Actively Comparing Clones Inside The Code Editor

Ferosh Jacob
Computer and Information Sciences
University of Alabama at Birmingham
Birmingham, AL 35294
jacobf@uab.edu

Daqing Hou
Electrical and Computer Engineering
Clarkson University
Potsdam, NY 13699
dhou@clarkson.edu

Patricia Jablonski
Engineering Science
Clarkson University
Potsdam, NY 13699
jablonpa@clarkson.edu

ABSTRACT
Tool support for code clones can improve software quality and maintainability. While significant research has been done in locating clones in existing source code, there has been less of a research focus on proactively tracking and supporting copy-paste-modify operations, even though copying and pasting is a major source of clone formation and the resulting clones are then often modified. We designed and implemented a programming editor, based on the Eclipse integrated development environment, named CSeR (Code Segment Reuse), which keeps a record of copy-and-paste-induced clones and then tracks and visualizes the changes made to a clone with distinct colors. The core of CSeR is an algorithm that actively compares two clones for detailed differences as a programmer edits either one of them. This edit-based comparison algorithm is unique to CSeR and produces more immediate, accurate, and natural results than other differencing tools.

Categories and Subject Descriptors
D.2.3 [**Software Engineering**]: Coding Tools and Techniques – *object-oriented programming.* D.2.6 [**Software Engineering**]: Programming Environments – *integrated environments.* D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement.

General Terms
Algorithms, Experimentation, Languages.

Keywords
Code clone, copy-and-paste programming, code comparison, differencing tools, Eclipse integrated development environment, Java, software evolution, software maintenance.

1. INTRODUCTION
For many programmers, copying and pasting is an inherent component of their coding practice [28]. In addition to numerous other justifications (see, for example, [27, 28]), this common practice seems to have a root in human cognition [13, 14]. In general, working with concrete examples may yield many advantages [13, 39]. It is thus surprising that existing programming editors do not offer much support for the copy and paste of code other than the usual editing operations.

IWSC2010 May 8, 2010, Cape Town, South Africa
Copyright 2010 ACM 978-1-60558-980-0/10/05…$10.00

A lot of information can be extracted from copy-and-paste actions, which has a strong potential to be leveraged to help programmers. The first example is the cloning relationship. Miryung Kim, et al., [28] observe, “In fact programmers employ their memory of copy-and-paste history as they make changes to code or decide when to restructure code.” They continue, “However, a programmer’s recollection of copy-and-paste history can be short-lived, somewhat inaccurate, and difficult to transfer from person to person.” In other words, there is an invisible relationship between the code fragments that are copied and pasted. This cloning relationship can be lost if the integrated development environment (IDE) does not track the copy-and-paste operations proactively as they happen.

Even if the cloning relationship can be guessed or detected automatically and retroactively, the programmer may still need to know the detailed differences between the clones in order to understand and properly maintain them. Such detailed information includes how the clones differ such as where new statements or sub-expressions were inserted and which statements have been deleted or moved. This fine-grained information is useful as it can facilitate understanding between clones. Knowing the commonalities and differences between related clones within the same clone group can also help the programmer maintain the similarity between the clones that is necessary to remain unchanged, while making sure that the differences are consistent and error-free. Finally, this information can also potentially act as a guide for similar copy-paste-modify operations in the future.

However, comparing clones in detail is a time-consuming process. The differences between two clones can be hard to spot, especially when the two clones share a large amount of common code that often buries other relatively small changes. Furthermore, when the two clones are not immediately visible at the same time in the same editor view, which happens frequently in practice, the programmer often has to move back and forth between the two clones, just to confirm one fact at a time. This is extremely time-consuming and tedious. However, their programming task often requires and indeed can benefit from an immediate overview of the similarity and differences between the clones. Therefore, there is a mismatch between programmer needs and available tool support. In fact, usability research has shown that making such information readily visible can be very helpful when a user is working with artifacts [19].

The CSeR (Code Segment Reuse) tool has been designed to track the copy-and-paste operations and store the cloning relationships between the related code fragments. In addition, it maintains the correspondence between the copied-and-pasted code by tracking editing actions and interactively parsing and inferring change categories. These user-made changes in the modified code are shown within the CSeR editor, making the clone commonalities and differences immediately obvious to the programmer.

CSeR represents a part of our overall effort in investigating tool support for the proactive tracking and management of copy-and-paste-induced clones over time for better coding efficiency [22, 24]. The specific goal for the CSeR software tool is to display detailed source code commonalities and differences in real-time as clones are modified, and to produce results that are more accurate, natural, and immediate than existing tools provide.

2. EXAMPLE SCENARIO

We will use a pair of classes from Eclipse JDT (Java Development Tools) framework to illustrate how CSeR tracks the cloning relationship and highlights the differences between a pair of cloned classes.

SetFilterWizardPage is a class from the Eclipse JDT representing one of the wizard pages in the new project creation wizard. In particular, SetFilterWizardPage specifies which sub-folders and files should be included or excluded on the build path of a newly created project using the so-called inclusion and exclusion patterns. ExclusionInclusionDialog, on the other hand, is a class with a similar purpose, but intended to be used after a project has been created. From a brief study into the history of these two classes, we have determined that this functionality of inclusion and exclusion patterns was earlier implemented as a dialog, in the ExclusionInclusionDialog class. Later, the functionality was made available inside the project creation wizard as it is currently, hence the creation of SetFilterWizardPage. The user interfaces that the classes produce are nearly identical such that the SetFilterWizardPage only has a different window title, an extra icon, and "Back" and "Next" buttons along the bottom of the window (due to the fact that it is a wizard page), which the ExclusionInclusionDialog window does not have. The earlier copy was never deleted and both classes were maintained in subsequent releases of Eclipse.

Our study of the two classes reveals that it is likely that SetFilterWizardPage was copied from ExclusionInclusionDialog. However, the detailed source code differences are not immediately obvious to a programmer who is not familiar with these two classes as he or she inspects and compares them. Tools such as Eclipse's Outline view are not very helpful either when a programmer compares the two classes for the first time. For example, corresponding structural elements between the two classes (which are in different files) and deleted elements (which are now present in only one of the classes) are not explicitly labeled in the Outline view for the programmer to easily see the relationship.

We have recreated this scenario within the CSeR editor, shown in Figure 1. CSeR is an Eclipse plug-in that extends the Eclipse Java editor without disturbing any of its existing features. CSeR currently supports the cloning of a class file (comparing only two class files at a time). In particular, for this example, the programmer would right click on the ExclusionInclusionDialog class file in Eclipse's Package Explorer view and copy and paste it to make the new class file SetFilterWizardPage.

When initially pasted, the two classes are identical except for the refactoring of the class and file name. Once the programmer starts making modifications to the pasted code, the changes will be detected and shown directly in CSeR's editor. CSeR determines or infers each user-made change as an insert, delete, update, or move. Unchanged code within a clone is not highlighted. The four kinds of user-made differences between related clones are

1. **Insert** – the addition of an AST (abstract syntax tree) node, highlighted in green.
2. **Delete** – the removal of an existing AST node, highlighted in red.
3. **Update** – the modification of an existing AST node, highlighted in yellow.
4. **Move** – the differences between the matching statements are that they have different neighbors, highlighted in blue.

Mouse hover events reveal details about the change, including what the updated code was before in the original and what has been deleted from the original.

In Figure 1, the change labeled (1) is categorized as an update change. When the programmer hovers the mouse over this highlighted update code, it shows that StatusDialog was changed (indicated by an arrow) to NewElementWizardPage. In other words, the ExclusionInclusionDialog class extends StatusDialog, while the newly modified SetFilterWizardPage class extends NewElementWizardPage.

The change labeled (2) shows the insertion of the field String PAGE_NAME and (6) and (7) are two newly added parameters (ArrayList existingEntries and IPath outputLocation). Other insertions throughout the class are also highlighted in green.

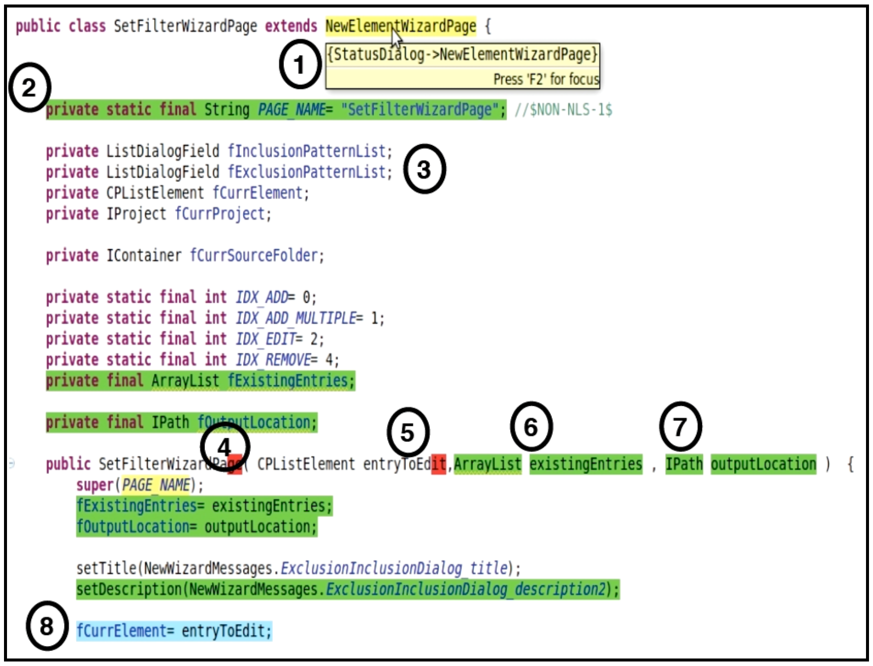
The group of fields labeled (3) in Figure 1 shows no highlighting from CSeR, which tells the programmer that these are currently the common elements between the clones (and are likely to remain unchanged over time in order to maintain the necessary level of similarity between the copied-and-pasted code).

The two changes labeled (4) and (5) correspond to the deletion of two parameters, one (Shell parent) before and one (Boolean focusOnExcluded) after the common parameter CPListElement entryToEdit. Finally, the change labeled (8) indicates a move operation from several lines above.

Since CSeR relies on parsing and exploits syntax, it is different from UNIX Diff and, in general, tools that are based on text differencing algorithms. CSeR is also unique in that it is interactive and integrated with code editors. In addition, CSeR is designed to work with fragments of cloned code and does not necessarily require whole files as clones. Tools designed for comparing two files, such as Eclipse Compare Editor, are not ideal, or at least are inconvenient, for comparing arbitrary code fragments.

3. CSER IMPLEMENTATION

The core data structure that CSeR operates on is a *bi-directional map* for pairs of corresponding code elements from two clones. Files opened inside of the Eclipse IDE editor are represented as document objects. A document object is basically a sequence of positioned characters. (In fact, the map tracks the *position* of each program element, which consists of the index of its first character in the document and the total number of characters that it contains.) Each edit to a document will activate the listeners linked with the document. These listeners are given with the



```
public class SetFilterWizardPage extends NewElementWizardPage {
                                  {StatusDialog->NewElementWizardPage}
                                                   Press 'F2' for focus

    private static final String PAGE_NAME= "SetFilterWizardPage"; //$NON-NLS-1$

    private ListDialogField fInclusionPatternList;
    private ListDialogField fExclusionPatternList;
    private CPListElement fCurrElement;
    private IProject fCurrProject;

    private IContainer fCurrSourceFolder;

    private static final int IDX_ADD= 0;
    private static final int IDX_ADD_MULTIPLE= 1;
    private static final int IDX_EDIT= 2;
    private static final int IDX_REMOVE= 4;
    private final ArrayList fExistingEntries;

    private final IPath fOutputLocation;

    public SetFilterWizardPage( CPListElement entryToEdit,ArrayList existingEntries , IPath outputLocation ) {
        super(PAGE_NAME);
        fExistingEntries= existingEntries;
        fOutputLocation= outputLocation;

        setTitle(NewWizardMessages.ExclusionInclusionDialog_title);
        setDescription(NewWizardMessages.ExclusionInclusionDialog_description2);

        fCurrElement= entryToEdit;
```

Figure 1. CSeR shows the changes that would be made to the ExclusionInclusionDialog class (highlighted code for inserts, deletes, updates, moves; and hover information for deletes, updates) to make the SetFilterWizardPage class. This figure should be viewed in the electronic copy of this paper to see color effects.

positions where the edit occurred and the nature of the edit (insert, delete, or replace). Based on the provided information, these listeners can maintain the integrity of the bi-directional map. For example, when a character is inserted or removed, the positions of all tracked program elements behind the edited location will be updated and shifted consistently. Because the CSeR map is bi-directional, it supports comparisons with either one of the clones as the origin and the other as the target. In this paper, we focus on the more fundamental operation of comparing a pair of clones. It should be feasible to extend this to handle three or more clones.

The copy-and-paste operations are tracked within the development environment. Initially, after a clone is created, it is parsed to obtain the necessary abstract syntax tree (AST) nodes in order to establish the correspondence map between the clone and its origin. Currently, the CSeR map tracks fields, methods, parameters, conditional expressions, method calls, simple names, and literal constants.

When a clone is edited within the code editor, the correspondences between the clone and its peer will need to be maintained or modified for consistency. For performance consideration, this is achieved by identifying the smallest sub-tree that contains the positions where the programmer last edited within the modified clone. For every user-edit, the tool relies on parsing to find the smallest sub-tree and compares it with the corresponding sub-tree from within the peer. If an edit makes parsing impossible, its position will be remembered and used to identify the smallest sub-tree when the code can be parsed successfully again.

Once the smallest containing sub-trees are identified, detailed clone differences such as insert, delete, update, and move can be determined by parsing the smallest sub-trees into parts and checking whether each part appears in the correspondence map. A part that appears in the map will be compared with its counterpart in the peer clone to conclude whether it has been updated. All parts from the respective clones that do not appear in the map are further compared pairwise in terms of several similarity metrics, including the Levenshtein distance. If a program element is considered dissimilar to any, it will be recorded as insert or delete, depending on the direction of comparison. If a pair of program elements is similar enough, they will be further checked as to whether their immediate neighbors correspond in the map. If they do, the pair will be added back into the correspondence map. Otherwise, they will be treated as move.

The identified changes are recorded and saved over IDE sessions, and can be stored in version control systems as well.

4. VALIDATION

The CSeR tool is designed to compute clone differences interactively while a programmer is typing and editing code. Thus, an important concern for validation is whether the tool can deal with all possible editing scenarios. A second concern is the usefulness of the tool. We can (partially) determine the usefulness of a tool by using it and examining and assessing the difference that it can possibly make. As an initial evaluation, we identified and applied CSeR to a set of test cases from real-world scenarios. This has also provided us with the first set of statistics as to how clones are actually changed. Furthermore, it shows evidence that CSeR's heuristics for inferring change types work well in presenting *natural* changes to a programmer (for example, classifying a change as an update rather than a pair of delete and insert operations can be considered more natural).

Table 1. Common editing scenarios and whether CSeR supports that kind of edit.

Num.	Type	Goal Description	Action Description	CSeR Support?
1	Name	Creating a name	Paste, Type	Yes
2		Replacing a part	Paste, Type	Yes
3		Correcting typos	Backspace, Type	Yes
4		Replacing a name	Backspace, Paste, Type	Yes
5		Removing a name	Backspace, Delete, Type	Yes
6		Splitting a name	Type in between	Yes
7		Renaming	Use tools	No
8	List	Creating a new list	Paste, Type	Yes
9		Inserting a new element	Paste, Type	Yes
10		Removing an element	Backspace, Delete, Type	Yes
11		Moving an element	CutPaste/CopyPasteDel, Type	Yes
12		Removing an entire list	Backspace, Delete, Type	Yes
13		Flattening a list in a list	Backspace, Delete, Type	Yes
14	Expression	Inserting a new expression	Paste, Type	Yes
15		Updating an expression	Backspace, Delete, Type	Yes
16		Moving an expression	CutPaste/CopyPasteDel, Type	Yes
17		Removing an expression	Backspace, Delete, Type	Yes
18	Comment	Commenting code	Type comment symbols	Yes
19		Creating an annotation	Paste, Type	No
20		Commenting in expression	Type comment symbols	Yes
21	Keyword	Inserting a keyword		No
22		Removing a keyword		No
23		Updating a keyword		No

4.1 Completeness and Robustness in Editing Support

A study of programmers' editing behavior by Andrew J. Ko, et al., has summarized some common editing scenarios [33]. We tested CSeR using the results of this study as a foundation. Table 1 lists twenty-three scenarios collected from their study [33] and whether CSeR supports a certain kind of edit. There is one goal (or purpose of editing) for every edit, but there can be many actions or different ways to achieve the goal. We distinguished between the goals and means of editing in order to get a grip of a complete list of editing scenarios. When we tested CSeR with the collected test cases, we applied all of the editing scenarios to make sure that CSeR worked as expected.

In Table 1, names refer to anything that is not a keyword such as method names, class names, and variable names. Lists correspond to structures that appear between list delimiters, such as curly brackets surrounding lists of statements or parentheses surrounding lists of parameters.List elements are delimited by single characters such as semicolons between statements and commas between parameters.

CSeR currently supports eighteen of the twenty-three editing scenarios in Table 1. For each of the eighteen scenarios, at least one representative test case is used to demonstrate that CSeR can properly handle it [24]. Although not listed in Table 1, we have also verified that CSeR can handle undo actions as well as the case when code becomes temporally un-parsable [24].

For the five editing scenarios that CSeR does not handle now, we have verified that they do not represent fundamental design problems and can be supported in the future without significantly changing the existing design of CSeR. Three of the five unsupported scenarios are related to keywords (rows 21 through 23). Although programmers do change keywords, for example, from "public" to "private", or from "if" to "while", such changes tend to be rare and we chose not to support them in the current prototype of CSeR. But CSeR can be easily extended to support them if so desired. The scenario for creating annotations (row 19) is not supported for the same reason. Finally, although supporting Rename Refactoring (row 7) is certainly technically feasible, it would require extensive modifications to the refactoring infrastructure of Eclipse, and, thus, we decided not to implement it for this version of CSeR. We conclude that the current version

Table 2. Summary of 533 user-made changes identified from the 37 pairs of clones.

Num.	Change Distribution	Description	Internal Distribution
1	Update (49%, 261)	Variable Name (V)	49%
2		Variable Type (T)	26%
3		Method (M)	15%
4		Literal (L)	8%
5		Other (O)	<2%
6	Delete (33%, 177)	Statement (S)	40%
7		Method Declaration (M)	28%
8		Field Declaration (F)	21%
9		Expression (E)	8%
10		Parameter (P)	<1%
11		Class Declaration (C)	<1%
12	Insert (16%, 82)	Statement (S)	46%
13		Field Declaration (F)	26%
14		Method Declaration (M)	14%
15		Parameter (P)	10%
16		Expression (E)	2%
17		Class Declaration (C)	2%
18	Move (2%, 13)	Method Declaration (M)	54%
19		Statement (S)	39%
20		Class Declaration (C)	7%

of CSeR can support the most common editing scenarios and that its core design is ready to be extended to support all of the possible editing scenarios.

4.2 Usefulness

We can determine the usefulness of the tool by using it and reflecting on the difference that it can make. Of course, this does not exclude other means of validation such as a user study. Nonetheless, this initial validation demonstrates that a tool such as CSeR can be useful in real-world projects in the sense that (1) there exist non-trivial clones for CSeR to work on, and that (2) CSeR's visualization makes the differences between a pair of clones immediately obvious, which usually aids in clone code understanding.

We identified and applied CSeR to a set of clones as test cases from real-world scenarios and the clone research literature. The test cases also serve as additional evidence for the robustness of the tool. In total, we collected 37 pairs of cloned classes from real-world projects and 10 pairs of cloned code fragments from research literature. The real-world projects are sub-projects of the Eclipse project[1] (JDT UI, JDT Core, and SWT) and the JLCP project (JavaLobby Community Platform)[2]. Clones appearing in the literature are usually carefully selected by researchers to illustrate extreme or common cases and are from diverse domains. Therefore, they represent a good source of use cases that CSeR should support. All of these clones are documented in [26].

We summarize the process of how we went about identifying classes that may have been created by copying and pasting. We used a clone detection tool, CCFinderX[3], to assist in identifying a set of smaller clones as potential candidates or seeds. Based on the smaller clones identified by CCFinderX, we made a quick assessment as to whether the enclosing classes may have been copied and pasted. We then manually went through related classes in the same package or related packages. When we felt that two classes were similar, we further assessed the likelihood that they were created via copying and pasting by comparing source code characteristics such as positions, spacing, and comments, as well as the meaning and the roles of the classes in the application. The two classes used as examples in Section 2, SetFilterWizardPage and ExclusionInclusionDialog, were selected in this way. Once we found two or more cloned classes, we divided them into pairs.

For each pair, we recreated the scenario of copy-paste-modify that the programmer would have done to create a new class within the CSeR editor. At the end, CSeR was able to smoothly process all of the test cases that we collected. In addition, for each pair of clones, we collected statistics for the four kinds of changes and

[1] http://www.eclipse.org (All URLs verified January 27, 2010.)

[2] https://jlcp.dev.java.net

[3] http://www.ccfinder.net/ccfinderx.html

their detailed distributions in terms of the kinds of program elements involved in each of the four changes. They help evaluate whether some of CSeR's design decisions were based on correct assumptions and inform possible new design decisions. However, the results form a table that is too large to be presented in this paper (see [26] for details). Instead, we present a summary of the change statistics in Table 2.

The results of the change statistics in Table 2 warrant some explanation. After analyzing the user-made changes within the different classes, we created categories for each type of change. We only considered the highest level of change in our analysis. For example, a newly inserted method would be considered one method change rather than counting all of the statements within the method, expressions within the statements, etc. as newly inserted as well. We found that the majority of the changes (49%) were considered "updates" and of those, most were either variable name (49%) or type (26%) updates. This is a significant finding because instead of showing this kind of change as a deletion followed by an insert, CSeR shows the change as a single update, which would be more useful and natural, and provide the programmer with more relevant information about their coding activity.

4.3 Programmer Productivity

We believe that without CSeR, a programmer could take longer in completing a programming task that involves comparing clones, and in general, more easily make programming errors due to not knowing the cloning relationship (the similarity level that must be maintained over time) or due to not knowing all of the detailed differences between clones that must also be consistently maintained. However, to fully explore this hypothesis, it would be necessary to test CSeR's impact in three main programming areas (impacts of comparing code on understanding, on debugging, and on modifying code) versus without the tool or with other tools. We are currently planning a user study with CSeR in order to see its effect on programmer productivity (in terms of both the speed of task completion and solution correctness). The results of this study will be particularly interesting as it will demonstrate how CSeR can help improve software quality versus without it. We look forward to seeing how CSeR performs with a variety of users and programming tasks.

5. RELATED WORK

5.1 Change Information from Version Control Systems

There is a large body of research that focuses on mining software repositories and then analyzing the historical information from version control systems, such as SVN (Subversion) or CVS (Concurrent Versions System), for a variety of reasons [2, 3, 4, 8, 9, 17, 20, 29, 30, 34, 44]. However, this approach is limited since the information obtained is only from snapshots of when the program's source code was checked-in and it often requires additional analysis and inferences to be useful. Furthermore, the program histories may contain a lot of irrelevant information that is not clone-related. Given program version changes, we would need to sort through to detect likely copied-and-pasted code and eliminate extra information. Also, although we might be able to obtain information about specific changes made to a particular file, we would not automatically have correspondence information (between files) from the histories alone.

5.2 Text- or LCS-based Differencing Tools

There are many text-based differencing tools available. Most make use of the diff algorithm [23] and are based on solving the LCS (Longest Common Subsequence) problem. Since this approach is developed for text files, it has obvious disadvantages when used for Java source code. The algorithm will not be able to distinguish between source code and comments and it will consider the order of fields and methods in Java even though the ordering does not change the meaning of the class. Another disadvantage is that if the positions of two methods are interchanged and the first method is, for example, five lines long while the second is ten lines, diff will sacrifice the first method and only show the correspondence between the second methods in both classes. In short, the correspondence between smaller methods or fields is lost in the case of move operations.

Some differencing tools that are based on the diff or LCS algorithm include UNIX Diff[4], Eclipse's Compare Editor (which can be invoked by right clicking selected file(s) in Eclipse's Package Explorer view and then choosing the "Compare With" menu option), Ldiff [9], and Version Editor (ve) [3]. Ve provides tight integration of the revision history and the editor so it has the limitations and disadvantages of the text-based tools and the version control system.

5.3 Graph- or AST-based Differencing Tools

Graph-based. There are a variety of graph-based differencing algorithms [1] and tools such as Cdiff [7, 41], Jdiff [1], Semantic Diff [23], and Exas [35]. The graph-based approach has an advantage over the text-based tools, which only focused on syntax, since these take into account the program's semantics as well. However, they can be slower and it is not always clear whether the extra analysis pays off.

Tree-based. Many differencing tools are abstract syntax tree (AST)-based such as LaDiff [10], Breakaway [11], Jigsaw [12], ChangeDistiller [15], and Coogle [38], including CSeR. These tools in general have the advantage of being able to obtain structured information from the tree-based representation of the source code. CSeR differs from these tools in terms of its purpose (clone differencing), its interactive and incremental updating of correspondence rather than re-computing from scratch, (in contrast to what is done in Breakaway [11]), and its heuristics that it uses to infer change categories (which differs from, for example, those of ChangeDistiller [15]).

5.4 Mapping/Origin- and Logic/Rule-based Differencing Tools

Another way of looking at program changes is to use mapping or origin analysis as part of the differencing algorithm [30, 36] or the tool implementation such as Beagle [17]. More recently, additional logic has been incorporated as well to get a better understanding of the changes. The UMLDiff approach tracks the evolution of higher-level program elements (at the level of UML models) over versions of systems [40] and other research utilizes a novel rule-based and combination algorithm (LSdiff) [27, 29] to

[4] http://directory.fsf.org/project/diffutils

infer regular change patterns and overcome some of the disadvantages of the other differencing approaches.

5.5 Synchronous Editing Tools

A few tools have been developed that address the problem of inconsistent changes being made between code regions, LAPIS [36], Codelink [41], and CloneTracker [15], such that editing in one region will simultaneously occur in the regions that are linked to it. These kinds of tools all have ways of detecting and maintaining similarities and differences between selected code regions and ways to visually present that information to the programmer. One main difference between CSeR and these tools besides implementation details (LCS-based vs. AST-based) and features (CSeR shows detailed differences) is that CSeR proactively tracks the copy-and-paste actions from the beginning and then visualizes the edits that are made to these clones automatically. Although CSeR does not do synchronous editing at the present time, it should be feasible to extend CSeR to support edit propagation without major changes to the current design.

6. CONCLUSION AND FUTURE WORK

We have presented an approach to computing and visualizing the changes applied to clones as they are edited. The core of our approach uses a combination of edit tracking and parsing to actively maintain a correspondence map between a pair of clones. We discussed a proof-of-concept implementation, CSeR, with an initial validation of its completeness and robustness with regard to supporting common editing actions and its usefulness in terms of maintaining and understanding clones.

So far, mainly we the authors have evaluated CSeR as its first set of users. We are fully aware of this limitation and are planning for a user study. Now we consider some of the possible extensions of this tool.

Supporting clone groups and side-by-side view. CSeR currently considers comparing a pair of clones at a time, which is the foundation for comparing three or more clones in a clone group. On top of the algorithm presented here, the extension to comparing more than three clones should be feasible. This may involve introducing new ways of interaction so that a programmer can select the clones to be compared, and new ways of displaying the differences of one clone with respect to any other clone within the clone group. In particular, a side-by-side view like the Eclipse Compare Editor employs would be highly desirable [19].

Inferring clone change templates and higher-level abstractions for clone evolution. Since CSeR tracks how clones are edited, we could infer clone change templates from these edits and use the inferred templates to guide future clone evolution. It may also be useful if CSeR can detect and show higher-level abstractions, such as refactorings, from the relatively low-level changes.

7. Acknowledgments

The authors wish to thank the two anonymous IWSC reviewers for their constructive reviews, which have helped improve the presentation of this paper.

8. REFERENCES

[1] T. Apiwattanapong, A. Orso, and M.J. Harrold, “A Differencing Algorithm for Object-Oriented Programs”, *ACM SIGSOFT-SIGART-IEEE International Conference on Automated Software Engineering (ASE)*, 2004.

[2] D. Atkins, T. Ball, T. Graves, and A. Mockus, “Using Version Control Data to Evaluate the Impact of Software Tools”, *ACM SIGSOFT-IEEE International Conference on Software Engineering (ICSE)*, 1999.

[3] D.L. Atkins, “Version Sensitive Editing: Change History as a Programming Tool”, *ACM SIGPLAN-SIGSOFT European Conference on Object-Oriented Programming (ECOOP)*, 1998.

[4] T. Ball, S. Diehl, D. Notkin, and A. Zeller, “Multi-Version Program Analysis”, *Dagstuhl Seminar*, 2005.

[5] S. Bates and S. Horwitz, “Incremental Program Testing Using Program Dependence Graphs”, *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 1993.

[6] D. Binkley, “Semantics Guided Regression Test Cost Reduction”, *IEEE Transactions on Software Engineering (TSE)*, 1997.

[7] D. Binkley, R. Capellini, L.R. Raszewski, and C. Smith, “An Implementation of and Experiment with Semantic Differencing”, *IEEE International Conference on Software Maintenance (ICSM)*, 2001.

[8] G. Canfora, L. Cerulo, and M. Di Penta, “Identifying Changed Source Code Lines from Version Repositories”, *ACM SIGSOFT-IEEE International Workshop on Mining Software Repositories (MSR)*, 2007.

[9] G. Canfora, L. Cerulo, and M. Di Penta, “Tracking Your Changes: A Language-Independent Approach”, *IEEE Software*, 2009.

[10] S.S. Chawathe, A. Rajaraman, H. Garcia-Molina, and J. Widom, “Change Detection in Hierarchically Structured Information”, *ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 1996.

[11] R. Cottrell, J.J.C. Chang, R.J. Walker, and J. Denzinger, “Determining Detailed Structural Correspondence for Generalization Tasks”, *European Software Engineering Conference (ESEC) and ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE)*, 2007.

[12] R. Cottrell, R.J. Walker, and J. Denzinger, “Semi-automating Small-Scale Source Code Reuse via Structural Correspondence”, *ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE)*, 2008.

[13] F. Detienne, “Reasoning from a Schema and from an Analog in Software Code Reuse”, *Workshop on Empirical Studies of Programmers (ESP)*, 1991.

[14] N. Flor and E. Hutchins, “Analyzing Distributed Cognition in Software Teams”, *Workshop on Empirical Studies of Programmers (ESP)*, 1991.

[15] E. Duala-Ekoko and M.P. Robillard, “Tracking Code Clones in Evolving Software”, *ACM SIGSOFT-IEEE International Conference on Software Engineering (ICSE)*, 2007.

[16] B. Fluri, M. Wuersch, M. Pinzger, and H.C. Gall, "Change Distilling: Tree Differencing for Fine-Grained Source Code Change Extraction", *IEEE Transactions on Software Engineering (TSE)*, 2007.

[17] H. Gall, M. Jazayeri, and J. Krajewski, "CVS Release History Data for Detecting Logical Couplings", *ACM SIGSOFT-IEEE International Workshop on Principles of Software Evolution (IWPSE)*, 2003.

[18] M.W. Godfrey and L. Zou, "Using Origin Analysis to Detect Merging and Splitting of Source Code Entities", *IEEE Transactions on Software Engineering (TSE)*, 2005.

[19] T. Green and A. Blackwell, "Cognitive Dimensions of Information Artefacts: A Tutorial", *British Computer Society Conference on Human-Computer Interaction (BCS HCI)*, 1998.

[20] S. Hayashi, M. Saeki, and M. Kurihara, "Supporting Refactoring Activities Using Histories of Program Modification", *The Institute of Electronics, Information and Communication Engineers (IEICE)*, 2006.

[21] S. Horwitz, "Identifying the Semantic and Textual Differences Between Two Versions of a Program", *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 1990.

[22] D. Hou, F. Jacob, and P. Jablonski, "Exploring the Design Space of Proactive Tool Support for Copy-and-Paste Programming", *IBM Conference of the Centre for Advanced Studies on Collaborative Research (CASCON)*, 2009.

[23] J.W. Hunt and M.D. McIlroy, "An Algorithm for Differential File Comparison", *Bell Laboratories*, Bell Laboratories Computing Science Technical Report #41, 1976.

[24] P. Jablonski and D. Hou, "CReN: A Tool for Tracking Copy-and-paste Code Clones and Renaming Identifiers Consistently in the IDE", *Eclipse Technology Exchange Workshop at OOPSLA (ETX)*, 2007.

[25] D. Jackson and D.A. Ladd, "Semantic Diff: A Tool for Summarizing the Effects of Modifications", *IEEE International Conference on Software Maintenance (ICSM)*, 1994.

[26] F. Jacob. "CSeR - A Code Editor for Tracking & Visualizing Detailed Clone Differences", *Clarkson University,* Master's Thesis, 2009.

[27] C.J. Kapser and M.W. Godfrey, "'Cloning Considered Harmful' Considered Harmful: Patterns of Cloning in Software", *Empirical Software Engineering*, Springer, 2008.

[28] M. Kim, L. Bergman, T. Lau, and D. Notkin, "An Ethnographic Study of Copy and Paste Programming Practices in OOPL", *ACM SIGSOFT-IEEE International Symposium on Empirical Software Engineering (ISESE)*, 2004.

[29] M. Kim and D. Notkin, "Discovering and Representing Systematic Code Changes", *ACM SIGSOFT-IEEE International Conference on Software Engineering (ICSE)*, 2009.

[30] M. Kim and D. Notkin, "Program Element Matching for Multi-Version Program Analyses", *ACM SIGSOFT-IEEE International Workshop on Mining Software Repositories (MSR)*, 2006.

[31] M. Kim, D. Notkin, and D. Grossman, "Automatic Inference of Structural Changes for Matching Across Program Versions", *ACM SIGSOFT-IEEE International Conference on Software Engineering (ICSE)*, 2007.

[32] S. Kim, K. Pan, and E.J. Whitehead, Jr., "When Functions Change Their Names: Automatic Detection of Origin Relationships", *IEEE Working Conference on Reverse Engineering (WCRE)*, 2005.

[33] A.J. Ko, H.H. Aung, and B.A. Myers, "Design Requirements for More Flexible Structured Editors from a Study of Programmers' Text Editing", *ACM SIGCHI Conference on Human Factors in Computing Systems (CHI)*, 2005.

[34] V. Kruskal, "Managing Multi-Version Programs with an Editor", *IBM Journal of Research and Development*, 1984.

[35] J. Laski and W. Szermer, "Identification of Program Modifications and its Applications in Software Maintenance", *IEEE International Conference on Software Maintenance (ICSM)*, 1992.

[36] R.C. Miller and B.A. Myers, "Interactive Simultaneous Editing of Multiple Text Regions", *USENIX Annual Technical Conference*, 2001.

[37] H.A. Nguyen, T.T. Nguyen, N.H. Pham, J.M. Al-Kofahi, and T.N. Nguyen, "Accurate and Efficient Structural Characteristic Feature Extraction for Clone Detection", *European Joint Conferences on Theory and Practice of Software (ETAPS) International Conference on Fundamental Approaches to Software Engineering (FASE)*, 2009.

[38] S.P. Reiss, "Tracking Source Locations", *ACM SIGSOFT-IEEE International Conference on Software Engineering (ICSE)*, 2008.

[39] E. L. Rissland, "Examples and Learning Systems", Adaptive Control of Ill-Defined Systems, *Plenum Press*, 1984.

[40] T. Sager, A. Bernstein, M. Pinzger, and C. Kiefer, "Detecting Similar Java Classes Using Tree Algorithms", *ACM SIGSOFT-IEEE International Workshop on Mining Software Repositories (MSR)*, 2006.

[41] M. Toomim, A. Begel, and S.L. Graham, "Managing Duplicated Code with Linked Editing", *IEEE Symposium on Visual Languages and Human-Centric Computing (VLHCC)*, 2004.

[42] Z. Xing and E. Stroulia, "Differencing Logical UML Models", *Automated Software Engineering*, Kluwer Academic Publishers, 2007.

[43] W. Yang, "Identifying Syntactic Differences Between Two Programs", *Software - Practice & Experience*, John Wiley & Sons, 1991.

[44] T. Zimmermann, P. Weissgerber, S. Diehl, and A. Zeller, "Mining Version Histories to Guide Software Changes", *ACM SIGSOFT-IEEE International Conference on Software Engineering (ICSE)*, 2004.