the textual representation of a distinctive statement associated with the block is enough to track a vast majority of the clone regions (see Section 5.1). For loops, we use the termination statement. For `if` statements, we use the branching predicate. For `switch` statements, we use the switch expression. For `try` blocks, we use the list of exception types caught in the `catch` clauses associated with the block. For `catch` blocks, we use the type of the exception caught. For `synchronized` blocks, we use the synchronization expression. For instance, the CRD for code block A in Figure 3 is

```
packagename.DeleteManager.java,DeleteManager,5
delete(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 "...instanceof..." predicate, nested within the `for` statement with the "...delete.size()..." termination predicate, within the scope of the `delete(int)` method, within the class `DeleteManager`, and within the file `packagename.DeleteManager.java`. The numbers represent the corroboration metric for each block.

Clone regions within a `finally` or an `else` code block are represented differently because of the challenges they present. The `finally` and `else` code blocks do not have an obvious `<anchor>` as described in the context of CRDs, that is, a distinguishing identifier for a given code block. For instance, a loop has a termination condition, an `if`-block has a branching predicate, but an `else` block has no such identifiers. However, the `finally` and `else` code blocks are never standalone, but are always defined in relation to a parent code block. The `finally` block always has a `try` block as parent, and the `else` block has one or more `if/else` code blocks as parent. We take advantage of this parent-child relation in generating CRDs for the `finally` and `else` code blocks. For instance,

```
public class SQLClient
{
  ...
  public void executeQuery(String query)
  {
    ...
    try
    {
        //some code
    } catch (SQLException ex) {
        //some code
    }finally {
        //some code                    B
    }
  }
}
```

Fig. 4. Sample clone region within a `finally` block.

the CRD for the `finally` code block, B, in Figure 4 is

```
packagename.SQLClient.java,SQLClient,35
executeQuery(String)
finally,SQLException,12
```

That is, this CRD points to the `finally` code block which is attached to a `try` block that throws an `SQLException`, within the scope of the `executeQuery(String)` method, within the class `SQLClient`, and within the file `packagename.SQLClient.java`. As before, the numbers represent the corroboration metric for each block. The CRD of a `finally` block whose parent `try` block contains multiple exception handlers is the list of the types of exceptions handled in the corresponding `catch` blocks, in their declared order. Therefore, the anchor of a `finally` code block is simply the anchor of its parent `try` block, with `try` replaced by `finally` to differentiate them. The CRD of an `else` block with multiple `if-then` blocks is defined similarly—the anchors of its preceding `if-then` blocks serves as its anchor.

3.1.1 *Conflict Resolution Heuristics.* The corroboration metrics are an important element of CRDs because, although CRDs effectively describe code blocks, such a definition can sometimes refer to more than one block. Figure 5 shows an example of such a scenario.

Let us assume that we are interested in representing block Y using a CRD. If our CRD only includes the fact that we refer to the `for` block with condition "i<delete.size()" in method `delete()`, the reference is ambiguous since there exists two such blocks (X and Y). We describe such an ambiguity as a *conflict*. Conflicting code blocks require further information to distinguish them within a nesting level.

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 nontrivial 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.

```
public class DeleteManager
{
  ...
  public void delete()
  {
    ...
    for(int i = 0; i < delete.size(); i++)
    {
        //some code
    }                                          X
    ...
    for(int i = 0; i < delete.size(); i++)
    {
        //some different code
    }                                          Y
  }
}
```

Fig. 5. Example of two code blocks at the same level with identical anchors.

We thus take advantage of this observation and, for each block, generate a number that reflects the overall structure of the block. This number is our *corroboration metric*, and is used only when the lookup algorithm, presented in Section 3.2, encounters an ambiguity (conflict) between two or more code blocks.

Our *corroboration metric* is a combination of three code metrics of a clone region: its *cyclomatic complexity* (number of linearly-independent paths) [McCabe 1976], *fan-out* (number of method invocations), and *decision density* (the proportion of the cyclomatic complexity to the number of lines of code in the clone region) [Gill and Kemerer 1991]. Although rare, we encountered cases with multiple code blocks having identical anchors, cyclomatic complexities, and fan-outs at the same level of nesting in our initial experiments. The decision density of the code block is therefore necessary to normalize the value of the cyclomatic complexity of the code block, and hopefully disambiguate such cases of conflict. Using the sum of these three metrics as our corroboration metric was enough to resolve 93% of the 1257 conflicts encountered in our evaluation (see Section 5.1). In the event of a conflict between two or more blocks at the same nesting level, we used the corroboration metric to select the block with the metric value closest to the one recorded in the CRD. The conflict resolver requires assistance from the developer when more than one code block is identified as being closest to that recorded in the CRD. The developer is requested to select the code block matching the CRD from a list of potential targets.

3.1.2 *Generating CRDs.* The support for generating CRDs of clone regions from the output of clone detection tools was implemented in Eclipse. Each clone group, as represented by the clone detection tool, includes the names of the files that contain clones, and the location (i.e., the `start` and `end` line numbers) of each clone region within their respective files. To generate the CRDs of the clone

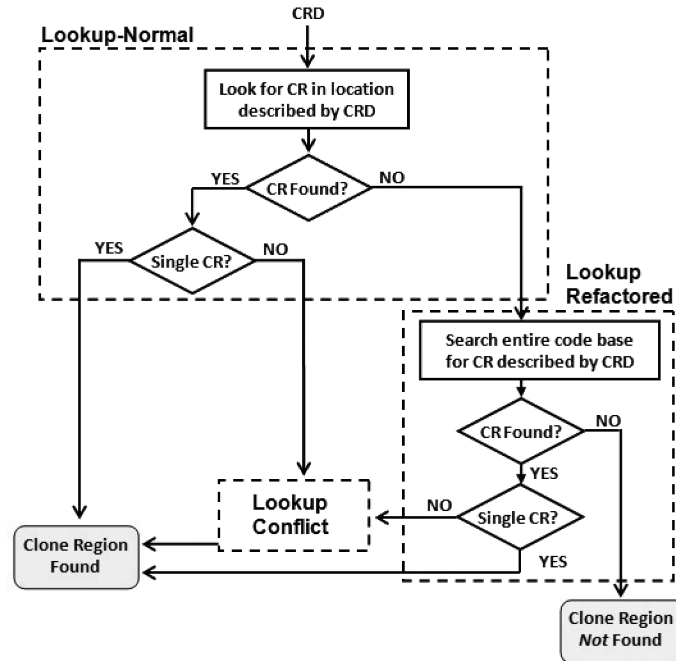

Fig. 6. An overview of the clone region lookup algorithm. Given the CRD of a clone region (CR), the lookup-normal search assumes the CR exists at the location described by the CRD. If this assumption is true, and there is a single code block matching the description of the CRD, then the CR has been found. If there are two or more code blocks matching the description of the CRD, then we call the lookup-conflict search to determine the most likely target. If the lookup-normal search did not find the CR described by the CRD, we assume the CR has been moved by a refactoring and call the lookup-refactored search. Lookup-refactored searches the entire code base, and proceeds similarly as the lookup-normal search if one or more code blocks are found matching the CRD, or terminates after all options are exhausted.

regions of a group, we obtain an abstract syntax tree (AST) representation of the source code of each file. The functionality to parse Java files and produce ASTs is provided with the standard Eclipse distribution, and is accessible through an API call. We implemented a Visitor class that traverses the AST looking for the node that fully encloses the `start` and `end` lines of a code region. The search is terminated once a node that represents the clone region is found. The components of a CRD, such as the anchors and corroboration metric, are all generated during the traversal. This entire process takes a fraction of a second, as will be described in Section 5.3.

3.2 Clone Region Lookup Algorithm

Given a CRD and a code base (which is not necessarily the one on which the CRD was defined), we identify the corresponding clone region through a series of automatic searches: *lookup-normal*, *lookup-conflict*, and *lookup-refactored* (see Figure 6). These searches rely on an AST representation of the source

code. We use the CRD of the clone region in Figure 3 to explain the features of the search algorithms.

3.2.1 *Lookup-Normal.* This is used in cases where a clone region exists at the location described by the CRD, and there is no need for conflict resolution. The lookup-normal search first retrieves the `<file>` (`packagename.DeleteManager.java`) in which the clone region is located, and then obtains the AST node of the enclosing `<class>` (`DeleteManager`). Next, lookup-normal searches for the `<method>` (`delete(int)`) from the AST of the enclosing class. Once an AST node corresponding to the method `delete(int)` is obtained, lookup-normal traverses its subtrees looking for the code `<block>` described by the CRD. Blocks are selected through a string comparison of their `<anchor>` condition as specified in the CRD (e.g., the termination condition for `for` blocks). In our example, lookup-normal would then search for the `for` block with termination condition "`i<delete.size()`" from the AST node of the enclosing method. Finally, the `if` block with the predicate "`delete.get(i) instanceof ElementNode`" is retrieved from the AST node of the `for` block.

The `finally` and `else` blocks are handled in a slightly different manner. Given the CRD of a `finally` block, the lookup-normal search traverses the AST looking for a try-catch block that matches the `anchor` description, and returns its corresponding `finally` block. Similarly, for an `else` block CRD, it looks for a sequence of `if-then` blocks that match the `anchor` description, and returns its corresponding `else` block. Lookup-normal is used for a large majority of the clone regions and terminates within a fraction of a second, as will be shown in the section on evaluation.

3.2.2 *Lookup-Conflict.* This algorithm is used whenever two or more code blocks matching the description of a CRD exists at a given location in the source code. The lookup-conflict search is similar to the lookup-normal search with one difference: when more than one code block is found at the location described by a CRD, the *conflict resolution* heuristic explained in Section 3.1 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. Although stated independently, lookup-conflict is needed whenever more than one potential target is encountered when searching for either a class, method, or code block, as in lookup-refactored.

3.2.3 *Lookup-Refactored.* Code refactoring such as class renaming, method renaming, or method movement may change the location of a clone region. In such cases, a clone region no longer exists at the location described by the CRD. The lookup-refactored search is used to recover the current location of a clone region. The first step of the lookup-refactored search is to identify the AST node for the type declaration enclosing the clone region. First, it assumes that the `<class>` is in the `<file>`. When this is not the case (e.g., is a renamed file), it searches the entire code base for type declarations with a name matching `<class>`. If a class is found, lookup-refactored then proceeds as lookup-normal. If a class is not found and a `<method>` is not specified, we

assume the class no longer exists and terminate the search. If a class is not found and a `<method>` is specified in the CRD, we employ a heuristic to locate the method described by the CRD.

The lookup-refactored search first assumes that the method has been moved to a different class. It looks for all method declarations in the system with a signature matching `<signature>`. If a method is found, lookup-refactored then proceeds as lookup-normal. If a matching method is not found, lookup-refactored assumes the method was renamed, and looks for all method declarations in the system with the same parameter list as that described by the CRD. If none is found, lookup-refactored further assumes the parameter list of the method was changed, and looks for all method declarations in the system with the same name as that described by the CRD. The result of this search is a list of potential method declarations. When more than one potential target is identified, the *lookup-conflict* search explained above is applied. Once a method declaration matching that described in the CRD is found, lookup-refactored then proceeds as lookup-normal. Our heuristics do not support changes to both the method name and parameter list since this would entail looking at every potential code block, which is unrealistic for interactive development. Our clone region lookup heuristics are similar to that employed by Xing and Stroulia [2005] in detecting structural changes between subsequent versions of a system at the design level. Whereas Xing and Stroulia used a UML representation of the components of a system to detect structural changes between versions, we used the structural properties, lexical information, and software metrics of code blocks to track clones across versions.

3.3 Discussion

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; however, lost clone relationships will be reestablished once the model is updated as explained in Section 4.2. Second, our definition of CRDs does not support anonymous classes. 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 5.1, which details the accuracy of the clone lookup algorithm with the above characteristics, substantiates our initial assumptions.

CRDs have been presented in the context of Java; notwithstanding, language constructs such as loops, decisions, and exceptions are not unique to the Java language. For instance, C++ has most of the constructs supported by our current definition of CRDs and there are IDEs that provide AST parsing for C++. CRDs can therefore be easily adapted to work for other object-oriented languages such as C++, and even procedural languages with similar constructs.

We have implemented our CRD representation and the lookup algorithms in a tool called *CloneTracker*. We present an overview of CloneTracker from the perspective of a user of the system in the next section.

4. CLONE TRACKING APPROACH

4.1 Clone Documentation, Clone Tracking, and Change Notification

Our clone tracking approach complements existing clone detection tools, refactoring technology, and clone management approaches by providing support for reusing knowledge about the location of clones in source code, and support for keeping track of clones when refactoring is considered costly, risky, not feasible, or simply when the clones are intended to be short-lived. Our current version of CloneTracker uses SimScan as the default clone detection tool, but also provides support for other tools such as Simian[8] and DECKARD [Jiang et al. 2007a]. CloneTracker is fully integrated with the Eclipse[9] platform. 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.

With CloneTracker, a developer concerned about the presence of code clones in a system sets the search options for the clone detector (for instance, with SimScan the developer may select the following configuration: Volume= medium, Similarity=fairly similar, Quality/speed=fast) and runs the tool. Simscan outputs a description of the clone relationships in the system represented as clone groups. The output is stored in a comma-separated-values file, and each clone region is represented in terms of a file name and a line range. CloneTracker provides a view to display the results of SimScan and to allow developers to indicate which clone groups to track. The results of the clone detection tool are displayed as children of the `Clone Detector` node, and the documented clone groups as children of the `Clone Documentation` node (see Figure 7).

After the clone detection process is completed, the developer must examine each reported clone group to determine an appropriate step of action. Browsing the results, the developer notices the locally unfactorable clone group in our working example (Figure 1). The clone group is represented as `Group148` in the clone detection results. The developer reached the conclusion that eliminating this clone group is risky—the clone regions are in methods with different return types—and selected clone management over refactoring. `Group148` is then transferred to the `Clone Documentation` node by the developer to initiate clone management.

CloneTracker then automatically translates the location of all the clone regions in the clone group into clone region descriptors (see Section 3), which then form an *active clone model*. The clone model now describes clones in a way that is resilient to refactoring changes such as file renaming, method renaming, or changes to a method's location. CloneTracker persists the clone model as part of the corresponding Eclipse project, hence allowing the clone model to be shared with other developers through a revision control system. On startup, CloneTracker automatically detects and loads the model.

[8] `www.redhillconsulting.com.au/products/simian/`.
[9] `www.eclipse.org`.

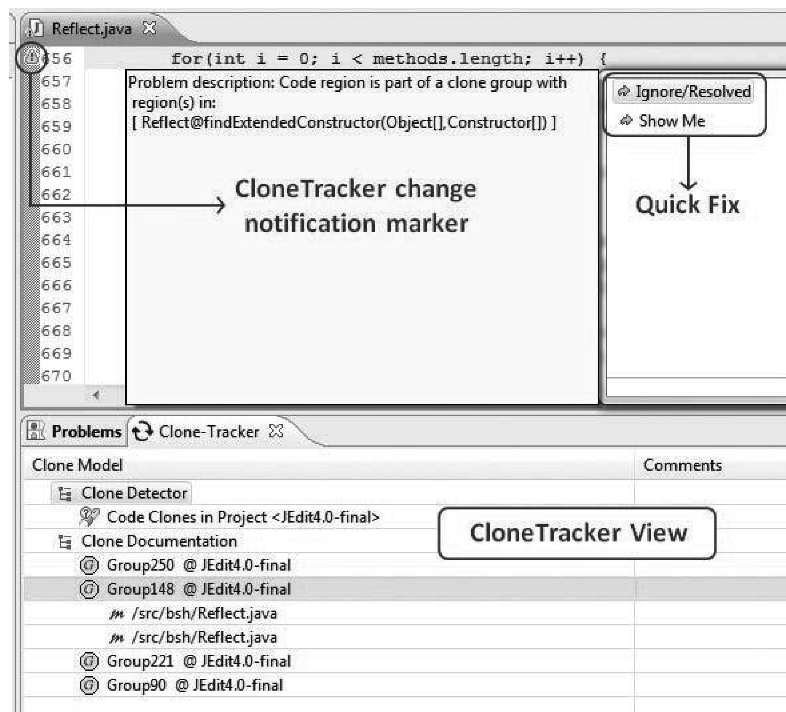

Fig. 7. The CloneTracker plug-in view and change notification features.

Developers may be aware of the presence of clones, but not sure about their current location in the source code. If, at any point in the future, a different developer with access to the clone model edits code in a clone region represented in the model, CloneTracker produces a notification that clone regions are being edited. Change notification in CloneTracker is integrated with the Eclipse warning mechanism. Our plug-in adds a warning to the Eclipse Problems View, and attaches an Eclipse warning marker at the beginning of the clone region. The message of the warning describes the clone group to which the modified region belongs. In our example, the developer is informed that the modified code region has a cloning relationship with a region in the method `findExtendedConstructor`, in the class `Reflect`, and is provided with QuickFix options (Figure 7).

The `ShowMe` option takes the developer to the current location of other code regions that have a cloning relationship with the region being modified, thus eliminating the need to repeat the clone detection and clone classification process for documented clone relationships. `ShowMe` is accomplished by highlighting the background of the clone group in the `Clone Documentation` node as yellow (shown as grey in the illustration in this article) (Figure 7). The `Ignore/Resolved` option is used to inform CloneTracker to ignore the clone region. Once selected, CloneTracker removes the marker from the clone region and from the Eclipse Problems View for the duration of the Eclipse session.

Modifications to regions of this clone group during the current session are not communicated to the developer. CloneTracker also provides support for updating the clone model to reflect the most up to date status of clone relationships in the current version of the source code. We discuss the approach for updating the clone model in the next subsection.

4.2 Updating the Clone Model

Future modifications such as the copy-and-paste reuse of a clone region, or its elimination, may invalidate the state of the documented clone relationships; hence, updates to the clone model are necessary for it to remain accurate. To provide this functionality, CloneTracker maintains a set of all the source code files that were modified between commit operations to the source code repository. These include files modified through edit operations by the developer, and those modified when the *update* command is used to bring local copies up-to-date with the newest versions on the server. We call the set of files modified between commit operations the change-set, and the set of files formed when the *change-set* is combined with the files tracked by the clone model the *delta*. Update to the model is automatically triggered after a commit operation. This begins clone detection on the files in *delta*, not the entire source code since the objective is to determine how documented relationships have changed, and not to look for new clones. Once clone detection is completed, CloneTracker generates CRDs for all the clone regions identified in the delta, and performs a two-phase comparison against the CRDs in the model to determine the status of the documented clone relationships. The first phase compares clone groups between versions, and the second phase compares clone regions within matched clone groups.

4.2.1 *Determining Clone Group Status.* In the first phase, the plug-in compares past and current clone group information, and assigns each group in the model a status of either *exists* or *disappeared* based on the results of the comparison. A documented clone group is assigned the status *exists* if it has one or more clone regions in common with a clone group in the delta; otherwise, the documented clone group is assigned the status disappeared. Disappearance may be due to refactoring or divergence of clone regions. CloneTracker identifies disappeared clone groups with a *Gx* icon (Figure 8).

4.2.2 *Group Evolution Pattern.* In the second phase, the plug-in determines how clone groups with status *exists* have changed in the delta. This is accomplished by comparing clone groups in the *model* against their counterparts in the *delta* (Figure 9). The changes are described using the clone group evolution patterns introduced by Kim et al. [2005].

—*Group unchanged*. All the clone regions of the group remained unchanged in the delta, and no new region was introduced.

—*Group addition*. At least one new clone region was introduced in its counterpart in the delta (Figure 9). For example, in Figure 8 (`Group41`), the developer is informed that a clone region in the class `GUIUtilities` has been cloned in

Problems | Clone-Tracker

Clone Model	Comments
Clone Detector	
Code Clones in Project <JEdit4.0-final>	
G+ ...Group41 @ JEdit4.0-final	group addition
m (279,304) @/src/org/gjt/sp/jedit/GUIUtilities.java	
m+ (15,39) @/src/bsh/reflect/ReflectManager.java	
m (199,223) @/src/org/gjt/sp/jedit/GUIUtilities.java	
G* ...Group221 @ JEdit4.0-final	group modification
G− ...Group90 @ JEdit4.0-final	group subtraction
m (61,73) @/src/org/gjt/sp/jedit/gui/EnhancedCheckBoxMenuItem.java	
m (66,78) @/src/org/gjt/sp/jedit/gui/EnhancedMenuItem.java	
Gx Group250 @ JEdit4.0-final	group deletion
Clone Documentation	
Group250 @ JEdit4.0-final	
Group221 @ JEdit4.0-final	
Group90 @ JEdit4.0-final	
m /src/org/gjt/sp/jedit/gui/EnhancedCheckBoxMenuItem.java	
m /src/org/gjt/sp/jedit/gui/EnhancedMenuItem.java	
m− /src/org/gjt/sp/jedit/gui/MarkersMenu.java	
Group41 @ JEdit4.0-final	

Fig. 8. Updating the clone model.

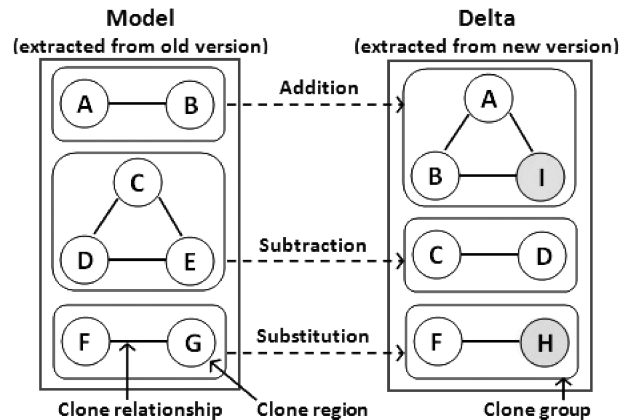

Fig. 9. Evolutionary patterns of clone groups.

the class `ReflectManager` in the delta. CloneTracker identifies such groups with a $G+$ icon, and the newly created clone regions with $m+$. Our tool informs the developer not only which clone groups have changed, but also how each group changed.

—*Group subtraction*. At least one clone region does not exist in its counterpart in the delta. For example, a clone region was refactored or diverged from the rest of the group. CloneTracker identifies such groups with a $G-$ icon, and the missing clone regions with m-. For example, in Figure 8, the developer is informed that a clone region of `Group90` in the class `MarkersMenu` no longer exists in the current version of the source code.

—*Group substitution*. An equal number of clone regions are found in the delta, but some regions are not the same as in the documented model. CloneTracker identifies such groups with a G^* icon, the substituted region with $m-$, and the replacement with $m+$.

Once completed, the developer can update the model to reflect the desired status of the clones being tracked (e.g., the developer would update the clone relationship of `Group41` with the new relation in the delta). Our update model approach only informs developers about the changes to the documented clone groups, but provides no information about clone groups that might have been introduced in newer versions of the code. To identify and track newly introduced clone groups, developers would have to repeat the clone detection process presented in Section 4.1.

5. EVALUATION

The techniques used to represent and track clone regions rely on a number of heuristics. We conducted experiments and a study to evaluate the suitability of our CRD representation, and our overall clone management approach. A comparison of the precision of CRDs to the line-based representations of multiple clone detection tools is presented in Section 5.1. Section 5.2 reports on a study that evaluates the robustness of CRDs in tracking clones across several versions of a system, and Section 5.3 reports on an experiment to determine the suitability of the algorithms for generating and looking up CRDs in interactive development.

5.1 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 clone regions technically can be 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). How does the precision of CRDs compare to line-based representation of clone regions for different clone detection tools? Is the precision of CRDs consistent for different clone detection techniques? We conducted several experiments using three clone detection tools and five subject systems, and report the results in this section of the article.

5.1.1 *Tools Used.* To work with our current version of CloneTracker, a clone detection tool must be able to detect clones in the Java source code and be able to output cloning information in the file-name and line-range format to a file, or provide an API for obtaining this information. We intended to evaluate the precision of CRDs derived from tools implementing the major clone detection techniques, because different tools may identify clones differently. This was not always feasible since some tools do not meet the requirement stated above, and thus could not be integrated with CloneTracker. Other detection

Table I. Description of Subject Systems Used in Evaluation

System	Version	kLOC	# Dev.	# Groups & % Within Code Blocks		
				SimScan	Simian	DECKARD
JMeter	2.3.2	53	>15	376(25%)	271(30%)	714(89%)
Hibernate	3.3.0	74	4	491(21%)	306(42%)	952(85%)
Eclipse PDE	20080630-1300	29	>5	281(32%)	169(40%)	698(96%)
DrJava	stable-20080106	56	>50	106(8%)	106(38%)	67(73%)
Lucene	2.3.2	25	>11	102(20%)	116(39%)	247(97%)

techniques, such as those based on metrics [Mayrand et al. 1996] or program dependency graphs [Krinke 2001], do not have publicly available tools. The tools used in our evaluation included SimScan, release 1; Simian, version 2.2.24; and DECKARD, version pre2008. SimScan is AST-based; Simian is text-based; DECKARD is based on a parse tree representation of the code.

5.1.2 *Subject Systems and Clone Groups.* To find clones for this study, we selected five Java-based open-source subject systems (JMeter, Hibernate, Eclipse PDE, DrJava, and Lucene)[10]; see Table I. These systems all have a recorded change history necessary to evaluate the robustness of CRDs, contain a significant 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.

Different tools provide different parameters, or similar parameters at different levels of granularity. For instance, SimScan uses “large,” “medium,” and “small” to define the size of clone regions, whereas Simian uses an integer value greater than 2. To standardize the results and make them comparable, we selected options that would report clones that are either Type 1 (exact copy without modifications, except for whitespace and comments) or Type 2 (syntactically identical copy; only variable, type, or function identifiers have been changed). As expressed by Koschke [2008], researchers have yet to reach a consensus on a suitable similarity measure for Type 3 clones (a copy with further modifications; statements have been changed, added, or removed). We decided to avoid Type 3 clones since we could not conclusively determine if all three tools provided search options for it. The search options used for SimScan were Volume=medium, Similarity=fairly similar, Speed/Quality=fast. The options used for Simian were ignoreIdentifier=true, ignoreModifier=true, ignoreCharacterCase=true, ignoreVariableName=true, ignoreSubtypeName=true, threshold=default. Threshold is the minimum number of lines in the clone region, and the default value is 6. The options used for DECKARD were Similarity=0.96, Min-Tokens=50, stride=0. Min-Tokens is the minimum number of tokens in the clone region, and stride is used for indicating whether comparisons should occur at the level of individual subtrees or subforests. Stride=0 means comparison should occur at the level of individual subtrees as this produces clones of greater similarity.

[10]JMeter: `jakarta.apache.org/jmeter`; Hibernate: `hibernate.org`; Eclipse PDE: `eclipse.org/pde`; DrJava: `www.drjava.org`; Lucene: `lucene.apache.org`.

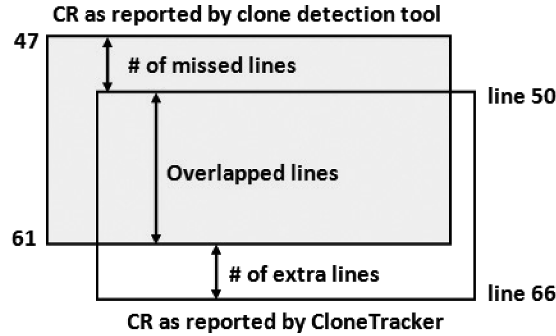

Fig. 10. Comparison of the overlap between the line-based representation of a clone region (CR) and its representation in CRDs. In this case, we have an overlap with 11 overlapped lines, three missed lines, and five extra lines, and a degree of overlap of 63%.

Table I reports the number of clone groups identified by the clone detection tools for each subject system, and the proportion of the clone groups with clone regions located within methods. For instance, SimScan identified 376 clone groups in JMeter, and 25% of these groups contained clone regions located within methods. The proportion of clone regions within code blocks varied widely, but consistently for each system, across the clone detection tools, with DECKARD at the top end, followed by Simian, then SimScan. This discrepancy may be due to the differences in comparison techniques or the differences in the precision and recall of the various tools; however, this was not the objective of our evaluation. One point stands out despite these variations: the proportion of clone regions within code blocks were significant enough to warrant their own representation.

5.1.3 *Methodology*. To evaluate the precision of CRDs, we generated CRDs automatically for each clone region of the groups in Table I. Next, we attempted to map the CRDs back to the source code, and analyzed the overlap and cases requiring conflict resolution between the initial regions, as reported by the clone detection tools, and the regions as represented by CRDs. The overlap analysis is made up of two components: first, whether or not there is an overlap, and second, the *degree of the overlap*. There is an overlap between the line-based representation of a clone region(CR) and its representation in CRDs if there is at least one line in common (see Figure 10). The degree of the overlap gives the percentage of the lines of a clone region correctly mapped by CRDs, and is computed for each system as

$$\left(\frac{\textit{Average length of CRs}}{\textit{Average length of CRs} + \textit{Average ML} + \textit{Average EL}}\right) * \left(\frac{\textit{Number of overlapped CRs}}{\textit{Total number of CRs}}\right).$$

Missed lines (ML) represent the number of lines in the original clone region that are not mapped by the CRD; *extra lines* (EL) represent lines not in the original clone region but mapped by our CRD.

Table II. Precision of CRDs Across Different Clone Detection Tools for JMeter

	SimScan		Simian		DECKARD	
# of CRs	1028		741		2559	
# of Overlaps	1023	(99%)	738	(99%)	2526	(98%)
Avg. length of CR	21		9		12	
Avg. ML per CR	2		2		2	
Avg. EL per CR	3		7		2	
Degree of Overlap	80%		50%		74%	
# of Conflicts	23		15		245	
# of Resolved Conflicts	21	(95%)	14	(93%)	216	(88%)

Table III. Precision of CRDs Across Different Clone Detection Tools for Hibernate

	SimScan		Simian		DECKARD	
# of CRs	1345		849		3548	
# of Overlaps	1330	(98%)	823	(97%)	3404	(95%)
Avg. length of CR	26		10		11	
Avg. ML per CR	6		2		2	
Avg. EL per CR	6		8		2	
Degree of Overlap	67%		49%		70%	
# of Conflicts	80		91		492	
# of Resolved Conflicts	79	(98%)	89	(97%)	456	(92%)

Table IV. Precision of CRDs Across Different Clone Detection Tools for Eclipse PDE

	SimScan		Simian		DECKARD	
# of CRs	835		504		2865	
# of Overlaps	831	(99%)	497	(98%)	2836	(98%)
Avg. length of CR	17		8		9	
Avg. ML per CR	2		3		2	
Avg. EL per CR	1		4		2	
Degree of Overlap	84%		53%		68%	
# of Conflicts	23		52		178	
# of Resolved Conflicts	22	(95%)	48	(92%)	168	(94%)

Specifically, for each clone region, we

(1) recorded the line range of the region, as reported by the clone detection tool;
(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;
(5) verified overlap, and recorded the number of missed and extra lines.

5.1.4 *Results.* Tables II to VI summarize and compare the results for each subject system for SimScan, Simian, and DECKARD. Table VII provides an aggregate of the results for each clone detection tool over all systems. These numbers have been rounded to the closest integer.

Table V. Precision of CRDs Across Different Clone Detection Tools for DrJava

	SimScan		Simian		DECKARD	
# of CRs	470		321		153	
# of Overlaps	466	(99%)	309	(96%)	146	(95%)
Avg. length of CR	24		9		18	
Avg. ML per CR	1		2		3	
Avg. EL per CR	2		9		7	
Degree of Overlap	88%		44%		62%	
# of Conflicts	0		7		4	
# of Resolved Conflicts	0	(–)	7	(100%)	4	(100%)

Table VI. Precision of CRDs Across Different Clone Detection Tools for Lucene

	SimScan		Simian		DECKARD	
# of CRs	257		256		601	
# of Overlaps	249	(96%)	226	(88%)	441	(73%)
Avg. length of CR	17		8		10	
Avg. ML per CR	2		2		2	
Avg. EL per CR	2		7		3	
Degree of Overlap	78%		42%		49%	
# of Conflicts	2		11		34	
# of Resolved Conflicts	1	(50%)	10	(90%)	32	(94%)

Table VII. Aggregation of the Precision of CRDs for Each Clone Detection Tool

	SimScan		Simian		DECKARD		ALL	
# of CRs	3935		2689		9726		16350	
# of Overlaps	3899	(99%)	2612	(97%)	9353	(96%)	15864	(97%)
Avg. length of CR	21		9		12		14	
Avg. ML per CR	3		2		2		2	
Avg. EL per CR	3		7		3		4	
Degree of Overlap	77%		49%		68%		68%	
# of Conflicts	128		176		953		1257	
# of Resolved Conflicts	123	(96%)	168	(95%)	876	(92%)	1167	(93%)

5.1.5 *Is There an Overlap?.* The second and the third row of each table present the results of the overlap. The second row (# of CRs) of each table presents the number of clone regions identified in the system by each clone detection tool; the third row (# of Overlaps) presents the number of clone regions for which the region mapped from the CRD overlapped with the original region. The results of the overlap were consistent across all three clone detection tools for JMeter, Hibernate, Eclipse PDE, and DrJava, with overlap ranging from 95% to 99%. Nonoverlapping regions for these four systems were primarily due to unresolved conflicts and the limitations of CRDs as described in Section 3.2. The results for Lucene, Table VI, were not as impressive as the other four systems. The overlap for DECKARD, Simian, and SimScan were 73%, 88%, and 96%, respectively. CRDs are based on programming conventions and idioms that did not occur as often in Lucene. For instance, we expect developers to provide exception handlers (that is, `catch` blocks) for exceptions that might occur within `try` blocks. However, over 15% of the clone regions identified by DECKARD in Lucene were either within `try` blocks with no `catch` statements or

`for` blocks with no termination condition. CloneTracker was therefore not able to generate an anchor to uniquely identify these clone regions. This explains the lower performance of CRDs for Lucene across the three clone detection tools. Nevertheless, even with our initial heuristics, CloneTracker reported an overlap for a large majority of the clone regions when the results are aggregated for each clone detection tool: 99% of the 3935 regions identified by SimScan, 97% of the 2689 regions identified by Simian, and 96% of the 9726 regions identified by DECKARD (see Table VII).

5.1.6 *Degree of Overlap.* The next four rows of each table report on how closely overlapping clone regions matched. The fourth row (Avg. length of CR) presents the average length of a clone region as reported by each clone detection tool, in lines of source code. The average length of a clone region was 21 for SimScan, 9 for Simian, and 12 for DECKARD. These values indicate that most of the clones detected represented significant cases of code duplication. The fifth row (Avg. ML per CR) presents the average number of *missed lines*, which was consistently below three for each system, and for all the clone detection tools. Missed lines are typically caused by a difference in the way CRDs and clone detection tools represent clone boundaries. CRDs systematically skip method headers (i.e., the whole declaration statement for a method before its curly braces, which at times spans several lines), Javadocs, and block headers, and start the clone region at the first curly brace; clone detection tools, on the other hand, typically include method headers, Javadocs, and block headers as part of the clone region. When the method or block header was on a separate line from the first curly brace, it became a missed line in our experiment. We do not consider this a problem since the method or block header is always in the vicinity of the clone region, and thus, in the view of the developer. The sixth row (Avg. EL per CR) presents the average number of *extra lines*, that is, the amount of noise included as part of a clone region. These lines result from the fact that clone regions have to be expanded to the closest enclosing block to be described by a CRD. This value was mostly below three for SimScan and DECKARD, but up to nine for Simian. This discrepancy can be explained by the difference in clone detection approaches employed by these tools. Both SimScan and DECKARD identify clones by transforming the code into a tree-based representation, and comparing nodes for similarity, whereas Simian is purely text-based, with no knowledge of the language syntax or structure. SimScan and DECKARD therefore identify clones in a way that is more conceptually related to the CRD representation than Simian. The seventh row presents the average degree of overlap for each system. The degree of overlap for each system hovered around 80%, 50%, and 70% for SimScan, Simian, and DECKARD, respectively. This was expected given that the degree of overlap is derived from both the missed and the extra lines, and as explained above the clones detected by both SimScan and DECKARD are more closely related to CRDs than the clones detected by Simian. Overall, clone regions represented by CRDs were about 70% overlapped with the regions identified by SimScan and DECKARD, but not Simian.

5.1.7 *Conflict Resolution.* The last two rows of each table report on the extent to which conflicts had to be resolved within a nesting level to find the correct code block, and the extent to which our corroboration metric helped in this operation. The seventh row (# 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. The results show that conflicts are uncommon (less than 8% of the total 16,350 clone regions identified in the study required conflict resolution), and even a simple metric can help disambiguate a majority (an average of 96% for SimScan, 95% for Simian, and 92% for DECKARD) of blocks with an otherwise equivalent representation.

5.2 Evolution Study

To evaluate the robustness of CRDs, and their effectiveness in describing clones in evolving software, we used CloneTracker to document clones in base versions of Eclipse PDE, DrJava, and JMeter, and attempted to track these clones across subsequent versions of the systems using the documented clone models. We selected these systems because of their long version history, and also because of the high number of clones in these systems that were not aligned with method boundaries.

We randomly selected 30 clone groups not aligned with method boundaries from each system, producing a total of 90 groups for the study. All clone groups studied evolved as part of the changes performed to different versions of the system. Each clone group was edited an average of eight times, with each clone region having an average of two lines added, three lines deleted, and six lines changed across multiple versions. In addition, two of the clone regions were located in methods affected by refactoring, and the line ranges of each clone region were modified several times during its evolution. We used CloneTracker on the base versions to automatically generate CRDs for all the clone regions studied. Then, using the generated clone models, we attempted to locate these clone regions for subsequent versions of the systems (versions 20071002-0800 to 20080414-1300 of Eclipse PDE: 40 versions in total; versions 20030924-1641 to 20080124-1942 of DrJava: 29 versions in total; and versions 1.7.3 to 2.3.2 of JMeter: 15 versions in total). Specifically, we sought answers to the following questions: how stable are the anchors of CRDs? What kinds of modifications are CRDs robust to? How effective are CRDs in tracking clones in evolving systems?

To answer these questions, we produced an explanation for every group that could not be located by CloneTracker. For instance, if a group was documented in version v_1, tracked successfully between versions v_2 and v_9, and could not be located by CloneTracker in version v_{10}, this may be because the clone group disappeared in v_{10} or the anchor was changed. We define *disappearance* and *successfully tracked* before discussing the results of the study.

—*Disappearance*. A clone group identified in version v_1 of a system is said to have *disappeared* in version $v_n, n > 1$ if it was not located by Clone-Tracker in version v_n, and has no clone regions in common with any of the

Table VIII. Evaluation of the Robustness of CRDs

	Eclipse PDE		DrJava		JMeter		All	
# of Clone groups	30		30		30		90	
Successfully tracked	27	(90%)	21	(70%)	24	(80%)	72	(80%)
Disappeared	6	(20%)	5	(17%)	17	(89%)	28	(33%)
Not successfully tracked	3	(10%)	9	(30%)	6	(20%)	18	(20%)

clones groups identified by a clone detection tool in v_n. The same clone detection tool, with the same parameter settings, is used to identify clones in both v_1 and v_n. DECKARD was used for this study because it identified the most clones within methods, and each case of disappearance was manually verified.

—*Successfully tracked*. A clone group is said to have been *successfully tracked* between versions v_2 and v_n of a system by CloneTracker if it was established that it disappeared at version $v_t, t < n$, or if all the clone regions of the clone group were located by CloneTracker for every version between v_2 and v_n, inclusively. Otherwise, we say the clone group was not successfully tracked.

The results of the study are presented in Table VIII. The third row (Successfully tracked) summarizes the robustness of CRDs for the different systems. For instance, 90% of the 30 clone groups of Eclipse PDE were successfully tracked by CloneTracker. Of those successfully tracked, 21 were tracked across the 40 versions of Eclipse PDE, and six were tracked until they disappeared from the system. The results for DrJava and JMeter were slightly different with success rates of 70% and 80%, respectively, and a disappearance rate of up to 89% for JMeter. The last row (Not successfully tracked) represents the proportion of clone groups not successfully tracked by CloneTracker. The failure to track clones was primarily due to changes in the anchors of the clone regions. For instance, the CRD below represents a clone region that was not successfully tracked due to a change in the anchor of the enclosing `if` code block.

```
org.eclipse.pde.internal.core.PluginPathFinder.java, PluginPathFinder
getFeaturePaths(String)
if, file.exists()
```

This clone region was tracked across seven versions, but in version 20071023-0800 of Eclipse PDE, the anchor `file.exists()` was changed to `file != null`. Since our current version of CloneTracker stores anchors as strings, this change in the anchor of the `if` block prevented us from successfully tracking it. However, a majority of the anchors were reasonably stable as up to 80% of the anchors in the clone regions studied remained unchanged throughout several versions of our subject systems.

We also encountered two cases of method signature refactoring in the groups studied, and CloneTracker was successful in tracking the clone regions affected. For instance, after tracking two clone regions (one within the method `find ExtensionsForPlugin(String)` and the other within `findExtensionPoints`

Table IX. Execution Time for Generating and Looking-Up CRDs for Hibernate

	# Clone Regions	Total Time (ms)	Avg. Time per Clone Region
Generate-CRDs	849	8067	0.009
Lookup-normal	756 (89% of 849)	4331	0.006
Lookup-conflict	91 (10% of 849)	1400	0.016
Lookup-refactored	2 (<1% of 849)	6062	3
Total cost		19,860 ms = 20 s	

`ForPlugin(String)`) across seven versions, CloneTracker observed changes to the method signatures, both occurring at version 20071029-1800 of Eclipse PDE. The signature `findExtensionsForPlugin(String)` was changed to `find ExtensionsFor Plugin(IPluginModelBase)`, and the signature `findExtension Points ForPlugin(String)` to `findExtensionPointsFor Plugin (IPluginModel Base)`. Using our search heuristics, CloneTracker was able to detect these changes, and to identify the correct methods containing the clone regions.

In general, using CRDs and our clone tracking system allowed us to successfully track 80% of the clone groups, not aligned with method boundaries, throughout different versions of the subject systems. 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.

5.3 Performance

We conducted another experiment to determine the cost of generating the CRDs for clone regions, and to evaluate the suitability of the lookup searches in interactive development. We generated CRDs for the 849 clone regions identified by Simian in version 3.3.0 of Hibernate because Hibernate is the largest of our subject systems, and Simian had the fastest clone detection time of 10 min (on Windows Vista, Core2Duo-2.2Ghz, 2GB RAM). We then looked up the clone regions in version 3.3.1 of Hibernate using the generated CRDs, and recorded the time required for lookup-normal, lookup-conflict, and lookup-refactored. The results of the experiment are summarized in Table IX.

The average time for generating the CRD of a clone region is below 0.01 s. This includes the cost of generating an AST representation of the source file in which the clone region is located, traversing the AST to the node enclosing the clone region, and generating the anchors and corroboration metrics. The lookup-normal row represents the case where a clone region exists at the location described by the CRD, and does not require conflict resolution. This represented a large majority of the clone regions for Hibernate (89% in total), and also for all the 16,350 regions of the study presented in Section 5.1 (close to 90%). The average cost of lookup-normal is below 0.006 s. The lookup-conflict row represents the case where two or more clone regions exist at the location described by the CRD, thus requiring conflict resolution. The average cost of lookup-conflict is three times greater than lookup-normal, but below 0.02 s. The lookup-refactored row represents the case where a clone region is not at the location described by the CRD, thus requiring the use of the lookup-refactored

search heuristic. The cost of lookup-refactored is proportional to the number of files in the system since the algorithm assumes the clone region might have been moved to any file in the system. Hibernate contains 1092 files, and the average cost is below 3 s. In general, it took about 20 s to generate the CRDs for the 849 clone regions identified in one version of Hibernate and to lookup the clone regions in a subsequent version. We conclude that the cost of generating the CRD of a clone region, and of tracking a clone region given its CRD, is fast enough to be used in interactive mode in a development environment.

6. 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, clone management, and source code representations.

—*Clone detection*. A vast body of work exists on techniques to efficiently detect and analyze clones in source code, for instance, Kamiya et al. [2002], Jiang et al. [2007a], and Basit and Jarzabek [2007]. Token-based clone detection tools take as input the source code text of a software system, preprocesses the text (e.g., to break lines into tokens and to remove nonessential differences such as comments and white spaces), and then perform a similarity analysis on the transformed input. In their presentation of the CCFinder tool, Kamiya et al. [2002] provided a clear and thorough description of this type of clone detection technology. Text-based techniques, such as Simian, compare entire lines to each other, with little preprocessing of the text. Consecutive lines are then grouped to form larger clone regions. Other clone detection approaches, such as SimScan, transform the code into an AST, and perform a pairwise comparison of the nodes in the AST to identify similar subtrees. DECKARD, on the other hand, transforms the code into a parse tree, and represents subtrees with numerical vectors. The numerical vectors are then clustered using Euclidean distance, and subtrees with vectors in the same cluster are reported as clones. Other clone detection approaches have been proposed that use inputs such as 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 [Tairas 2008].

—*Clone genealogy analysis*. Kim et al.'s [2005] empirical study of code clone genealogies provided an important part of the motivation for this research. For their study, Kim et al. built a clone genealogy extraction tool. This tool uses 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.

However, our work differs fundamentally from that of Kim et al. [2005] both in the objectives and representation of clone regions. Kim et al. sought

to understand the frequency and ways in which clones change, and the life span of clones in a system. Our objective was to look for a better way of representing and managing clone relationships in evolving software. To study clone evolution, Kim et al. first ran the clone detection tool on every version of their subject systems to identify clone groups. Then they matched similar clone groups from different versions using textual similarity measures based on the entire text of the clone regions. Our representation, CRDs, avoids the need to rerun clone detection to recover clones that have already been documented, and uses a unique feature of the clone region, such as the loop termination condition, not the entire text of the region, to track clones.

—*Bug detection*. Jiang et al. [2007b] defined three types of inconsistencies that might be introduced in systems when developers fail to appropriately customize duplicated code regions in their new context, and proposed an approach for detecting such inconsistencies. The proposed approach consists of three major phases: first, clone groups are identified using a clone detection tool; second, for each clone group, potential inconsistencies between clone regions are identified; and finally, heuristics are employed to identify clone-related bugs from these inconsistencies. In a different study of how clones are maintained, Aversano et al. [2007] found that a nonnegligible number of clones (18%) were not consistently modified, and modifications to 13 out of 17 bug fixes involving clones were not propagated to sibling clone regions. These projects on the analysis of clone-related bugs further motivate the need for clone management techniques, such as CloneTracker, that support change notification.

—*Clone management*. Previous work on clone management has mostly focused on techniques for simultaneously modifying clone regions. Miller and Myers [2001] proposed using *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, and therefore are not as resilient as CRDs to changes that occur out of the supported environment. A technique similar to LAPIS, called *linked editing*, has been proposed by Toomin et al. [2004]. The technique is implemented in a tool called *Codelink*, that allows a user to manually select clone regions and to link them. To track clones, Codelink stores a tokenized version of the text in the clone regions, and employs token comparison techniques to locate the clone regions for different versions. CRDs, on the other hand, rely on a unique feature of the clone region, such as the loop termination condition, not the entire text of the region, and are therefore relatively lightweight.

—*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 [Robillard and Murphy 2007], Aspect Browser [Griswold et al. 2001], Intentional Views [Mens et al. 2002], and

CME [Harrison et al. 2004]). 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 [Harbulot and Gurd 2006]. 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.

7. CONCLUSION

The elimination of certain clone groups in the source code of a system is not always feasible or practical. Such clone groups increase the risk of regression faults if appropriate notification mechanisms are not available during the evolution of the source code. We have developed CloneTracker, a tool capable of automatically generating abstract representations for clone regions from the output of a clone detection tool, detect modifications to tracked clone regions, and notify developers about modifications to clones. 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. In our evaluation of CRDs, 97% of the 16,350 clone regions we studied were successfully mapped back to their corresponding line-based representation, with the average percentage of overlapped lines reaching 70% for tree-based clone detection techniques. A study of CRDs defined on evolving source code also showed we could track the majority of the clone groups investigated, with 80% of the CRDs remaining accurate for the entire lifetime of the clone regions tracked. We expect that further adjustments are bound to provide mostly marginal improvements, as the evidence we have collected so far indicates that CRDs are a practical and robust representation for tracking code clones in evolving software. In our ongoing work, we are considering other possible applications of CRDs outside the domain of code clones.

ACKNOWLEDGMENTS

The authors would like to thank Barthélémy Dagenais and the anonymous reviewers for their valuable comments on earlier revisions of this article.

REFERENCES

AVERSANO, L., CERULO, L., AND PENTA, M. D. 2007. How clones are maintained: An empirical study. In *Proceedings of the 11th European Conference on Software Maintenance and Reengineering*. 81–90.

BAKER, B. S. 1995. On finding duplication and near-duplication in large software systems. In *Proceedings of the 2nd Working Conference on Reverse Engineering*. 86–95.

BASIT, H. A. AND JARZABEK, S. 2007. Efficient token based clone detection with flexible tokenization. In *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*. 513–516.

FOWLER, M. 2000. *Refactoring—Improving the Design of Existing Code*. Addison-Wesley, Reading, MA.

GEIGER, R., FLURI, B., GALL, H. C., AND PINZGER, M. 2006. Relation of code clones and change couplings. In *Proceedings of the 9th International Conference of Funtamental Approaches to Software Engineering*. Lecture Notes in Computer Science, vol. 3922. Springer, Berlin, Germany, 411–425.

GILL, G. K. AND KEMERER, C. F. 1991. Cyclomatic complexity density and software maintenance productivity. *IEEE Trans. Softw. Eng. 17*, 12, 1284–1288.

GRISWOLD, W. G., YUAN, J. J., AND KATO, Y. 2001. Exploiting the map metaphor in a tool for software evolution. In *Proceedings of the 23rd International Conference on Software Engineering*. 265–274.

HARBULOT, B. AND GURD, J. R. 2006. A join point for loops in AspectJ. In *Proceedings of the International Conference on Aspect-Oriented Software Development*. 63–74.

HARRISON, W., OSSHER, H., SUTTON JR., S., AND TARR, P. 2004. Concern modeling in the concern manipulation environment. Tech. rep. RC23344. IBM Research, Yorktown Heights, NY.

JABLONSKI, P. AND HOU, D. 2007. CReN: A tool for tracking copy-and-paste code clones and renaming identifiers consistently in the IDE. In *Proceedings of the OOPSLA Workshop on Eclipse Technology eXchange*. 16–20.

JIANG, L., MISHERGHI, G., SU, Z., AND GLONDU, S. 2007a. Deckard: Scalable and accurate tree-based detection of code clones. In *Proceedings of the 29th International Conference on Software Engineering*. 96–105.

JIANG, L., SU, Z., AND CHIU, E. 2007b. Context-based detection of clone-related bugs. In *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*. 55–64.

KAMIYA, T., KUSUMOTO, S., AND INOUE, K. 2002. CCFinder: A multi-linguistic token-based code clone detection system for large scale source code. *IEEE Trans. Softw. Eng. 28*, 7, 654–670.

KAPSER, C. AND GODFREY, M. W. 2006. "Cloning considered harmful" considered harmful. In *Proceedings of the 13th Working Conference on Reverse Engineering*. IEEE Computer Society Press, Los Alamitos, CA, 19–28.

KIM, M., BERGMAN, L., LAU, T., AND NOTKIN, D. 2004. An ethnographic study of copy and paste programming practices in OOPL. In *Proceedings of the International Symposium on Empirical Software Engineering*. 83–92.

KIM, M. AND NOTKIN, D. 2006. Program element matching for multi-version program analyses. In *Proceedings of the 3rd International Workshop on Mining Software Repositories*.

KIM, M., SAZAWAL, V., NOTKIN, D., AND MURPHY, G. C. 2005. 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*. 187–196.

KOSCHKE, R. 2008. *Identifying and Removing Software Clones*. Springer, Berlin, Germany, Chap. 2, 15–36.

KRINKE, J. 2001. Identifying similar code with program dependence graphs. In *Proceedings of the 8th Working Conference on Reverse Engineering*. 301–309.

LANGE, B. M. AND MOHER, T. G. 1989. Some strategies of reuse in an object-oriented programming environment. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. 69–73.

MAYRAND, J., LEBLANC, C., AND MERLO, E. 1996. Experiment on the automatic detection of function clones in a software system using metrics. In *Proceedings of the 1996 International Conference on Software Maintenance*. 244–253.

MCCABE, T. J. 1976. A complexity measure. *IEEE Trans. Softw. Eng. 2*, 4, 308–320.

MENS, K., MENS, T., AND WERMELINGER, M. 2002. Maintaining software through intentional source-code views. In *Proceedings of the 14th International Conference on Software Engineering and Knowledge Engineering*. 289–296.

MILLER, R. C. AND MYERS, B. A. 2001. Interactive simultaneous editing of multiple text regions. In *Proceedings of the USENIX Annual Technical Conference*. 161–174.

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

ROBILLARD, M. P. AND MURPHY, G. C. 2007. Representing concerns in source code. *ACM Trans. Softw. Eng. Methodol. 16*, 1, Article 3 (Feb.).

ROBILLARD, M. P. AND WEIGAND-WARR, F. 2005. ConcernMapper: Simple view-based separation of scattered concerns. In *Proceedings of the OOPSLA Workshop on Eclipse technology eXchange*. 65–69.

TAIRAS, R. 2008. Bibliography of code detection literature. http://www.cis.uab.edu/tairasr/clones/literature/.

TOOMIM, M., BEGEL, A., AND GRAHAM, S. L. 2004. Managing duplicated code with linked editing. In *Proceedings of the IEEE Symposium on Visual Languages and Human Centric Computing*. 173–180.

XING, Z. AND STROULIA, E. 2005. In *Proceedings of the 20th International Conference on Automated Software Engineering*. 54–65.

Received September 2008; revised January 2009; accepted March 2009