Context-Sensitive Cut, Copy, and Paste

Reid Kerr
University of Waterloo
Waterloo, Canada

Wolfgang Stuerzlinger
York University
Toronto, Canada
http://www.cse.yorku.ca/~wolfgang

ABSTRACT

Creating and editing source code are tedious and error-prone processes. One important source of errors in editing programs is the failure to correctly adapt a block of copied code to a new context. This occurs because several dependencies to the surrounding code usually need to be adapted for the new context and it is easy to forget one of them. Conversely, this also makes such errors hard to find.

This paper presents a new method for identifying some common types of errors in cut, copy and paste operations. The method analyzes the context of the original block of code and tries to match it with the context in the new location. It utilizes a simple, pattern-based model of context, which we found to be well suited to the analysis of relocated blocks of text. Furthermore, we discuss the ability of our technique to detect semantic errors. While semantic errors are relatively difficult to recognize in a static document, our new technique can infer important information from the original context to detect some semantic mismatches. Finally, we present a proof-of-concept implementation and discuss our simple user interface for context-sensitive cut, copy and paste.

Categories and Subject Descriptors

D.1.m [**Programming Techniques**]: Miscellaneous, D.2.3 [**Coding Tools and Techniques**]: Program editors, Structured programming, H.1.2 [**User/Machine Systems**]: Human factors, Software psychology,

General Terms

Algorithms, Human Factors, Languages.

Keywords

Programming errors, context sensitive code, cut, copy, paste, development environment, code development.

1. INTRODUCTION

Cut, copy and paste (CC&P) functionality is ubiquitous in current editors. While it is a powerful and useful tool, CC&P can also cause problems. One such problem is the introduction of 'contextual errors' when text is relocated.

A segment of text within a document is often closely related to its surroundings. Blocks of text generally have connections to the text around them, and these connections are important for the block to 'fit' in its context. In a natural language document, for example, a segment of text must normally match its surroundings in terms of tense, person, use of singular or plural forms, etc. Pasting a block of text into a new location may introduce errors due to differences in context between the original and new locations: the tense may no longer match, singular and plural forms may conflict, etc.

While software developers are often cautioned against using copy and paste, several studies [6,8,10,14,16] have found that programmers frequently make use of this functionality for a variety of reasons. Within a source code file, any given segment of code likely has strong connections to its context. Such connections might include participation in syntactic constructions, conditional execution of code, consistent indentation, etc. These connections are also important if the code segment is to 'fit' with its surroundings; in this case 'fit' may mean 'execute correctly'.

Previous work has noted the need for editors that assist programmers in avoiding the introduction of dependency-related errors via CC&P [8,16]. In this paper, we present a new method for *context-sensitive cut, copy & paste* (CSCC&P). When a block of text is relocated, this method can be used to find connections with the original context that no longer exist in the new location. Alerted to these broken connections, the user can then avoid errors that may otherwise go undetected.

2. PREVIOUS WORK

Several projects have investigated the use of CC&P by programmers, and the implications of such use.

In one study of code reuse [10], the vast majority of new methods were created using existing methods as templates, with CC&P being the most common means of doing so.

Another study [6] found that programmers make frequent, significant use of CC&P. The authors found that the most common intention of CC&P actions consisting of more than one line of code was to use one block of code as a template for another, with the pasted block being modified appropriately for use in the new context. A second study [7] by two of the same authors investigated the evolution of 'code clones' in source code, often created by use of copy and paste. Their observations lead them even to question the common assumption that aggressive refactoring can and should be used to eliminate clones. They concluded that programmers often have no alternative to the creation of clones, and discuss approaches to deal with the difficulties in maintaining groups of clones.

In [15], a high-level tool intended to enhance the ease of changes in source code was presented. A study was conducted in which programmers were given access to the tool, and asked to perform a restructuring task in an IDE. During the task, four of seven participants encountered errors after relocating code; the programmers were alerted to these errors by the compiler. [14] discusses the extensive use of CC&P for code reuse as well, and also notes the tendency of programmers to rely on the compiler to catch errors introduced by pasting.

A detailed study of source code editing during software maintenance [8] identified CC&P as an important source of errors. When copying an existing block of code and modifying it to perform a similar function, the programmers in the study had difficulty detecting dependency-related errors that were not syntactic in nature, because these errors were not detected by the compiler. The authors propose highlighting of dependencies and hiding of unrelated code as techniques to assist in preventing these types of errors.

In [3] and [4], the authors discuss the common use of CC&P in the restructuring of code. They detail a system that aids in the restructuring of programs. This system automatically makes changes in other related places in the code when the programmer initiates common transformations, in order to maintain consistency. However, this system does not provide support for some common uses of CC&P such as code templates; instead, its goal is to ensure that a program's meaning does not change during restructuring.

While contextual errors caused by CC&P have been recognized as a problem, little work appears to have been done on tools to deal with this problem. One such tool is proposed in [16]. This paper discusses the heavy reliance on CC&P for code reuse, which takes the form of copying a block of code from one location and pasting it into another, and then modifying it so that it is appropriate for the intended purpose. The authors discuss the impact of CC&P operations on dependencies that exist between a block and its surrounding text. They describe a prototype visualization tool, which is intended to help the programmer understand the consequences of a CC&P action by displaying all dependencies of a selected block of code. However, this tool was not a fully interactive editing system, did not provide information that was specific to the user's desired CC&P action, and made no effort to detect errors introduced into the code by the action.

3. CONTEXT SENSITIVE CUT, COPY AND PASTE

Various Computer Science research projects have tried to use knowledge about the context to solve a given problem better. However, the models and methods typically used in this kind of research appear to be too general to apply to the use of CSCC&P to find errors. Examples include the use of context for word correction [9], word classification [2] in the analysis of natural language, and the use of heuristic matching of context in source code to select an appropriate code example from a database [5].

In this work, we are focused on a very concrete and structured domain, namely software development and CC&P in particular. Restricting ourselves to the analysis of relocated blocks of text, we found that a simple, pattern-based model of context is effective for detecting many common types of errors. This model allows for a straightforward approach to CSCC&P.

3.1 Context Model

In this paper, a *contextual relationship* (or *context relationship*) is a relationship involving at least one element contained within the subject block of text (*internal elements*), and at least one element outside the block (*external elements*). The definition of relationship has intentionally been left unspecified, since there are many types of relationships and which one of them is meaningful depends on the application. For example, a relationship may be one of Java syntax (e.g., in a variable declaration, the variable's type is followed by the variable's name), or one of semantics (e.g., in a counter-controlled loop, the termination condition is dependent on a variable's state, and at least one statement inside the loop increments that variable). Similarly, the notion of *element* is dependent on the type of relationship being considered.

Limiting the definition of context to (elements in) the surrounding text omits valuable information about the relationships themselves. In our work context can be more usefully defined as the set of context relationships for a particular block of code. In CC&P operations, we are concerned with relocated blocks of text, and hence, changes in context. We have defined two types of context to facility analysis of these changes.

A *literal context relationship* is a context relationship in which all participants are specific elements of the text: the relationship is fully 'bound' to elements of the text. A *literal context* is a context consisting of literal context relationships. Considering a CC&P operation, literal context consists of relationships between a copied block and its original surroundings, and literal context relationships exist exclusively between elements in the subject block and elements in the block's surroundings in its original location.

When a block is pasted in a new location, literal context relationships will often be severed. However, elements in the new surroundings may be capable of fulfilling the roles of elements that are missing in the new location. While the elements may change, the relationship itself can be maintained if it can be rebound to 'equivalent elements' in the new context.

For each literal context relationship, we can define a corresponding context relationship of this nature. A *pattern context relationship* consists of a relationship and internal elements that are identical to those of its literal counterpart, and external elements that are 'equivalent' to those of the literal relationship, where 'equivalent' means fulfilling the same role. The external elements of a pattern context relationship are 'unbound'—essentially, the relationship constitutes a pattern that can be matched using external elements of matching role. *Pattern context* is a context consisting of pattern context relationships.

To illustrate, consider the short code segment in Figure 1. A contextual relationship exists between the reference to the variable x in the `if` statement's decision, and the reference to x in the `println` statement. The relationship is one of 'guarding': the first reference to x is used to prevent the `println` statement from being executed if x is null.

```
if (x != null){
    System.out.println("X is: " + x);
    return;
}
```

Figure 1: A code sample, with a block to be copied highlighted.

In this situation, a literal context relationship exists between these two specific references to *x*. In contrast, if the subject block is guarded against being called when *x* is null, then a pattern context relationship is satisfied, regardless of which reference to *x* is used to do so—the instance of *x* used to do the guarding is 'equivalent' to that in the original context.

This pattern-based model of context is consistent with the use of CC&P in the editing of source code — the new surroundings of the pasted block may be different from the original, but certain relationships must still be fulfilled in order for the code to work properly. From this perspective, the relationships are more important than the context elements themselves, since the elements may change, but the relationships persist from one location to another.

3.2 Focus on pattern context

One can easily imagine a CC&P system with sensitivity to literal context. Such a system might be applicable when blocks are relocated to relatively nearby positions (e.g., in reordering of instructions), since we may be interested in the consequences of changes to existing relationships.

However, as discussed above, it has been frequently observed that CC&P is commonly employed to reuse code and create templates. These uses imply that blocks are often pasted into unrelated locations, severing literal context relationships, which is consistent with the idea of pattern context. For this reason, the system presented here focuses on pattern context.

3.3 Method

Based on the idea of pattern context, a straightforward method can be used to detect some types of relationships that may have been violated due to contextual changes.

Relationships are represented as patterns. More specifically, a pattern defines a participant in a contextual relationship in terms of its surroundings (i.e., the other participants in the relationship.) Using the example of Figure 1, a variable reference which is guarded might be defined as "a variable contained in the body of an `if` statement which executes its body only if the variable's value is not null". A match found for this pattern in a document indicates that an instance of the relationship exists, and identifies the target element of the pattern, one of the participants. A complete definition of a relationship consists of a set of patterns, one identifying each participant. The set of all defined patterns represents the set of all known contextual relationships.

When the user issues the copy command, the copied block is analyzed to determine its relationship with its context. Specifically, this consists of testing the copied block against all of the known patterns. Each pattern match found with a target located in the copied block indicates an instance of a contextual relationship that involves the copied block. All such matches are stored, and constitute the entire (known) pattern context of the block.

When the user issues the paste command, the pasted block is analyzed to determine how well it 'fits' in its new location. Each pattern match found during the copy action is now tested against the pasted block in the new surroundings. If a corresponding match is found in the pasted block, then the contextual relationship is satisfied in the new location. If no match is found then no equivalent element exists in the new context, and the relationship is unsatisfied. The user is then made aware of all such broken relationships, so that he may take corrective action if necessary.

Note that a strict definition of pattern context includes only relationships which 'cross the boundary' between the subject block and surroundings. This implies that relationships contained entirely within the subject block at the time of copy should be ignored. For practical purposes, however, these relationships need not be excluded. Any such relationships found within the copied block will also be present in the pasted block, and as such, will not be highlighted as violations.

Clearly, some form of pattern matching algorithm is required to implement this idea. The choice of matcher is important because the relationships in the known set need not be of uniform type, and the nature of the relationships that can be detected will be limited by the expressive power of the pattern language.

3.4 Detecting Semantic Errors

A valuable feature of the method presented here is its ability to detect some semantic errors. The analysis of text during CC&P affords certain advantages over analysis of text in a single, static instance of a document.

When analyzing a static document, it is possible to tell what semantic role an element is playing, but it is difficult to determine what role it is *intended* to play, or what role it *should* play, if these differ from the actual role. For example, consider the Java code segment in Figure 2, which a programmer has just typed into an editor.

```
while(x < 10){
    System.out.println
        ("Incrementing");
    x = someComplexFunction(x);
    y+=1;
    z+=1;
}
```

Figure 2: A static code sample.

In a loop, one often includes a statement that increments a counter inside the loop body, and use that counter to control loop termination. There is no obvious statement in the above loop body that performs this role; if a mistake has been made, this loop may run without termination. Should one of these statements have incremented *x*? Perhaps one of the references to y or z was a mistake. Perhaps the call to `someComplexFunction` will modify *x* appropriately, but it may be impossible to verify this. Perhaps the `while` statement's termination condition was supposed to be based on *y* or z instead of *x*. It is difficult to provide an error message to the user in this case, because there is no certainty that an error even exists, let alone what the error is or where it occurs. We cannot be certain, because we do not know the semantic roles the programmer intended for each element, and there are no clear cues to guide our analysis.

In contrast, consider if the loop body was copied and pasted from another block of code. The body is highlighted in its original surroundings in Figure 3, and in its new surroundings in Figure 4.

In this case, we can infer the semantic relationships that should hold in the destination by determining which role each element played in the original location. In the source block, the statement '*y+=1*' incremented the counter; there is a relationship between the instance of y in the while statement's termination condition, and the instance of *y* inside the loop body. When a programmer copies code from one location and pastes it into another, there is a high likelihood that each element is intended to continue to serve a similar role in the new location — this is consistent with the notion of a template. This allows us to identify a likely problem in the destination: '*y+=1*' was incrementing a loop control variable in its old context, but is no longer doing so in the new context. Perhaps the statement needs to be changed to '*x+=1*', or perhaps the while condition should be changed to '*y<10*'; in either case, we can alert the programmer so that he can decide if and how to correct the situation.

Thus, while CC&P often causes errors to be introduced through changes in context, it also allows detection of some types of semantic errors that cannot be easily found during analysis of a static document. Given the frequency of CC&P operations performed by programmers, it is important to pursue this opportunity to improve efficiency and accuracy.

```
while(y <= 100){
    System.out.println
        ("Incrementing");
    x = someComplexFunction(x);
    y+=1;
    z+=1;
}
```

Figure 3: A copied block.

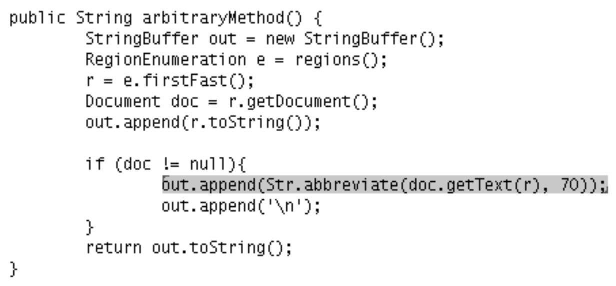
```
while(x < 10){
    System.out.println
        ("Incrementing");
    x = someComplexFunction(x);
    y+=1;
    z+=1;
}
```

Figure 4: A pasted block.

3.5 Interface

It is fairly straightforward to design a user interface for the discussed CSCC&P functionality. We choose to model our user interface after techniques commonly used in word processors and IDEs, and describe it in this section.

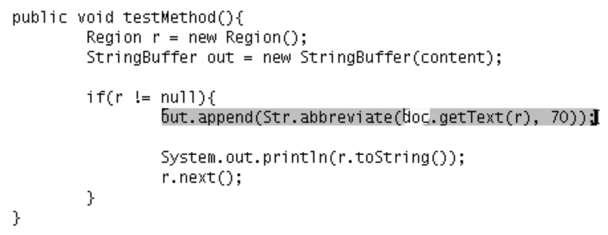
The block of code selected in Figure 5 is used as an example in this section. When the user executes a paste, the analysis introduced above is performed on the pasted block in the new location, and the editor enters 'context-correction mode' (or 'CCM'), shown in Figure 6. While CCM is active, the newly pasted block is highlighted in green, while any internal elements participating in context relationships that are now violated are highlighted in yellow. If a user right-clicks on an element, a list and brief explanation of the violation(s) that apply to the element are provided in a pop-up menu, shown in Figure 7. The menu allows the user to perform two actions on any of the violations. The 'ignore' option excludes the relationship from further analysis. The 'explain' option displays a window (Figure 8) containing both a full description of the relationship broken, and the complete text of the contextual relationship in its original form/location, with participants highlighted.



```
public String arbitraryMethod() {
        StringBuffer out = new StringBuffer();
        RegionEnumeration e = regions();
        r = e.firstFast();
        Document doc = r.getDocument();
        out.append(r.toString());

        if (doc != null){
                out.append(Str.abbreviate(doc.getText(r), 70));
                out.append('\n');
        }
        return out.toString();
}
```

Figure 5: Copying a block.



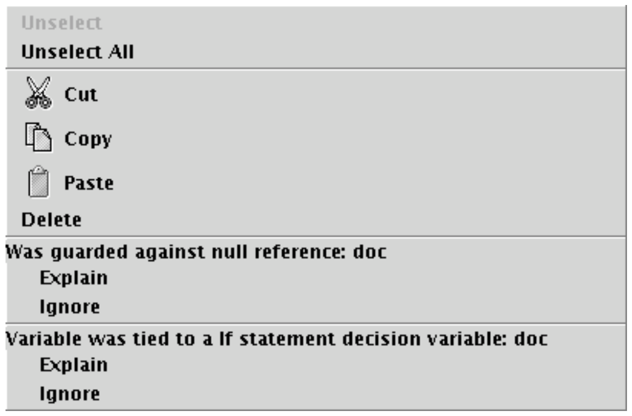
```
public void testMethod(){
        Region r = new Region();
        StringBuffer out = new StringBuffer(content);

        if(r != null){
                out.append(Str.abbreviate(doc.getText(r), 70));

                System.out.println(r.toString());
                r.next();
        }
}
```

Figure 6: Pasting a block.

While CCM is active, the user also has the opportunity to make changes to the document, and 'recheck' the contextual relationships against the updated document. Ideally, rechecking would be performed automatically as the document is modified. However, if performance constraints make this impractical, the document can only be rechecked on demand. CCM remains active until the 'end check' button on the tool bar is pressed, at which point all state and indicators relating to the current paste block are dismissed.



Was guarded against null reference: doc

Variable was tied to a If statement decision variable: doc

Figure 7: The context-sensitive menu.

4. IMPLEMENTATION

To implement a proof-of-concept system, we used an existing programming editor with suitable pattern matching capabilities. For this, we selected the LAPIS editor.

4.1 LAPIS

LAPIS [13] provides both a text editor interface and a Java parser. More importantly, LAPIS provides advanced pattern matching support using a system of *lightweight structured text processing*.

Lightweight structure is defined in LAPIS as "the ability to recognize text structure automatically, using an extensible library of patterns and parsers." Patterns are defined in a *text constraints* language based on elements recognized by various pattern-matching systems and parsers, connected by operators of a so called region algebra. This capability is extremely useful in the creation of context relationship patterns. For example, it is useful to be able to write patterns consisting of both Java elements and positional relationships.

E.g., to find variables which have been involved in the initialization of a loop, we might look for "variables located between the first bracket and the first semicolon of a for loop". In LAPIS's text constraint language, this is expressed as shown in Figure 9:

```
basicvariable
    in from 1st '(' in forstatement
        to first ';' in forstatement
```

Figure 9: A text constraint pattern in LAPIS.

(For details of the structure of patterns using the text constraints language, please refer to [13].)

4.2 Modifications to LAPIS

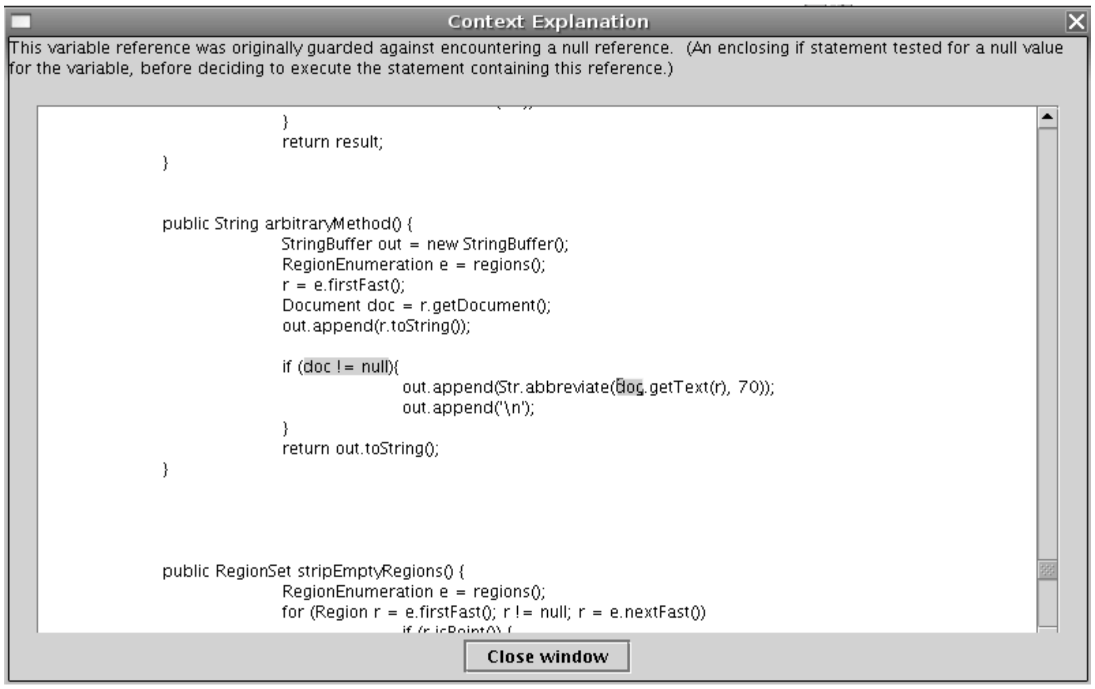
As work proceeded, we learned that our pattern matching requirements were slightly outside the LAPIS system's capabilities. While there was great flexibility in the composition of patterns, no form of variable binding was provided. The importance of variable binding is illustrated in Figure 10, which shows a naïve attempt to define a pattern targeting a guarded variable reference.

```
basicvariable
   in ifbody
          just after ifcontrols
                    contains
                              basicvariable
                                        then /!=/ then "null"
```

Figure 10: A flawed pattern, without binding.

This pattern again demonstrates the power of the LAPIS matching engine, incorporating Java elements, spatial relationships, and regular expressions. The pattern, expressed in English, matches elements which consist of "a variable located in the body of an if statement following an if's decision expression which contains a variable then '!=null'". There is one problem with this definition—it does not specify that the variable in the if body is the same variable found in the if statement's decision. This pattern will match any element of type 'basicvariable' in the body of an if statement testing for a null reference, whether the 'basicvariable' matches that in the test or not. We were unable to find a way to correctly express these types of relationships in general using the standard LAPIS text constraint implementation.

To implement CSCC&P, we needed variable binding capabilities. Unfortunately, the LAPIS matching system was not intended to support this, and was optimized for speed in a way that made it very difficult to add this capability directly. Our solution was to



This variable reference was originally guarded against encountering a null reference. (An enclosing if statement tested for a null value for the variable, before deciding to execute the statement containing this reference.)

```
                }
                return result;
        }


        public String arbitraryMethod() {
                StringBuffer out = new StringBuffer();
                RegionEnumeration e = regions();
                r = e.firstFast();
                Document doc = r.getDocument();
                out.append(r.toString());

                if (doc != null){
                        out.append(Str.abbreviate(doc.getText(r), 70));
                        out.append('\n');
                }
                return out.toString();
        }



        public RegionSet stripEmptyRegions() {
                RegionEnumeration e = regions();
                for (Region r = e.firstFast(); r != null; r = e.nextFast())
```

Figure 8: The context-explanation window.

implement a preprocessing layer that finds matches for the variables in a pattern first, and then generates bound, variable-free patterns by replacing the variable references with the matched text. These bound patterns are then matched using the standard LAPIS pattern engine. The performance implications of this decision are discussed below.

4.3 Patterns

With the chosen system for binding variables, pattern design can have a major impact on performance. Two equivalent patterns (in terms of the elements that they will match) may have dramatically different execution times, depending on the structure of the patterns. In this section, the design of efficient context patterns with variable bindings will be illustrated.

The preprocessing phase consists of two passes. In general, it is very advantageous to restrict the range of variables to the smallest possible set of values, to minimize the number of patterns generated by the preprocessor in these phases. For this reason, two terms were added to the pattern language for variable binding. Both the 'bind' and the 'var' terms were designed to be replaced with the values of the associated variable in the second pass of the preprocessing algorithm. However, the bind term is used to explicitly indicate which sub-pattern will be used in the first pass to generate the set of values for the variable. Binding the variable to the most restrictive sub-pattern results in significant performance improvements. Figure 11 illustrates the use of these terms, and provides a correct example of a pattern targeting a guarded variable reference.

```
basicvariable
        bind(var::A)
  in ifbody
        just after ifcontrols
                contains
                        basicvariable
                            equal to var::A
                          then /!=/ then "null"
```

Figure 11: A pattern employing binding.

A more complicated pair of patterns, used to identify a variable that has been declared previously, is presented in Figures 12 and 13. While this relationship is not particularly interesting from the standpoint of CSCC&P (since it can be readily detected in static documents by a compiler), it demonstrates that more complex relationships, like those involving scoping rules, can be recognized with these patterns. It also illustrates another limitation of the variable binding implementation. When multiple variables are used, each variable corresponds to one level in the preprocessor's recursion tree; restricting the number of variables used has a dramatic impact on performance. The pattern in Figure 12 is simpler, but makes use of two variables — variable B is used to ensure that the same 'scoping unit' contains both instances of the variable name, to handle instances of nested blocks. The pattern in Figure 13 is more complex, but makes use of only one variable — the last four lines of this pattern ensure that the variable declaration is not nested in a deeper block than the later variable reference.

The first pattern, matched against an entire document in one test, was manually stopped after 15 minutes of execution. Matched against the same document, the second pattern completed in several seconds (Note that typical execution is much faster than this, since searches are usually constrained to sections of code that are typically quite short.)

While the construction of patterns is not trivial, we found that we were able to define suitable patterns for each context relationship we wished to test.

```
identifier
    bind(var::A)
  in
    scopingunit
        bind(var::B)
  anywhere after
        variableInDeclaration
                equal to var::A
           in
              scopingunit
                equal to var::B
```

Figure 12: A simple pattern with two variables.

```
identifier
    bind(var::A)
  anywhere after
    variableInDeclaration
      equal to var::A
 in
   scopingunit contains
                variableIndeclaration
                  equal to var::A
     not contains
           /./ then scopingunit
                   contains
                        variableindeclaration
                                    equal to var::A
```

Figure 13: A more complex pattern with one variable.

4.4 Pattern generation

While the method proved effective, a CSCC&P system's value will depend heavily on the set of relationships used for analysis. While there are several potential sources of patterns, a key source is likely to be hand-coded patterns. Other forms of pattern generation may seem more attractive, but this approach is common in the analysis of language — consider the grammar-checking capabilities of common word processors. Furthermore, hand-generation of patterns allows the author to include detailed descriptions of a relationship, which may be of value to the user in correcting violations.

4.5 Other Issues

The Java parser used by LAPIS does not allow patterns to be matched if a document contains syntax errors. Other parsers may behave similarly, and this is an issue because CC&P actions may result in syntax errors. Our approach to this problem was to highlight the syntax error for the programmer at the time of copy or paste, instead of trying to analyze the context. If the error existed at the time of copy, the user could then correct the error and attempt to copy again. If an error was introduced by pasting, our editor remains in CCM, and allows the user to correct the error and subsequently recheck the block.

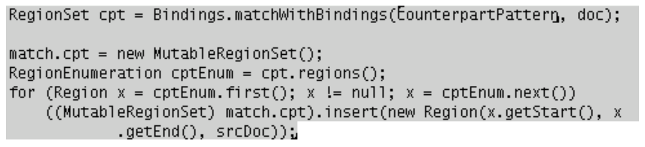
For performance reasons, our implementation did not make use of automatic rechecking during CCM, but instead allowed on-demand rechecking using a button on the tool bar.

5. RESULTS

In initial testing, our implementation was able to accurately and consistently detect violations of a number of contextual relationships when a block of code was relocated. The set of relationships tested included the following:

- A variable reference that was guarded against a null reference.
- A reference to a variable inside the body of an if statement that was also referenced in the if's decision (i.e., execution of the statement using the variable was dependent on the value of the variable).
- A variable appearing in an if statement's decision that also appeared in its body (i.e., the variable reference was used to determine if a later reference would be executed).
- A variable reference inside a for loop body that was a reference to the loop's iterator.
- A variable reference in a for loop body that was a reference to a variable used to control the loop.
- A variable reference in a for loop's control block that was referenced inside the loop.
- A variable reference in a while loop body that was a reference to the variable used to control the loop.
- A variable reference in a while loop's termination condition that was referenced inside the loop.
- A region of code that was repeated.
- A region of code that was conditionally executed.
- A region of code that could not generate exceptions (i.e., inside a try block).
- A region of code that was executed only in the case of an exception (i.e., inside a catch block).
- A region of code that was executed whether or not an exception was encountered (i.e., inside a finally block).
- A variable that was a field in a class.
- A variable that was enclosed in a method (i.e., was not a field in a class).

Testing was performed on a mid-range desktop machine. When copy and paste blocks consisted of several lines of code and all of the relationships listed above were tested, analysis typically completed in less than two seconds, despite the performance limitations of the pattern matching system. A paste block resulting in several violations with overlapping elements is shown in Figure 14. The context-sensitive menu resulting from right-clicking on a token which belongs to several of these elements is shown in 15.



```
RegionSet cpt = Bindings.matchWithBindings(counterpartPattern, doc);

match.cpt = new MutableRegionSet();
RegionEnumeration cptEnum = cpt.regions();
for (Region x = cptEnum.first(); x != null; x = cptEnum.next())
    ((MutableRegionSet) match.cpt).insert(new Region(x.getStart(), x
            .getEnd(), srcDoc));
```

Figure 14: A pasted block with several violated relationships.

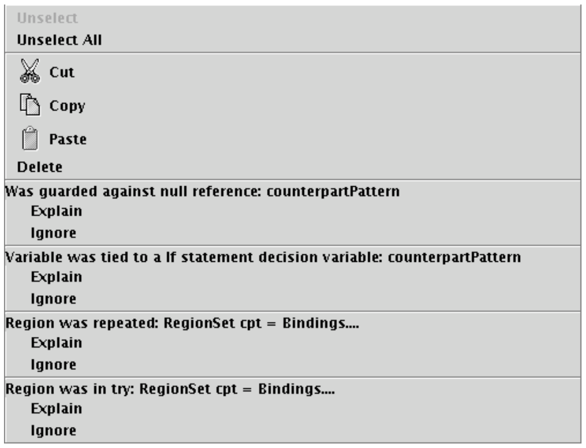

Unselect
Unselect All
Cut
Copy
Paste
Delete
Was guarded against null reference: counterpartPattern
Explain
Ignore
Variable was tied to a If statement decision variable: counterpartPattern
Explain
Ignore
Region was repeated: RegionSet cpt = Bindings....
Explain
Ignore
Region was in try: RegionSet cpt = Bindings....
Explain
Ignore

Figure 15: The right-click menu for a token, which participates in several violated relationships.

Although further testing is required, we would expect any failures in the detection of these relationships to be due to faults in the patterns we have used, and not faults in the method itself.

Based on the success in detecting violations of this initial set of relationships, our method for CSCC&P appears to be very promising, and should be investigated further. That said, our implementation is unlikely to have value for further development. First, while the performance limitations discussed above were acceptable for a proof of concept, the system may not provide adequate performance with larger bodies of code. Second, while LAPIS provides a Java parser, it does not provide most of the features of a specialized programming editor. Further development and testing should likely make use of a more complete, specialized platform.

5.1 Tracking changes in CCM

One important question arose, related to the handling of changes to the text while a user corrects context-related problems. During CCM, the user is free to edit and recheck the document. Changes outside of the paste block are trivial to handle; since the relationships are not tied to specific external elements, we can simply try to rebind to new elements. However, changes in the paste block itself are problematic — targets of relationships are moved or deleted, names are changed, etc. Ideally, the system will continue to offer meaningful analysis despite such changes.

There are two obvious approaches that might be taken here:

- Treat internal elements as strings without consideration of location within the block. By doing so, movements of elements within the block can be largely ignored. However, there is the risk of confusion between elements with different types but equal string values. Moreover, textual changes to the elements will hinder analysis in this case.
- Track internal elements based on their location within the block. This method will deal admirably with name changes. It could also handle positional changes if element locations are updated to mirror the changes introduced in the text during editing. However, this method will have difficulty dealing with complicated restructuring of text. Further, the actual movement of

elements in the document's text may not match the movement intended or perceived by the user.

Neither of these choices offers a perfect solution, and there is clearly room to develop a more sophisticated method. The second approach, tracking elements by location, was chosen for our system because it provided the greater accuracy.

5.2 Discussion

The current system is targeted at the Java programming language. The general principles of CSCC&P apply to many other programming languages. However, for each language, the set of patterns will have to be adapted to the programming patterns particular to a language.

The list of patterns presented at the beginning of section 5 is also only a starting point. Many other patterns exist and are sensible. In that sense, our initial exploration of CSCC&P has only scratched the surface of the potential of this new technique. However, any significant step further would require an in-depth study of programming patterns *in correlation with* common programming errors. This goes far beyond the scope of this work.

6. CONCLUSION AND FUTURE WORK

We have demonstrated a first implementation of a context sensitive cut, copy and paste system. Our results indicate that a pattern-based implementation of CSCC&P is worthy of further investigation. The new method we presented proved successful in detecting violations of a variety of common contextual relationships, including semantic relationships.

We presented the addition of variable bindings to an existing, powerful text constraints language. We then used this expanded language to define patterns that detect a variety of important contextual relationships. While this language may prove useful for further investigation, other alternatives should be investigated for this task as well. The LAPIS pattern matcher platform may unfortunately not prove to be fast enough for such tasks.

To investigate CSCC&P further, the presented ideas would have to be re-implemented in a state of the art software development environment. With that, user testing can be conducted to determine the value of CSCC&P to programmers in practice. We expect that this would provide great insights. A larger set of code relationships should be developed, as well.

Also, additional work is needed in the area of pattern generation and management. While a set of hand-coded rules proved quite beneficial in our experiments, it may also be possible to use methods to extract patterns from code samples. For example, extensive research into the use of clustering to understand software structure [1, 11, 12] may have relevance here. In addition, refining and prioritizing the patterns in the database, once user studies have revealed the patterns that provide the most value in practice, might improve the system.

7. REFERENCES

[1] P. Andritsos, V. Tzerpos. Software clustering based on information loss minimization. Working Conference on Reverse Engineering, 334-344, 2003.

[2] W. Cohen, Y. Singer. Context-sensitive learning methods for text categorization. ACM Trans. Inf. Syst., 17(2):141-173, 1999

[3] W. Griswold, D. Notkin. Computer-aided vs. manual program restructuring. ACM SIGSOFT Software Engineering Notes, vol. 17, no. 1, pages 33-41, 1992.

[4] W. Griswold, D. Notkin. Automated Assistance for Program Restructuring. ACM Trans. on Software Engineering and Methodology vol. 2, no. 3, pages 228-269, 1993.

[5] R. Holmes, G. Murphy. Using structural context to recommend source code examples. In Conference on Software engineering, 117-125, 2005.

[6] M. Kim, L. Bergman, T. Lau, D. Notkin. An Ethnographic Study of Copy and Paste Programming Practices In OOPL. In Symposium on Empirical Software Engineering, 83-92, 2004.

[7] M. Kim, D. Notkin. Using a Clone Genealogy Extractor for Understanding and Supporting Evolution of Code Clones. In Workshop on Mining Software Repositories, 17-21, 2004.

[8] A. Ko, H. Aung, B. Myers. Eliciting Design Requirements for Maintenance-Oriented IDEs: A Detailed Study of Corrective and Perfective Maintenance Tasks. In CHI '05 extended abstracts, 1557-1560, 2005.

[9] K. Kukich. Techniques for automatically correcting words in text. ACM Comput. Surv., 24(4):377-439, 1992.

[10] [B. Lange, T. Moher. Some strategies of reuse in an object-oriented programming environment. In CHI '89, 69-73, 1989.

[11] T. Lethbridge, N. Anquetil. Comparative study of clustering algorithms and abstract representations for software remodularization. IEE Software, 150(3):185-201, 2003.

[12] S. Mancoridis, B. Mitchell, Y. Chen, E. Gansner, Bunch: A clustering tool for the recovery and maintenance of software system structures. In Conference on Software Maintenance, pg. 50, 1999.

[13] R. Miller. Lightweight Structure in Text. PhD thesis, Computer Science Department, CMU, May 2002.

[14] M. Rosson, J. Carroll. The Reuses of Uses in Smalltalk Programming. ACM Trans. on Computer-Human Interaction, 3(3), 219-253, 1996.

[15] V. Sazawal, M. Kim, D. Notkin. A Study of Evolution in the Presence of Source-Derived Partial Design Representations. Workshop on Principles of Software Evolution, 21-30, 2004.

[16] G. Wallace, R. Biddle, E. Tempero. Smarter Cut-and-Paste for Programming Text Editors. Australasian User Interface Conference, 56-63, 2001.