CReN[*]: A Tool for Tracking Copy-and-Paste Code Clones and Renaming Identifiers Consistently in the IDE

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

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

ABSTRACT

Programmers often copy and paste code so that they can reuse the existing code to complete a similar task. Many times, modifications to the newly pasted code include renaming all instances of an identifier, such as a variable name, consistently throughout the fragment. When these modifications are done manually, undetected inconsistencies and errors can result in the code, for example, a single instance can be missed and mistakenly not renamed. To help programmers avoid making this type of copy-paste error, we created a tool, named CReN, to provide tracking and identifier renaming support within copy-and-paste clones in an integrated development environment (IDE). CReN tracks the code clones involved when copying and pasting occurs in the IDE and infers a set of rules based on the relationships between the identifiers in these code fragments. These rules capture the programmer's intentions, for example, that a particular group of identifiers should be renamed consistently together. Programmers can also provide feedback to improve the accuracy of the inferred rules by specifying that a particular instance of an identifier is to be renamed separately. We introduce our CReN tool, which is implemented as an Eclipse plug-in in Java.

Categories and Subject Descriptors

D.1.5 [**Programming Techniques**]: Object-oriented Programming. D.2.3 [**Software Engineering**]: Coding Tools and Techniques – *Object-oriented programming.* D.2.6 [**Software Engineering**]: Programming Environments – *Integrated environments.* D.3.2 [**Programming Languages**]: Language Classifications – *Java, Object-oriented languages.*

General Terms

Verification, Reliability, Languages, Human Factors.

[*] http://www.clarkson.edu/~dhou/projects/CReN

Keywords

Abstract syntax tree, code clone, consistent renaming, copy-and-paste programming, Eclipse integrated development environment, error detection, intent inference, Java.

1. INTRODUCTION

One reason why programmers copy and paste code is to reuse existing code as a template. In this type of copy-and-paste programming, the programmer sees a similarity in the software solution of a previous task and the current task, and intends to modify the newly pasted code accordingly. A common modification to the new code is to rename all instances of an identifier, such as a variable name, consistently throughout the fragment. We refer to such changes as the consistent renaming usage pattern.

When programmers manually modify a code fragment to rename all of an identifier's instances, for example, they may miss an instance, resulting in an inconsistency or error that may go undetected by the compiler and themselves. We consider any unintended inconsistent renaming of identifiers as an error.

Copying and pasting code results in code clones, which are similar code fragments (an average of 10 lines of code) that are repeated throughout the larger source code. Programmers cannot always remove and replace these clones with a procedure, sometimes making clones unavoidable. Other times, for example, with code fragments that are less than 10 lines of code, it may not be practical to extract them into procedures.

There are examples from literature that show an inconsistent renaming of identifiers within a copy-and-pasted clone in production code. Three examples are shown in Figure 1.

The first example in Figure 1, published in a paper by Li, et al., [3] is from the file memory.c in Linux version 2.6.6. The original code fragment (on the left) is a for loop that is copied and pasted and then modified. In the modified pasted code fragment (on the right), the programmer intended to change all instances of the array name "prom_phys_total" to "prom_prom_taken". The programmer unintentionally did not change one instance of the array's name (in the last line). The compiler did not detect this error because "prom_phys_total" is still in scope. In this example, the for loop was copied and pasted within the same function: void __init prom_meminit(void), which begins at line 68 in memory.c (not shown).

The second example that is shown in Figure 1, from a paper by Liblit, et al., [4] is code that is part of the GNU command “bc”, in the file storage.c. The original copied code fragment (on the left) is a function named “more_variables” that allocates a larger amount of memory for the “variables” array. It then copies the values over from the smaller array “old_var” to the larger array “variables” (in the first for loop), and then fills in the rest of the space in the “variables” array with NULL. In the modified pasted code fragment (on the right), the function’s name was renamed from “more_variables” to “more_arrays”, the type “bc_var” was renamed to “bc_var_array”, and all instances of the arrays “old_var”, “variables”, and “v_names” were renamed to “old_ary”, “arrays”, and “a_names”, respectively. However, one instance of the variable “v_count” in this function was missed and not renamed to “a_count” (in the second for loop’s condition). Because “v_count” is defined as a global variable, this copy-paste error is not detected by the compiler.

Figure 1’s third example is from a paper by Jiang, et al., [2] and is code in the file dependency.c from the GCC Fortran compiler. In this example, the identifier “l_stride” in the if statement’s condition is also used in the if statement’s body. However, in the modified code fragment, the “r_stride” identifier was supposed to be left as “l_stride”. This is a different type of error than the other two, but is still an inconsistency in renaming that was not caught by the compiler or the programmer during development.

All of the examples presented here contain inconsistent renaming errors that were found in existing production source code (there are also many more examples of this in practice). We hope to prevent this type of error from occurring at all, by catching it during program development. This should be more cost effective than detecting and fixing inconsistent renaming errors after they have happened. Existing tools typically involve computationally expensive, sophisticated algorithms, like statistical bug isolation [4], or running a clone detection tool followed by a number of error detection and pruning algorithms, which still results in many false positives [2, 3]. However, we believe that our consistent renaming tool (CReN) complements existing error detection tools, which are still needed to find potential errors in legacy code.

	The Original Copied Code Fragment	**The Modified Pasted Code Fragment (Buggy)**
1	File: linux-2.6.6/arch/sparc64/prom/memory.c (lines 92-99)	File: linux-2.6.6/arch/sparc64/prom/memory.c (lines 111-118)

1 — The Original Copied Code Fragment:

```
for(iter=0; iter<num_regs; iter++){
  prom_phys_total[iter].start_adr =
    prom_reg_memlist[iter].phys_addr;
  prom_phys_total[iter].num_bytes =
    prom_reg_memlist[iter].reg_size;
  prom_phys_total[iter].theres_more =
    &prom_phys_total[iter+1];
}
```

1 — The Modified Pasted Code Fragment (Buggy):

```
for(iter=0; iter<num_regs; iter++){
  prom_prom_taken[iter].start_adr =
    prom_reg_memlist[iter].phys_addr;
  prom_prom_taken[iter].num_bytes =
    prom_reg_memlist[iter].reg_size;
  prom_prom_taken[iter].theres_more =
    &prom_phys_total[iter+1];  //error
}
```

	The Original Copied Code Fragment	**The Modified Pasted Code Fragment (Buggy)**
2	File: bc-1.06/bc/storage.c (lines 118-150)	File: bc-1.06/bc/storage.c (lines 152-185)

2 — The Original Copied Code Fragment:

```
void
more_variables ()
{
  int indx;
  int old_count;
  bc_var **old_var;
  char **old_names;

  /* Save the old values. */
  old_count = v_count;
  old_var = variables;
  old_names = v_names;

  /* Increment by a fixed amount and allocat...
  v_count += STORE_INCR;
  variables = (bc_var **) bc_malloc (v_count...
  v_names = (char **) bc_malloc (v_count*siz...

  /* Copy the old variables. */
  for (indx = 3; indx < old_count; indx++)
    variables[indx] = old_var[indx];

  /* Initialize the new elements. */
  for (; indx < v_count; indx++)
    variables[indx] = NULL;

  ...
}
```

2 — The Modified Pasted Code Fragment (Buggy):

```
void
more_arrays ()
{
  int indx;
  int old_count;
  bc_var_array **old_ary;
  char **old_names;

  /* Save the old values. */
  old_count = a_count;
  old_ary = arrays;
  old_names = a_names;

  /* Increment by a fixed amount and allocat...
  a_count += STORE_INCR;
  arrays = (bc_var_array **) bc_malloc (a_co...
  a_names = (char **) bc_malloc (a_count*siz...

  /* Copy the old arrays. */
  for (indx = 1; indx < old_count; indx++)
    arrays[indx] = old_ary[indx];


  /* Initialize the new elements. */
  for (; indx < v_count; indx++)  //error
    arrays[indx] = NULL;

  ...
}
```

	The Original Copied Code Fragment	**The Modified Pasted Code Fragment (Buggy)**
3	File: gcc-4.0.1/gcc/fortran/dependency.c (lines 414-415)	File: gcc-4.0.1/gcc/fortran/dependency.c (lines 422-423)

3 — The Original Copied Code Fragment:

```
if (l_stride != NULL)
  mpz_cdiv_q (X1, X1, l_stride->value.integer);
```

3 — The Modified Pasted Code Fragment (Buggy):

```
if (l_stride != NULL)
  mpz_cdiv_q (X2, X2, r_stride->val...  //error
```

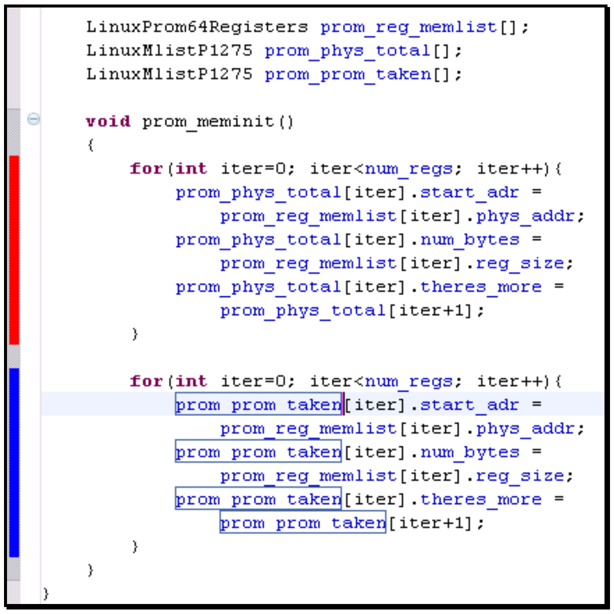
Figure 1. Three examples from literature that show an inconsistent renaming of identifiers in the pasted code fragment.

2. USAGE SCENARIOS

In this section, we demonstrate how CReN would catch each of the identifier renaming errors in the three examples from Figure 1 in the scenario that each of these programs is currently being written in the IDE. The examples have been rewritten in Java.

In each example, when the original code fragment is copied and pasted, CReN will group identifiers within a code fragment and map pairs of identifiers that are at the same location in the copied and the pasted code fragments. When the code is initially pasted, the pasted fragment is identical to the original. This makes the identifier mapping possible, with modifications being tracked as they happen.

In the first example in Figure 1, the for loop is copied and pasted from lines 92-99 to lines 111-118 in the memory.c file. CReN detects this and, with support from the ASTs, extracts a rule stating that all occurrences of the identifier "prom_phys_total" in lines 93-98 should be changed to the same identifier in the new copy. With this rule, when the programmer changes any instance of "prom_phys_total" in the pasted code fragment to "prom_prom_taken" all of the other instances (in the group) will also be renamed to "prom_prom_taken" consistently, as shown in Figure 2. Hence, CReN will be able to prevent the missed renaming shown in the first cell of the buggy column in Figure 1.



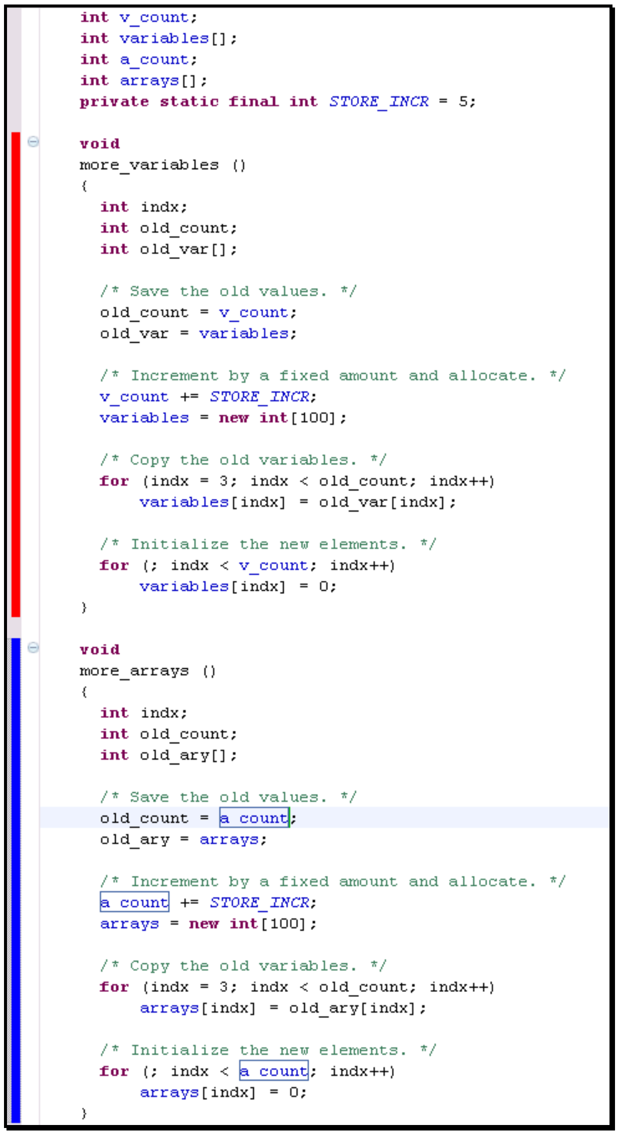
```
LinuxProm64Registers prom_reg_memlist[];
LinuxMlistP1275 prom_phys_total[];
LinuxMlistP1275 prom_prom_taken[];

void prom_meminit()
{
    for(int iter=0; iter<num_regs; iter++){
        prom_phys_total[iter].start_adr =
            prom_reg_memlist[iter].phys_addr;
        prom_phys_total[iter].num_bytes =
            prom_reg_memlist[iter].reg_size;
        prom_phys_total[iter].theres_more =
            prom_phys_total[iter+1];
    }

    for(int iter=0; iter<num_regs; iter++){
        prom_prom_taken[iter].start_adr =
            prom_reg_memlist[iter].phys_addr;
        prom_prom_taken[iter].num_bytes =
            prom_reg_memlist[iter].reg_size;
        prom_prom_taken[iter].theres_more =
            prom_prom_taken[iter+1];
    }
}
}
```

Figure 2. CReN consistently renames all instances of "prom_phys_total" to "prom_prom_taken" in the fragment when any one instance of "prom_phys_total" in the fragment is modified.

In the second example in Figure 1, the entire function is copied and pasted from lines 118-150 to lines 152-184 in the file storage.c. CReN detects the copying and pasting and, from the ASTs, extracts a rule that states that all occurrences of the identifier "v_count" in lines 118-150 should be changed to the same identifier in the new copy. With this rule, when the programmer changes any instance of "v_count" in the pasted code fragment to "a_count" all of the other instances (in the group) will also be renamed to "a_count" consistently, as shown in Figure 3. CReN will be able to prevent the missed renaming that is in the second for loop shown in this example.



```
int v_count;
int variables[];
int a_count;
int arrays[];
private static final int STORE_INCR = 5;

void
more_variables ()
{
  int indx;
  int old_count;
  int old_var[];

  /* Save the old values. */
  old_count = v_count;
  old_var = variables;

  /* Increment by a fixed amount and allocate. */
  v_count += STORE_INCR;
  variables = new int[100];

  /* Copy the old variables. */
  for (indx = 3; indx < old_count; indx++)
      variables[indx] = old_var[indx];

  /* Initialize the new elements. */
  for (; indx < v_count; indx++)
      variables[indx] = 0;
}

void
more_arrays ()
{
  int indx;
  int old_count;
  int old_ary[];

  /* Save the old values. */
  old_count = a_count;
  old_ary = arrays;

  /* Increment by a fixed amount and allocate. */
  a_count += STORE_INCR;
  arrays = new int[100];

  /* Copy the old variables. */
  for (indx = 3; indx < old_count; indx++)
      arrays[indx] = old_ary[indx];

  /* Initialize the new elements. */
  for (; indx < a_count; indx++)
      arrays[indx] = 0;
}
```

Figure 3. CReN consistently renames all instances of "v_count" to "a_count" in the fragment when any one instance of "v_count" in the fragment is modified.

The third example in Figure 1 is different from the other two. In this example, an if statement was copied and pasted from lines 414-415 to lines 422-423 in the dependency.c file. CReN detects the copying and pasting and, from the ASTs, extracts a rule that states that all occurrences of the identifier "l_stride" in lines 414-415 should be changed to the same identifier in the new copy. With this rule, when the programmer changes any instance of "l_stride" (for example, the bottom "l_stride") in the pasted code fragment to "r_stride", all of the other instances (in the group, for example, the top "l_stride") will also be renamed to "r_stride" consistently. However, according to Jiang, et al., [2] while the GCC developers confirmed that the inconsistency (one "l_stride" and one "r_stride") is a bug, it is not for this reason. In fact, the programmers intended to not rename either of the instances of “l_stride” in this clone at all. We don’t focus on this case exactly, since we expect the pasted code to be modified (we consider the type of copy-and-paste where code is reused as a template as opposed to exact duplication), but CReN would still be able to alert the programmer of the inconsistency. When the other instance of “l_stride” is being renamed to “r_stride”, programmers should then realize that they didn’t intend to make either modification. (This is still different from the case when the programmer intends to rename an instance of an identifier independently from the others. We directly provide the functionality in CReN for the programmer to be able to remove an instance of an identifier from a group that is to be renamed consistently together).

3. THE CReN TOOL

Our consistent renaming (CReN) tool is an Eclipse plug-in written in and for Java. Once installed, CReN starts automatically listening to document activity in the Eclipse IDE’s editor. To capture information about the source code, CReN uses the AST API of the Eclipse JDT framework. Abstract syntax trees (ASTs) allow CReN to establish relationships of the copy-and-pasted code, and infer knowledge about consistent renaming. Such knowledge is then used to help programmers consistently rename identifiers in the clone fragments. This tool also interacts with the programmer to incrementally refine the inferred knowledge, ensuring that the knowledge matches the programmer’s intentions. Conceptually, the CReN tool consists of two parts: tracking copy-and-paste clones and performing the consistent identifier renaming.

CReN automatically tracks the clones involved when the copying and pasting operations happen in the IDE. Because of this, no clone detection tool or manual selection of clones is needed. The CReN tool keeps track of and continuously updates the related clones’ locations and all identifiers’ locations within each clone. Specifically, CReN represents a clone region by the Java file name where the clone is located, and its range location in the file (offset and length). A clone group, which in general can contain two or more members, is also tracked. With support from Eclipse, clone and identifier locations can be updated automatically when edits happen to the Java files that contain the clones. On the user interface, CReN highlights the statements of the copy-and-paste clones with a bar on the left-hand side of the editor pane (red for the original code and blue for the copies) so that programmers can visualize and manage their copy-and-paste activity in the IDE, which could help avoid errors and navigate a clone group.

CReN automatically renames identifiers within a clone when any identifier in the defined group is renamed by the programmer. Identifiers are put into the same group if they share the same binding, or the same name when bindings are not available. The programmer can also provide feedback to CReN, eliminating a specific identifier from the group so that it can be modified individually. This new rule will be automatically applied to all members of the same clone group. Data about clones and the inferred rules are persisted between sessions.

4. RELATED WORK

Some features in Eclipse can help with consistent renaming. There are also some published tools that manage and track code clones and a few other tools that focus on copy-paste error detection in the context of traditional clone detection.

4.1 Related Eclipse Features

The Find & Replace, Refactoring (Rename), and Linked Renaming features in Eclipse can assist a programmer with consistently renaming identifiers in the IDE. Each has its own set of limitations and differences from CReN.

Find & Replace in Eclipse allows the programmer to find specified text and replace it with another text. Find & Replace is simply a text-based search and has no knowledge of the structure of the program. It does not infer intent and must be initially requested by the programmer. In addition, Find & Replace is not limited to within a clone code fragment, so the programmer must know where renaming in the clone begins and ends and manually replace only those instances.

The Rename refactoring allows the programmer to rename various program elements. As such, binding is an important condition for it to work, which is not necessary for CReN. Furthermore, Rename is automatically applied to the whole project instead of a clone.

Linked Renaming allows the programmer to rename identifiers within a file scope. The Rename refactoring applies to the whole project instead. Furthermore, Linked Renaming neither works with code that does not type check nor renames identifiers only within a clone as CReN does.

4.2 Clone Tracking Tools

There are some published tools (Codelink [5] and CloneTracker [1]) that focus on managing and tracking code clones to help programmers make more consistent code modifications among them. Both tools require manual clone selection first. Both can help keep modifications consistent *between* code fragments, for example, if there is a common modification that is needed between all related clones. However, these tools do not infer change rules *within* a code fragment, which is where we define the consistent renaming usage pattern.

Codelink, which is an extension to XEmacs developed by Toomim, et al., implements the concept of Linked Editing [5]. In the Codelink editor, the programmer has to manually select the clones in order to link them. Once the code fragments are linked, modifications made in one clone can be made to all of the others that it is linked to simultaneously, or edits can be made to a single clone individually.

More recently, Duala-Ekoko, et al., presented a clone tracking system named CloneTracker, which they implemented as an Eclipse plug-in using ASTs [1]. They introduced the concept of Clone Region Descriptors (CRDs) and created a new method of Simultaneous Editing. CloneTracker relies on the output of the SimScan clone detection tool and requires the programmer to manually select the clone groups of interest to be documented. Once the clone groups are identified, CloneTracker translates the location of all clone regions from a file name and line range notation into CRDs. Instead of using the clone’s exact text or its physical location in the file, the CRD technique uses syntactic, structural, and lexical information (the clone region's alignment with code blocks) to determine its relative location in a file. While this technique has some benefits, it only gives an approximate location.

4.3 Error Detection Tools

Few tools are made to detect copy-and-paste errors, and those that do (CP-Miner [3] and the DECKARD-based tool [2]) utilize clone detection techniques on existing source code. Each tool attempts to first detect the copy-and-paste clones and then report any inconsistencies as potential errors. As a result, many false positives occur (both in the clone detection and error detection phases) and human intervention is required to consequently confirm or deny a reported bug as real. This process of finding errors "after-the-fact" is not the most effective way by itself to handle the creation and existence of bugs. We believe that it is better to prevent and detect bugs during software development while the programmer can fix mistakes on the spot.

CP-Miner, developed by Li, et al., [3] is the first known tool to do error detection in addition to clone detection. CP-Miner uses data mining techniques to more efficiently detect copy-and-paste code clones in existing, large software systems. Furthermore, CP-Miner is able to detect beyond exact clones and identify clones that have insertions, deletions, and modifications in them. It actually has an option to return only copy-and-paste clones with identifier renaming. For inconsistency and error detection, Li, et al., use "identifier mapping" such that an identifier is considered consistent when it always maps to the same identifier (which could be a different name) in the other fragment and it is inconsistent when it maps itself to multiple identifiers. Of course, false positives remain and actual bugs still need to be verified manually. Li, et al., conclude that copy-and-paste error detection should be provided in an IDE like Microsoft Visual Studio, but this feature is not yet available in the IDE.

More recently, Jiang, et al., developed a DECKARD-based tool [2] to detect bugs based on the clone's surrounding code, called its context. This tool has a clone detection component and an inconsistency or error detection component. First, Jiang, et al., use their clone detection tool, named DECKARD, to detect code clones in existing source code. The identifier renaming error, which Jiang, et al., call a "type-3 inconsistency", is determined by traversing the parse trees of clones and counting all of the unique identifiers that are visited. In particular, they count all identifiers, including macros, variable names, function names, type names, data fields, etc. (not including keywords and punctuations) and they do not distinguish between each kind of identifier. Their heuristic claims that a "type-3 inconsistency" exists if the two code fragments contain different numbers of unique identifiers. The process of simply counting unique identifiers, by itself, produces many false positives. As a result, manual inspection of the inconsistencies returned by the tool is still required. Jiang, et al., also suggest that automated tools could help prevent programmers from making copy-and-paste errors, such as identifier renaming errors, in the future.

5. CONCLUSION AND FUTURE WORK

The renaming of identifiers in source code can be error-prone for programmers to perform manually. Automated tool support in the integrated development environment (IDE) is needed to assist programmers with code modification in order to ensure code consistency. The CReN tool is our first step in inferring programmer's intent to help combat copy-paste errors, specifically with a focus on the consistent renaming usage pattern.

In the future, we would like to add some features to CReN. First, we would like to include support for the consistent renaming of any kind of identifier (right now it just renames variable names). We would also allow programmers to revert their intention of taking an identifier out of a consistently-renamed group. Also, instead of having CReN automatically decide when two clones become unrelated as a result of excessive editing, we would like to leave this decision to the programmers by allowing them to take clones out of a clone group.

In addition, we would like to generalize where consistent renaming is done. Right now CReN does consistent renaming just within a clone (copy-and-pasted code fragments). In the future, we would like to support consistent renaming within any user-defined scope. Furthermore, while CReN currently *infers* rules across all clones (for example, when a specific instance of an identifier is removed from the group to be renamed together, its corresponding identifier instance in all related clones is also removed), it does not *apply* the renaming itself across all of those related clones. We would like to possibly include this type of consistent modification feature into CReN (similar to linked editing or simultaneous editing). This feature would provide *structural synchronization* between members of a clone group. This would make CReN an all-in-one consistent renaming tool.

6. REFERENCES

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

[2] L. Jiang, Z. Su, and E. Chiu, "Context-Based Detection of Clone-Related Bugs", *European Software Engineering Conference (ESEC) and ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE)*, 2007.

[3] Z. Li, S. Lu, S. Myagmar, and Y. Zhou, "CP-Miner: A Tool for Finding Copy-paste and Related Bugs in Operating System Code", *USENIX-ACM SIGOPS Symposium on Operating Systems Design and Implementation (OSDI)*, 2004.

[4] B. Liblit, A. Aiken, A.X. Zheng, and M.I. Jordan, "Bug Isolation via Remote Program Sampling", *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2003.

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