Representing Clones in a Localized Manner

Robert Tairas
AtlanMod (INRIA & École des Mines de Nantes)
44307 Nantes, France
robert.tairas@inria.fr

Ferosh Jacob and Jeff Gray
Department of Computer Science
University of Alabama
Tuscaloosa, AL 35487
fjacob@crimson.ua.edu, gray@cs.ua.edu

ABSTRACT
Code clones (i.e., duplicate sections of code) can be scattered throughout the source files of a program. Manually evaluating a group of such clones requires observing each clone in its original location (i.e., opening each file and finding the source location of each clone), which can be a time-consuming process. As an alternative, this paper introduces a technique that localizes the representation of code clones to provide a summary of the properties of two or more clones in one location. In our approach, the results of a clone detection tool are analyzed in an automated manner to determine the properties (i.e., similarities and differences) of the clones. These properties are visualized directly within the source editor. The localized representation is realized as part of the features of an Eclipse plug-in called CeDAR.

Categories and Subject Descriptors D.2.6 [**Software Engineering**]: Programming Environments – Integrated environments.

General Terms Management, Languages.

Keywords Code clones, visualization, representation.

1. INTRODUCTION
Refactoring tools, including those that are part of an IDE, have been documented to be under-utilized [14]. One of the reasons is the need to configure the refactoring activity through multiple modal dialog boxes that forces a separation between the activities of program editing and the actual refactoring task. That is, the programmer must switch from the activity of editing the source code in a source editor to answering configuration questions in dialog boxes, thus making the source editor unfocused. The refactoring tool Refactor! Pro [4] proposes a solution to reduce the need for dialog boxes by visualizing refactoring changes directly in the source editor.

The principle of keeping most programming activities within the source editor also can be applied to the representation of code clones. Code clones are sections of code that are duplicated in multiple locations in the source files of a program. Often a programmer has to evaluate the clones identified by clone detection tools, by observing directly the actual sections of code that represent the clones. This evaluation can be for program comprehension purposes only or can also include analysis to determine refactoring opportunities. In some instances, the clones can be scattered within a large source file. The clones can also be scattered in several files, such that opening each file containing the clones can clutter the view of the code in an IDE. This suggests a similar situation with the display of dialog boxes in refactoring tools as described in the previous paragraph. This motivates the need to provide a localized representation of the clones that visualizes the properties of each clone in a clone group and the relationships among them.

This paper describes a representation of clone groups that is localized, such that the information about each clone in a group can be viewed in one location. Such representation is not limited to a pair of clones, but can represent multiple clones (i.e., three or more clones) in a clone group. This representation is implemented as part of the features of an Eclipse plug-in called CeDAR (Clone Detection, Analysis, and Refactoring) [17]. The rest of this paper is structured as follows: the next section introduces our representation and the information gathered for the representation. Section 3 shows how the representation is generated and examples of the representation are given. Section 4 evaluates the instances of the localized representation and Section 5 offers a discussion related to the approach. Sections 6 and 7 provide related work, a conclusion and future work.

2. REPRESENTING CLONES IN ONE LOCATION
Our representation of a clone group in a single location provides a summary of the differences and implicit similarities that is not limited to just a pair of clones, but can include more than two clones. The code associated with one of the clones is used as the primary display for the clone group in what we call the "default" clone. Differences among the clones in a group are incorporated within this section of code. The focus of our approach is on clone types II and III (as categorized in [3]).

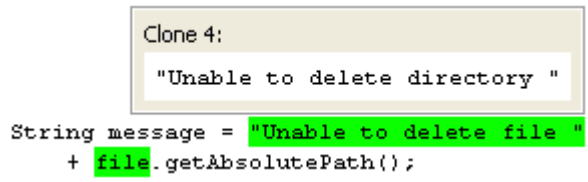


Figure 1. Sample clone group representation

A simple example of the localized display of a clone group in CeDAR can be seen in Figure 1. This example represents a code fragment in a group of four clones found in Apache Ant 1.6.5. The code elements that differ between at least two of the clones in the group are highlighted in neon green (i.e., the string `"Unable to delete file"` and variable `file` in the declaration of variable `message`). When the cursor is placed above one of the highlighted code elements, a pop-up will be displayed that lists

ICSE'11, May 21-28, 2011, Waikiki, Honolulu, HI, USA.

the differing values associated with the code element. In Figure 1, placing the cursor above the `"Unable to delete file"` string spawns a pop-up reporting that Clone 4 consists of a different string (i.e., `"Unable to delete directory"`). This also implies that the remaining clones consist of the same string as the string that is displayed in the figure.

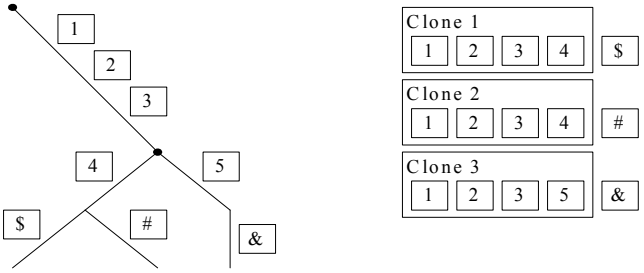


Figure 2. Portion of tree identifying clone similarities

2.1 Detecting clone similarities and differences

The initial step of displaying localized clone group information in CeDAR is to obtain the location of clones. Such information is obtained from clone detection tools, in particular the textual results of the various tools. It should be noted that part of the clone detection process related to these tools includes the determination of differences among the clones. However, in the majority of the tools such difference information is not made available in the final results that are reported to the user. In our approach, analysis of the clones reported by a clone detection tool is performed to determine the parameterized differences among the clones. In effect, CeDAR performs a "second pass" on the code associated with the clones to determine these properties. However, this independent process allows for CeDAR to utilize results from different clone detection tools (i.e., CCFinder [12], CloneDR [2], Deckard [10], Simian[1], and SimScan[2]).

To process the similarities and parameterized differences among the clones in a group, the Abstract Syntax Tree (AST) representation of each clone in the group is obtained. Because CeDAR is an Eclipse plug-in, the Eclipse Java Development Tools (JDT)[3] is used in the process. After the AST of each clone in the group is obtained, comparisons can be made on these ASTs to determine predefined parameterized elements that are recognized. Comparisons are currently performed on the statement-level sub-trees within the ASTs. The suffix tree technique [8] is used to determine the levels of similarities among the clones, which include identifying similarities among the subset of clones in a clone group. Because the clone detection tool has already determined a group of code snippets as clones, the suffix tree in this case is used as a way to identify the similarities and differences within the clones themselves.

To generate the suffix tree, the sequence of elements in which comparisons are performed is the concatenation of all first-level statement nodes from each clone. This is a lower granularity level compared to using each AST node for the sequence, but in this case, the suffix tree is not used to find clones, but rather to determine first-level statement characteristics of the clones that have already been found by a third-party clone detection tool. Furthermore, these similarities and differences are identified for the entire region of the reported clones from the clone detection tool. CeDAR, for example, will not represent duplicate sections of code within a single clone if there were such cases, because it is basing its representation on the original clone reported by the clone detection tool.

Currently in our case, two statements are considered similar if the statements exhibit characteristics such as a Type I or Type II clone. Otherwise, the statements are considered different. For example, in Figure 2, statement 1 of all three clones must either exactly match each other or consist of recognized parameterized elements to be considered as a "match." The next section outlines the currently recognized parameterized elements (or matches).

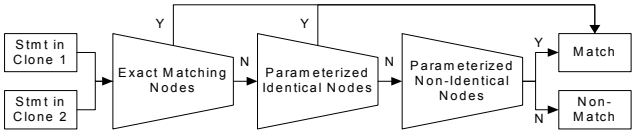


Figure 3. Statement matching levels

2.2 Similarity Levels

When two statement nodes from two clones are compared and are determined to match each other, these two statements share the same branch in the suffix tree. For Type II clones, the matching process is relaxed to allow statements that do not exactly match each other to still be considered similar. In this work, the matching process compares the AST nodes representing the statements. The AST nodes are compared by calling the `subtreeMatch` API call in the JDT, which compares two sub-trees representing the two statements and their children. To identify the parameterized elements between two statements and the differences between two statements, several similarity levels are considered during the matching process. These similarity levels are described in the following paragraphs. Figure 3 outlines the filtering process. The initial matching looks at exact matching nodes. A non-match will compare the nodes for predefined parameterized elements, which is then followed by parameterized elements of non-identical nodes if the nodes still do not match.

Exact matches – This represents a `subtreeMatch` that utilizes the default matcher as provided by the JDT. The sub-trees that are compared must exactly match each other. Otherwise, the sub-trees are considered not equal to each other. The comparison process performs this type of matching initially to determine whether the two sub-trees are exact matches.

Parameterized matches of identical nodes – In this situation, the matcher is customized to allow certain nodes to differ corresponding to the parameterized elements of the clones. The matcher can identify differing values such as variable names and string literals. Currently, the customized `subtreeMatch` in CeDAR allows for differences between two AST nodes that are of type `MethodInvocation`, `NumberLiteral`, `QualifiedName`, `SimpleName`, or `StringLiteral`. This allows clones to have parameterized values for method calls, primitive integers, identifiers separated by the '.' operator, simple identifiers, and strings.

In each case where a sub-tree is "matched," but contains parameterized differences, these differences are stored by

[1] Simian, http://www.redhillconsulting.com.au/products/simian
[2] SimScan, http://blue-edge.bg/simscan
[3] Eclipse JDT, http://www.eclipse.org/jdt

mapping the two differing values. The mapping is between the value of the "default" clone and the corresponding clone with the differing value. The mapping provides two properties regarding the parameterized elements. The first property is the different values between the clones, which will be displayed to the user. The second property is the code element in the default clone that needs to be highlighted to signify that the element represents differing values among the clones.

If an allowed parameterized difference is encountered, then `subtreeMatch` will still report the sub-trees as matching, but will also record the mapping between the parameterized elements. For example, for the code snippet in Figure 1, the `subtreeMatch` comparisons between Clone 1 and Clones 2, 3, and 4 yield the mapping of string `"Unable to delete file"` in Clone 1 to string `"Unable to delete directory"` in Clone 4. In addition, as can be seen in Figure 4, two other mappings are given: variable `file` in Clone 1 to variable `f` in clones 2 and 3; variable `file` in Clone 1 to variable `dir` in Clone 4.

Parameterized matches of near-identical nodes – CeDAR also allows for differences to be considered as matches for combinations of non-identical nodes. The currently allowable combinations consist of nodes listed in the "Parameterized matches of identical nodes" section. For example, a `MethodInvocation` is matched with a `SimpleName` in two AST sub-trees of two clones.

Non-supported parameterized matches / statement differences – This case technically does not represent a similarity, but rather identifies situations when a comparison produces a non-match where two sub-trees are considered not equal to each other. If the matcher can not identify two corresponding nodes in the sub-trees being compared as matching, then the statements represented by the sub-trees are considered not equal. In this case, the two nodes are not exactly equal and do not represent elements that are allowed to be different (i.e., parameterized). The non-matching statement will also be displayed to the user.

```
1:   function DISPLAY(T: suffix tree, S: statement): void
2:     E ← GetEdge(T, S);
3:     if ( E ≠ Ø)
4:       P ← GetParameterizedPairs(E);
5:       PE ← PE ∪ P
6:       if (!RepresentsAllClones(E))
7:         NM ← NM ∪ S
8:       end if
9      else
10:      NM ← NM ∪ S
11:    end if
12:  end function
```

Listing 1. Determining statement similarities

3. LOCALIZED REPRESENTATION IN CEDAR

The localized display of a group of clones is generated when the user selects a clone group in CeDAR. Initially the first clone in the group is set as the default clone. This default clone can be changed to any other clone in the group by the user. When a clone group is selected, the suffix tree process outlined in the previous section is performed. Because the code associated with the default clone will be used to display the localized representation of all the clones in the group, each statement in the default clone needs to be determined whether it is an exact match, a parameterized match, or a non-match with the corresponding statements in the other clones. This is done by evaluating the generated suffix tree based on the statements of the default clone. After the properties of each statement have been determined, the display of the clone group can be done. The following sub-sections describe the process of visualizing the localized representation in CeDAR.

3.1 Obtaining clone similarity properties

The method DISPLAY in Listing 1 outlines the process of determining how the default clone will be displayed in the source editor. For each statement in the default clone, an edge in the suffix tree that represents the statement is obtained using the `GetEdge` method (line 2). The edge must also represent statements from at least one other clone, which signifies that the statement is matched in at least one other clone in the clone group. If no edge is found, then the statement is considered to have no match with the other clones in the group and is added to the list of non-matched statements (*NM* in line 10). If an edge is found, the process looks for mappings or pairings (i.e., by method `GetParameterizedPairs` in line 4) that signify a parameterized element in the statement (i.e., at least two statements of two clones consist of differing values for a specific element). The mappings are stored in the list of parameterized elements (i.e., *PE* in line 5). It may be the case that not all clones match the default clone's statement. This is determined by whether the edge represents all clones in the clone group (i.e., by method `RepresentsAllClone`). If a statement does not match all clones, then it is added to *NM* (line 7).

```
1:   function GETEDGE(T: suffix tree, S: statement): edge
2:     for each E in T do
3:       if (StmtInEdge(E, S))
4:         if (RepresentsMultipleClones(E))
5:           return E;
6:         end if
7:       else
8:         for each B in GetBranches(E) do
9:           return RecursiveStmtInEdge(B);
10:        end for
11:      end if
12:    end for
13:  end function
```

Listing 2. Determining matching statements

The GETEDGE method in Listing 2 (i.e., used in Line 2 in the DISPLAY method) searches the suffix tree to find an edge containing the statement of the default clone. The process recursively evaluates each edge of the tree starting from the top-most edges. The method `StmtInEdge` looks to see if the given statement is part of the edge that is being evaluated (line 3). If not, then the branches of that edge are recursively evaluated (i.e., by the method `RecursiveStmtInEdge` in line 9). If the edge contains the statement, then it is evaluated to determine whether it also represents two or more clones (i.e., by the method `RepresentsMultipleClones` in line 4). This is determined by looking at the branches of the edge to see if at least two branches represent sequences of statements of two separate clones. For

example, in Figure 2, the top-most edge represents the first three statements of all three clones, because the edge has branches that represent the three clones, which in this case are the branches for the three special terminating characters (i.e., '$,' '#,' and '&').

During the suffix tree generation process in this application, when a statement in an edge matches a new statement being compared, the new statement is also stored in the edge. For example, in Figure 2, the branch with statement 4 will contain the statement nodes from both Clone 1 and Clone 2. The determination of whether an edge represents at least two clones can be done by evaluating the matching statements that are stored in the edge.

3.2 Displaying clones in one location

The examples in this section represent clones found in Apache Ant 1.6.5. In CeDAR, the localized representation is visualized directly in the source editor. An example can be seen in Figure 4 where the section of the default clone, which in this case is Clone 1, is highlighted in light blue and bordered by two horizontal lines in the figure. The sections of all other clones in the same file are highlighted in light grey, which is not shown in the figure. As stated at the beginning of Section 3, the representation is activated when a clone group is selected, hence only a single clone group is represented per user request. When a user selects another clone group, the representation is reset for the new clone group.

For all parameterized elements in *PE*, the corresponding code element in the default clone is highlighted. These elements are highlighted in neon green. Hovering the mouse over one of these elements will invoke a pop-up containing the parameterized differences associated with that code element. In Figure 4, the parameterized differences of the `file` variable are visualized in the pop-up. In this case, clones 2 and 3 use a different variable name (i.e., `f`), while clone 4 uses the variable name `dir`.

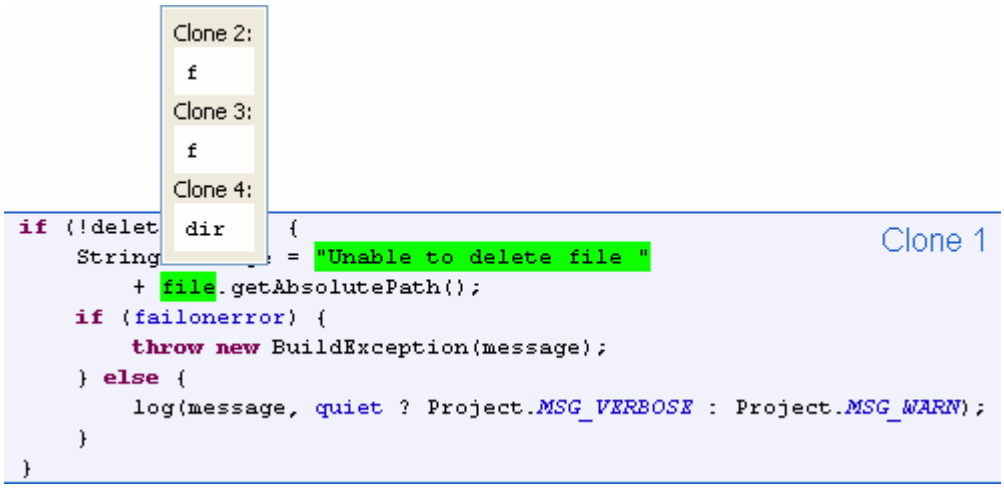
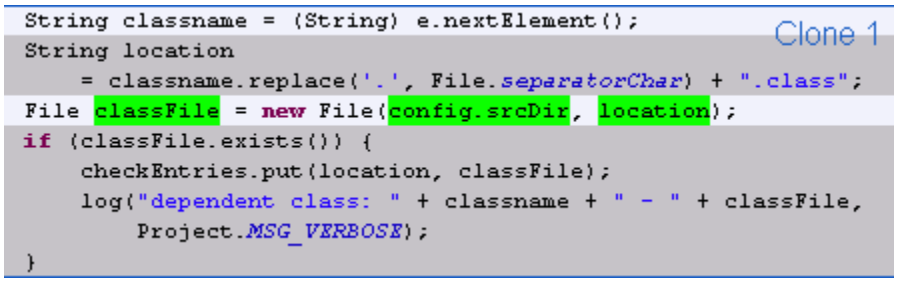


Figure 4. Pop-up of simple variable differences



(a) Clone 1

```
String classname = (String) e.nextElement();
String filename = classname.replace('.', File.separatorChar);
filename = filename + ".class";
File depFile = new File(basedir, filename);
if (depFile.exists() && parentSet.containsKey(filename)) {
    // This is included
    included.addElement(filename);
}
```

(b) Clone 2

Figure 5. Highlighted statement differences

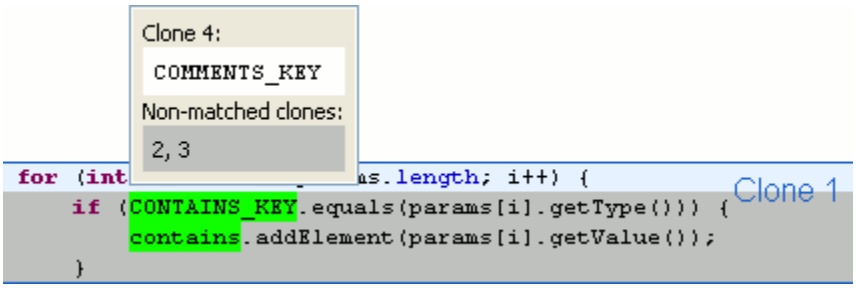


Figure 6. Display of two sub-groups of parameterized clones

For all non-matching statements in *NM*, the corresponding statement in the default clone is highlighted. In CeDAR, these statements are highlighted in dark grey. Figure 5 provides an example of the highlighting of statements that are not equal. In Figure 5 (a), the second statement is highlighted (i.e., in dark grey). The differences can be seen as compared to the corresponding clone in Figure 5 (b), where the second and third statements in Clone 2 perform the task that was only done by one statement in Clone 1 (i.e., the second statement). In addition, the differences between the fourth statement in Clone 1 and the fifth or last statement in Clone 2 are more profound. The only exact matching statement is the first statement. The third statement in Clone 1 is a parameterized match with its corresponding statement in Clone 2, where it consists of parameterized elements of identical node types (i.e, variables `classFile` – `depFile` and `location` – `filename`). A non-identical node type match is also evident (i.e., `config.srcDir` – `baseDir`). This example demonstrates the various display properties for the clones, starting from exact matching statements to non-matching statements.

Figure 6 is an example scenario where the display of parameterized elements is within the display of a non-matching statement. The four clones in the clone group associated with this display consist of two separate sub-groups where each sub-group represents a tighter similarity. In this case, clones 1 and 4 are more similar to each other than with clones 2 and 3, and vice versa. The parameterized constant `COMMENTS_KEY` in Clone 4 is visualized in the pop-up as the only clone with a parameterized difference (i.e., with the constant `CONTAINS_KEY` in Clone 1). The statement is not equal for the remaining clones; hence, clones 2 and 3 are also listed in the pop-up as non-matches.

4. EVALUATION

The instances where the localized representations can visualize parameterized elements or statement differences were evaluated on multiple open source software artifacts. Parameterized elements represent differences that are acknowledged by CeDAR, such as variable name differences. Statement differences represent statements that differ syntactically or those that contain parameterized elements that are not currently supported by CeDAR (i.e., AST nodes not listed in Section 2.2). The clones were detected using Deckard, a tree-based clone detection tool that reports syntactically meaningful clones. In this case, the reported clones represent clearly separated statements. The similarity value was set to 0.95 to allow non-exact clones, including parameterized clones. The evaluation considers the number of clone groups that can be represented appropriately by the localized representation in CeDAR.

Four different scenarios were considered. The first scenario is when a clone group consisted of exact matching clones (i.e., Type I clones). In this case, the localized representation will not show any differences among the clones. The second scenario is when a clone group consists only of recognized parameterized elements, which are listed in Section 2.2. The third scenario is when a clone group consists of only statement differences. The final scenario is when a clone group consists of both recognized parameterized elements and statement differences (e.g., as seen in Figure 5).

Table 1. Clone types identified by CeDAR in various open source programs

Program	#CG	Exact (%)	Param (%)	StmtDiff (%)	Mixed (%)
Apache Ant 1.6.5	429	61 (14%)	152 (35%)	131 (31%)	85 (20%)
ArgoUML 0.26	650	61 (9%)	214 (33%)	124 (19%)	251 (39%)
Jakarta-JMeter 2.3.2	377	77 (20%)	158 (42%)	71 (19%)	71 (19%)
JBoss AOP 2.1.5	159	51 (32%)	81 (51%)	14 (9%)	13 (8%)
JFreeChart 1.0.10	847	151 (18%)	415 (49%)	168 (20%)	113 (13%)
JRuby 1.4.0	318	113 (36%)	70 (22%)	63 (20%)	72 (23%)
EMF 2.4.1	285	54 (19%)	136 (48%)	52 (18%)	42 (15%)
JEdit 4.2	345	91 (26%)	120 (35%)	88 (26%)	46 (13%)
Squirrel-SQL 3.0.3	428	78 (18%)	164 (38%)	70 (16%)	116 (27%)

#CG = Total clone groups
Exact = Clone groups with exactly matching clones
Param = Clone groups with parameterized clones
StmtDiff = Clone groups with non-supported parameterized clones or near-exact clones
Mixed = Clone groups consisting of both "Param" and "StmtDiff" instances

Instances of the first two scenarios can be fully represented by CeDAR. When the clones are exactly the same, then no annotations are needed. When the clones only contain recognized parameterized elements, then the representation of the clones can be summarized accordingly within the source editor. The last two scenarios represent instances where CeDAR cannot fully identify the differences of the clones. This is especially the case for the non-matched statements, as these non-matches can signify several properties related to the differences of the clones. In these cases, CeDAR currently reports that the statements do not match, but does not provide information about the reason for the non-match.

Table 1 documents the instances after running a batch process that looked at each clone group to determine what type of scenario is related to the clone group. The first scenario ("Exact" column) occurred considerably. The second scenario ("Param" column) occurred in the majority of cases except for ArgoUML and JRuby. In fact, in four of the artifacts (i.e., Jakarta-JMeter, JBoss-AOP, JFreeChart, and EMF) this scenario occurred in almost half of all the instances. The third ("StmtDiff" column) and fourth ("Mixed" column) scenarios consist of statement differences. Only in ArgoUML did the number of instances containing statements that could not be matched by CeDAR (i.e., "Mixed" column) exceed the number of instances of parameterized elements that are currently recognized (i.e., "Param" column).

The evaluation from Table 1 suggests that in the majority of the cases, the clone groups consisted of parameterized elements that are currently recognized and supported by CeDAR. For the cases where statements could not be matched, there is a possibility of eliminating these non-matches by supporting more parameterized elements. Further evaluation of these instances can help determine the additional parameterized elements that can be included for the localized representation.

5. DISCUSSION

This section describes some points for consideration related to the representation process described in the previous sections.

Initial clone detection tool process – As stated in Section 2, the process described in this paper is considered a "second pass" on the code associated with the clones that were initially reported by a clone detection tool. Several tools (e.g., [6][12][16]) have utilized the suffix tree technique to find clones. However, in the process used by these tools, the actual values of the identifiers are ignored as long as the identifier token or node matches. In some cases, the actual identifier values are only evaluated during a post-processing stage to determine the type of the clone (i.e., exact or parameterized). Incorporating the identification of the parameterized elements that can be used in the display of the clones within the clone detection process itself can remove the need for a second pass on the code. An alternative is to perform post-processing on the clone detection results directly within the tool rather than working with the textual output of the tool. However, this removes the flexibility of utilizing the technique on different clone detection tools.

Version clones – The existence of multiple versions of a class in a version control system can give rise to what we call "version clones." Related work on visualizations of code changes based on version history is described in Section 6. The focus of the work described in this paper is on "snapshot clones," which we define as clones detected by a clone detection tool in a single snapshot of the source code. A question that arises is what are the characteristics of version clones as compared to snapshot clones? If we look at the techniques associated with version clones, they focus mainly on changes on a class as a whole and the differences between two versions of the class. In contrast, for the work described in this paper, clones can vary from statement-level, method-level, and also class-level clones. In addition, the CeDAR approach allows for comparison of two or more clones. An interesting exercise would be to extend the techniques for version clones to support snapshot clones, and vice versa.

Limitations – A particular limitation of the technique described in this paper is that it generates a suffix tree on the first-level statements associated with the detected clones. This presents a fixed granularity level for the representation of the clones. The disadvantage in this case can be seen when the statement consists of multiple nested levels. When the first-level statements cannot be matched, then the entire statement is displayed as a non-match even though the difference may only reside several levels below the top level statement.

A further limitation is seen in the results of the evaluation in Section 5, specifically related to the display of non-matched statements. The reason for non-matches may be due to unsupported parameterized elements. Increasing the support for additional parameterized elements can potentially reduce the number of non-matched statements that are displayed.

6. RELATED WORK

This section outlines related work on localized clone representation and other clone visualization techniques.

Localized representation – CloneDR provides a similar functionality called "Clone Abstraction," which lists the parameterized elements associated to a group of clones. The HTML reports generated by CloneDR include a section that presents in textual format a single representation for a clone that identifies the parameterized elements. More recently, the (COBOL) CloneDR[4] tool displays parameterized elements within the Rational Development for System Z (RDz) Environment. The localized representations in CeDAR consist of additional capabilities, which include the display of the varying values of the parameterized elements directly in the source editor and the display of non-matching statements among the clones.

Clone visualizations – Several techniques to visualize clones have been proposed, but these mostly visualize clones in a system-wide level, whereas the work presented in this paper visualizes the similarities and differences of clones at the source code level. The use of scatter plots is a popular clone visualization mechanism and is included in several clone detection tools such as [12]. In addition to scatter plots, duplication web, duplication aggregation tree map, and system model views are introduced in [15]. Clone information is included in a visualization tool of software architectures and their dependencies [13]. A clone system hierarchy graph was proposed by Jiang and Hassan [11].

Code version visualizations – Various efforts have been made to visualize the differences between versions of code. The Eclipse Compare editor uses a tokenized version of the Unix *diff*

[4] (COBOL) CloneDR, http://www.semdesigns.com/Products/Clone/COBOLCloneDR.html

command to show differences between two code segments. Version Editor (VE) [1] provides tighter integration between editors and version control systems to show the changes of a file with a source code repository. However, it uses a text-based algorithm, which cannot distinguish between comments and code. CSeR (Code Segment Reuse) uses an AST-based algorithm to keep track and visualize copy-paste induced changes [9]. Compared to CeDAR, CSeR is limited to copy-paste induced clones and support for clone groups (more than two clones) is also limited. SolidSDD[5] shows differences between clones that it detects, but similar to CSeR, the differences are limited to a pair of clones. Breakaway [5] introduces an approach to find correspondences between two code segments, but the representation is textual. Change distiller [7] extracts changes from a repository and visualizes the changes using the Compare editor in Eclipse. The changes are extracted from hierarchically structured data, but are limited to class-level changes. Furthermore, comparisons are performed only on multiple versions of the same class.

7. CONCLUSION AND FUTURE WORK

The clone group representation described in this paper allows the visualization of clone properties for all the clones in a clone group on just one clone instance. The representation of clones in a clone group in one location provides a quick summary of the properties of the clones and allows the programmer to learn about the clones without the need to open every occurrence of each clone in the source files. Although an evaluation of several software artifacts demonstrates the majority of clone groups can be displayed appropriately by CeDAR, the main future work to be considered is the inclusion of more parameterized elements to reduce the number of non-matched statements. In addition, an evaluation will be considered to determine to what extent this representation can be utilized without becoming just a cluttered and hence less useful representation.

8. ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under Grant No. CCF-0702764.

9. REFERENCES

[1] Atkins, D. L. 1998. Version Sensitive Editing: Change History as a Programming Tool. In *Proceedings of the Scm-8 Symposium on System Configuration Management* (Brussels, Belgium, Jul. 1998), pp. 146-157.

[2] Baxter, I. D., Yahin, A., Moura, L., Sant'Anna, M., and Bier, L. 1998. Clone Detection Using Abstract Syntax Trees. In *Proceedings of the International Conference on Software Maintenance* (Bethesda, MD, Mar. 1998), pp. 368-377.

[3] Bellon, S., Koschke, R., Antoniol, G., Krinke, J., and Merlo, E. 2007. Comparison and Evaluation of Clone Detection Tools. *IEEE Transactions on Software Engineering*. 33, 9 (Sep. 2007), pp. 577-591.

[4] Campbell, D. and Miller, M. 2008. Designing Refactoring Tools for Developers. In *Workshop on Refactoring Tools* (Nashville, TN, Oct. 2008), Article 9, 2 pages.

[5] SolidSDD, http://www.solidsourceit.com/products/SolidSDD-code-duplication-cloning-analysis.html

[5] Cottrell, R., Chang, J. J., Walker, R. J., and Denzinger, J. 2007. Determining Detailed Structural Correspondence for generalization tasks. In *Proceedings of the Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering* (Dubrovnik, Croatia, Sep. 2007), pp. 165-174.

[6] Falke, R., Frenzel, P., and Koschke, R. 2008. Empirical Evaluation of Clone Detection using Syntax Suffix Trees. *Empirical Software Engineering*. 13, 6 (Dec. 2008), pp. 601-643.

[7] Fluri, B., Wuersch, M., PInzger, M., and Gall, H. 2007. Change Distilling: Tree Differencing for Fine-Grained Source Code Change Extraction. *IEEE Transactions on Software Engineering*. 33, 11 (Nov. 2007), pp. 725-743.

[8] Gusfield, D. 1997 *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. Cambridge University Press, Cambridge, United Kingdom.

[9] Jacob, F., Hou, D., and Jablonski, P. Actively Comparing Clones Inside The Code Editor. In *Proceedings of the International Workshop on Software Clones* (Cape Town, South Africa, May 2010), pp. 9-16.

[10] Jiang, L., Misherghi, G., Su, Z., and Glondu, S. 2007. DECKARD: Scalable and Accurate Tree-Based Detection of Code Clones. In *Proceedings of the International Conference on Software Engineering* (Minneapolis, MN, May 2007), pp. 96-105.

[11] Jiang, Z. M. and Hassan, A. E. 2007. A Framework for Studying Clones In Large Software Systems. In *Proceedings of the International Working Conference on Source Code Analysis and Manipulation* (Paris, France, Sep. 2007), pp. 203-212.

[12] Kamiya, T., Kusumoto, S., and Inoue, K. 2002. CCFinder: a Multilinguistic Token-based Code Clone Detection System for Large Scale Source Code. *IEEE Transactions on Software Engineering*. 28, 7 (Jul. 2002), pp. 654-670.

[13] Kapser, C. and Godfrey, M. W. 2005. Improved Tool Support for the Investigation of Duplication in Software. In *Proceedings of the International Conference on Software Maintenance* (Budapest, Hungary, Sep. 2005), pp. 305-314.

[14] Murphy-Hill, E. and Black, A. P. 2008. Refactoring Tools: Fitness for Purpose. *IEEE Software*. 25, 5 (Sep. 2008), pp. 38-44.

[15] Rieger, M., Ducasse, S., and Lanza, M. 2004. Insights into System-Wide Code Duplication. In *Proceedings of the Working Conference on Reverse Engineering* (Delft, The Netherlands, Nov. 2004), pp. 100-109.

[16] Tairas, R. and Gray, J. 2006. Phoenix-based Clone Detection Using Suffix Trees. In *Proceedings of the ACM Southeast Regional Conference* (Melbourne, FL, Mar. 2006), pp. 679-684.

[17] Tairas, R. and Gray, J. 2009. Get to Know Your Clones with CeDAR. In *Companion to the International Conference on Object-Oriented Programming Systems Languages and Applications* (Orlando, FL, Oct. 2009), pp. 817-818.