Practical Language-Independent Detection of Near-Miss Clones

James R. Cordy
Queen's University
cordy@cs.queensu.ca

Thomas R. Dean
Queen's University
thomas.dean@ece.queensu.cam

Nikita Synytskyy
University of Waterloo
synytskyy@cs.uwaterloo.ca

Abstract

Previous research shows that most software systems contain significant amounts of duplicated, or cloned, code. Some clones are exact duplicates of each other, while others differ in small details only. We designate these almost-perfect clones as "near-miss" clones. While technically difficult, detection of near-miss clones has many benefits, both academic and practical. Finding these clones can give us better insight into the way developers maintain and reuse code, and we can also parameterize and remove near-miss clones to reduce overall source code size and decrease system complexity.

This paper presents a simple, general and practical way to detect near-miss clones, and summarizes the results of its application to two production websites. We use standard lexical comparison tools coupled with language-specific extractors to locate potential clones. Our approach separates code comparisons from code understanding, and makes the comparisons language independent. This makes it easy to adapt to different programming languages.

1 INTRODUCTION

Clones occur in all applications of more than trivial size. Over time, they have been the subject many scientific studies using a variety of methods [1,2,3,4,8,13,14,15,16,17,18]. Clones are interesting because studying them can further the understanding of how programmers use and re-use their code. This is possible because clones are more than textually identical pieces of code. For better or worse, cloning is an accepted—and widely used—code reuse technique, and it would be wise to use it to get some insight into the system's functionality and history. Near-miss clones are particularly useful for this purpose because they can show how a piece of code propagates through the system, evolving as it does so.

Apart from learning the code's history, study of cloning can also suggest opportunities for code refactoring and improvements. For example, the detected clones can be automatically or semi-automatically resolved in order to make the system simpler and easier to understand.

Locating near-miss clones is a rather challenging task, in part because the definition of near-miss clone is very imprecise. An exact clone, by comparison, has a very clear definition—if two pieces of code are same, they are clones; otherwise, they are not. Even with such a precise definition, exact clone detection is a complicated endeavor. Near-miss clones do not have an accepted definition, so one must find a suitable test for near-miss cloning before attempting any detection.

Lexical clone detection tools for near-miss clones [2,3] treat them as any sequence of strings that are the same with respect to a certain parameter, or parameters. Because of the purely lexical approach, detected clones do not correspond to structural elements of the language. Others based on metrics, concept analysis or dataflow analysis [13,14,15,16, 17,18] are aimed at procedural or ADT level structures only.

Our aim is to locate clones that correspond to discrete structural constructs of the language we are analyzing at many levels of detail. This, by necessity, calls for some form of parsing or other code comprehension technique. However, for both processing speed and ease of implementation, we wanted to use lexical tools to perform the comparisons themselves. As a result, our approach relies neither on purely textual compari-

Copyright © 2004 Nikita Synytskyy, James R. Cordy, Thomas R. Dean. Permission to copy is hereby granted provided the original copyright notice is reproduced in copies made.

sons, nor solely on compiler-based techniques, which involve parsing, generation and analysis of ASTs or similar data structures. Instead, we adopt an approach that is a combination of the two.

2 GENERAL APPROACH

This work is an extension of our previous work on detection and resolution of exact clones [8] using TXL [7]. As in our previous work, we use a robust island grammar to identify and isolate the syntactic constructs that are of interest [12]. This grammar represents the bulk of the code (HTML text in our experiment) as sequences of uninteresting tokens known as *water*. Interesting structural features (or *islands*) are identified and explicitly parsed as specific non-terminals in the grammar. In this work, an *extractor* generates a separate pretty-printed text file for each island found. Each such island (and thus each separate file) represents a *potential clone*. Comparing the potential clone files allows us to determine which islands actually are clones.

The island grammar itself is the only language dependent part of our process. The actual identification of near miss clones is done using entirely language independent lexical tools. To adapt the process to another language, all that is needed is a grammar for the language and the list of the non-terminals that represent the structural elements to be considered potential clones. This gives the process broad flexibility. All islands are extracted as potential clones, including nested islands. To our knowledge, this simultaneous multi-level comparison is different than any other method. The grammar for the target language does not even need to be complete, only detailed enough to identify the syntactic structure of potential clones.

When the islands are extracted to separate files, they are reformatted to a standard form (i.e., "pretty-printed") using annotations in the TXL grammar. The pretty-printing we do is geared towards better comparison results, rather than better human readability—our pretty-printer tries to make every line correspond to some meaningful source feature, thus spreading the source over many lines. As many program elements as possible are located on their own lines. This helps improve the precision and granularity of line-based comparisons.

We then compare these extracted source code snippets to each other line-by-line, using Unix's *diff* [9,10] utility. This method benefits from the simplicity of purely lexical approaches, but provides more accurate results, because any formatting irregularities are removed from the source, and because all clones correspond to some language construct in the language being analyzed.

While it is true that this method may miss potential clones that are semantically the same, clones that are the result of cut, paste and modify on the part of developers are easily found. The result is a simple, efficient and effective method without the use of heavy weight metrics and language dependent computations. We have evaluated our approach as applied to several thousand lines of HTML in two medium-sized production web sites.

3 METHOD OF OPERATION

Clone detection in HTML. Thus far the method presented in this paper has been primarily aimed at clone detection in mark-up languages, namely HTML. As a result, although the technique we describe is independent of language, the examples in this paper are aimed at clones in HTML pages. Research shows that web sites contain at least as many clones as other software; usually, however, the percentage of cloned content in a web site is far greater than in normal software [11].

Authors of web sites are under severe pressure to clone, much more so than programmers using executable languages. Consider the following points: web sites are developed quickly and updated often, leaving the maintainers little time to ponder the benefits of following good software engineering practices; in many cases web site maintainers have not been exposed to these practices in the first place [5]; HTML offers virtually no code reuse tools, forcing authors to rely on cloning as the only way of reuse; finally, web pages of the same web site are almost always expected to have a common "look and feel" necessitating frequent code reuse, and thus promoting cloning even further.

These factors, coupled with the fact that the use of mark-up languages in general and HTML in particular has experienced huge growth with the emergence of the World Wide web, make HTML a prime target for reverse engineering research, and particularly for clone detection efforts.

```
<html>
<table id="outerTable" border=1>
  <tr><td>
   <tableborder=1
id="innerTable1">
       <tr><td>
         content of inner table 1
       </td></tr>
   </table>

   < t a b l e          b o r d e r = 1
id="innerTable2">
       <tr><td>
         content of inner table 2
       </td></tr>
   </table>
  </td></tr>
</table>
</html>
```

Figure 1: A sample web page with two near-miss clones.

Figure 1 shows a sample HTML file that will be used as an example throughout this paper. It is a simple file that consists of three tables, with the outermost table containing the other two. The two inner tables are very similar to each other, differing only in one table parameter and the content of their single cell.

Clone detector structure. Our clone detector tool is structured, conceptually at least, according to a pipe-and-filter design pattern. Clone detection is performed in three separate steps, with the output of one step becoming the input to the next one. These steps are:

1. Potential clone extraction;
2. Clone comparisons;
3. Output generation.

Figure 2 shows the conceptual architecture of the system presented here, and the data flow through it. The following sections will examine each step in turn.

3.1 Extraction of potential clones

Every clone extraction tool designates—sometimes implicitly—a "minimal clone", i.e. the smallest piece of code that the tool considers to be worthwhile to examine on its own. This step is important for two reasons: it reduces the amount of work the clone detector has to do, and makes the results more relevant. The amount of work is cut because the tool does not spend time looking for clones of program entities that are too small, and the results are improved because they are not polluted with information about the "cloning" of single tokens or statements.

In our system, the extractor is responsible for enforcing the minimal clone restrictions. Its task is to extract potential clones from the source code for further study, and it is responsible for extracting features no smaller than our designated minimal clone.

The definition of a minimal clone also varies from language to language. In the case of HTML, we designate individual tables and forms as minimal clones. Each potential clone is extracted only once, but if potential clones are nested (see Figure 1 for an example of a table occurring inside another table), the inner candidate is listed twice: once with its parent and once on its own. All extracted potential clones are stored as text files, with the file names indicating their origin.

During extraction, potential clone files are also pretty-printed. Pretty-printing ensures consistent

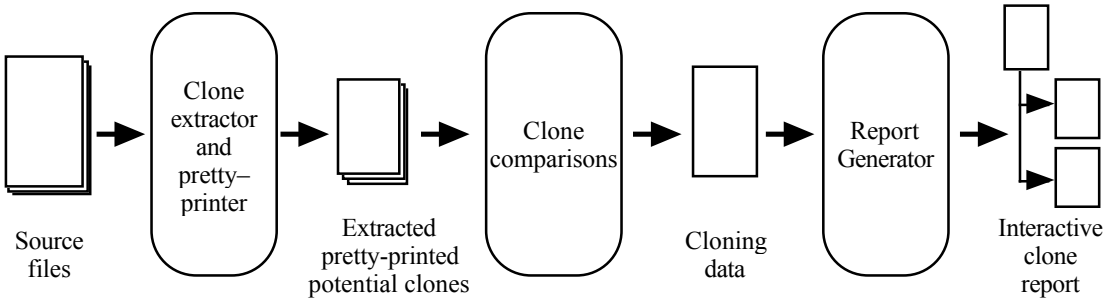

Figure 2: Conceptual architecture of the language-independent near-miss clone detection system.

layout of code for later comparisons. When code is cloned, it is often changed—whitespace and comments are inserted or removed, tags or block markers are moved around to suit the developers' tastes better, and so on. Whitespace and comment removal would have addressed these changes to some degree, but will not have necessarily eliminated all of them. By using pretty-printing, which is more invasive, we can guarantee that all code has uniform layout, which in turn yields an improvement in comparison accuracy.

Pretty-printing does more than just remove formatting inconsistencies, however. Our pretty-printing rules are designed with clone detection in mind—they localize the changes in the code, if any, to as few lines as possible. Whenever possible, code features, like HTML tags and parameters, are placed on separate lines. If a parameter of a table has been changed after the table has been cloned, only the line holding that parameter is affected. Because we use *diff* for comparing files to each other, our approach is inherently line-based, and spreading the code over more lines increases the granularity of the approach, and helps us determine the magnitude of a particular change correctly.

Figure 3 shows what extracted and pretty printed potential clones look like when HTML is being processed. The original file contains three potential clones—three tables. The tables are nested, with the outer table containing the inner two. Figure 3 shows how the first of the nested tables looks once extracted and pretty-printed. Other extracted tables are converted to a similar format, and are not shown for the sake of brevity.

```
<table
border=1
id="innerTable1"
>
  <tr>
    <td>
      content of inner table 1
    </td>
  </tr>
  </table>
```

Figure 3: An extracted and pretty-printed potential clone.

3.2 Comparison stage

Potential clones are compared to each other using the Unix *diff* command. *Diff* is a standard utility on most Unix systems, and can be used to determine the differences between two files (usually on a line-by-line basis), and display them in a number of formats.

Comparison Rules. To determine whether two source code structures are clones of each other, we compare them using a whitespace-insensitive *diff*. Specifically, we ignore changes that only add or delete blank lines, and ignore whitespace when comparing lines. We then use the output of *diff* to determine the number of unique lines in each potential clone. We then compute the percentage of unique lines for each file, defined as the number of lines unique to that file divided by the total number of lines in the file. If these ratios for both files are under a certain threshold, the files are considered to be clones of each other. The thresholds we used ranged between 30% and 50%. After a period of experimentation, we settled on a threshold of 30%, because it is liberal enough to allow clones with a significant amount of change to still be detected, while at the same time filtering out spurious results. The choice of threshold will likely vary between projects and programming languages, as the analysts using the system tune it to deliver the results they expect.

There are two special cases to be dealt with after the comparison has been performed and the number of unique lines in both files is found to be under the threshold. First, the number of unique lines might be equal to zero for both files. In this case the files are not merely clones, but identical clones—they have been copied to various places in the source code without any changes, or with changes that the pretty-printing step was able to remove. Another case is when only one of the two code snippets being compared has unique lines—i.e. one of the potential clones can be obtained from the other by merely deleting some lines. Instances of the second case are not considered clones and are removed from the result set. This is done because, based on our experiments, most of these results are caused by nesting, where a child construct is identified as a clone of its parent.

The comparison rule described above is more stringent than it might seem at first, because of the effects of pretty-printing, and the use of *diff*. Because of the pretty printing every line corresponds roughly to one source code feature (e.g. an HTML tag, or a name/value parameter pair). With the help of *diff*, the comparison is also able to take ordering into account. As part of its operation, *diff* finds the longest common line sub-

sequence between the two files being compared, defined as the longest sequence that can be obtained from either one of the two files by merely deleting some lines. For a line to be declared "common" between the two files, it is not enough for it to occur in both files; it has to be a line that occurs in the longest common subsequence of the two files. Correspondingly, the lines that have to be deleted from the files to generate the longest common subsequence are considered to be unique to their respective file. In fact, a line that occurs in both files might be considered as unique to both of them, if it is not part of the longest common subsequence.

Sub-clone removal. Sub-clones occur when two code structures (e.g. subroutines) have been determined to be clones, and their child structures (e.g. loops that both subroutines contain) have been listed as clones of each other as well. In tools that detect exact clones, sub-clone removal is a very necessary step. When working with exact clones, it is always the case that child structures are clones if their parents are clones. In this case information about the cloning of children is redundant, and merely pollutes and inflates the result set.

The same is not true when working with near-miss clones, because if two structures are clones

`<table`
`border=1`
`id="innerTable1"`
`>`
`  <tr>`
`  <td>`
`  content of inner table 1`
`  </td>`
`  </tr>`
`</table>`
	`<table`
	`border=1`
	`id="innerTable2"`
	`>`
	`  <tr>`
	`  <td>`
	`  content of inner table 2`
	`  </td>`
	`  </tr>`
	`</table>`

Unique lines: **2** Total lines: **10** Unique content percentage: **20%**	Unique lines: **2** Total lines: **10** Unique content percentage: **20%**

Figure 4a: Cloned tables. Lines unique to each table are highlighted.

`<table`
`id="outerTable"`
…14 unique rows omitted…
`  <table`
`  border=1`
`  id="innerTable2"`
`  >`
`  <tr>`
`  <td>`
`  content of inner table 2`
`  </td>`
`  </tr>`
`  </table>`
`  </td>`
`  </tr>`
`</table>`
	`<table`
	`border=1`
	`id="innerTable2"`
	`>`
	`  <tr>`
	`  <td>`
	`  content of inner table 2`
	`  </td>`
	`  </tr>`
	`</table>`

Unique lines: **19** Total lines: **29** Unique content percentage: **65%**	Unique lines: **0** Total lines: **10** Unique content percentage: **0%**

Figure 4b: Non-cloned tables. Lines unique to each table are highlighted. Some lines omitted for brevity.

of each other, it is not always the case that their children are clones of each other as well—after all, something has been changed to make the clones near-miss. For this reason, the sub-clones are interesting in themselves, and don't warrant automatic removal. Presently our system keeps the sub-clones, as well as the parents, in the result set. Our classification system (described below) makes sure that the children are always in a class of their own and never intermixed with their parents; this helps keep the result set manageable and understandable.

Comparison results. Figure 4 shows the results of two comparisons—one between the two inner tables from our example file, and one between the inner and the outer table. The lines unique to a given potential clone are highlighted. In Figure 4a we see the results of a comparison between two inner tables, which are very similar to each other. Most of the code that makes up the tables is the same—there are minor changes that affect only two lines in each table. The changes affect only 20% of the tables' line counts; this falls below the cloning threshold of 30%, so after the comparison these tables will be considered clones of each other.

Figure 4b shows the comparison between the outer table and one of its children. In this case the outcome of the comparison is very different—the inner table, unsurprisingly, has no unique lines because it is completely contained by the outer table. The outer table, however, has a lot of extra content—in fact, 65% of its content is unique with respect to the inner table. Because this number far exceeds the 30% cloning threshold we use, the tables are not considered to be clones of each other.

3.3 Performance Optimizations.

Clone detection efforts are inherently computationally intensive, and performance optimizations therefore remain an important issue. The complexity of clone detection, if not optimized, is quadratic or worse, because ultimately every line of code has to be compared to every other line of code. For large systems, such exercises quickly become prohibitively time-consuming.

Our system is no exception when it comes to performance issues, and also needs optimization. If no optimizations are used, every potential clone will have to be compared to every other potential clone, to determine whether they are actually clones of each other. Even for relatively small systems of 10,000 lines or so this can mean performing almost half a million comparisons (one of the web sites we analyzed, almost exactly 10,000 lines in size, generated 700 potential clones, meaning 490,000 potential comparisons).

Performing a comparison of two files is an expensive operation in our system. The most time consuming action in a comparison, by far, is running the *diff* command, which consumes over 80% of all time needed to perform a comparison. While the implementation of *diff* on Unix is very fast, it is the biggest performance bottleneck in our system. With that in mind, most of our optimization efforts are focused on reducing the number of comparisons we need to perform, by intelligently guessing which files don't need to be compared to each other.

Size restrictions. To save time on file comparisons, we have to be able to guess the outcome of a comparison based on some easily determined parameter. Size, or in our case, line count, is a good first indicator. If line counts of two potential clones are radically different, it is highly unlikely that they are clones of each other. We consider comparing two files worthwhile only if their line counts are within a certain range. More specifically, two files are compared only if the larger one is no more than twice the size of the smaller one, in terms of line count.

Size restrictions reduce the number of required comparisons significantly, but not enough to eliminate the need for other performance optimizations. Most of the extracted potential clones are quite small (in one of our experiments, 40% of the clones were between 4 and 22 lines), and still have to be compared to each other.

Clone Classes and Exemplars. When two potential clones (A and B) are compared and found to be clones, B is marked as being of class A. That is, A is considered to be the exemplar for the pair of clones. After this finding B is never compared to anything else. Instead, A is compared to all other potential clones(of comparable size) and any near misses are added to the class identified by A.

The effectiveness of this technique depends heavily on the code being analyzed. Obviously, the more clones in the code, the better this optimization works. In all our experiments, we have achieved significant speed-ups; analysis time requirements were reduced from days to minutes after clone classes were implemented.

Clone classes are useful for more than performance enhancement, however. They make the results of clone detection easier to understand for humans, and therefore much more useful. If the results were not allocated into classes, the ana-

lysts performing the clone detection would be faced, upon analysis completion, with hundreds, if not thousands, of clone pairs; the analysts would then be left to perform the classification themselves—a task too tedious, and therefore unlikely, to be done by humans. By automatically grouping the found clones into a comparatively small number of classes, we make the results of clone detection comprehensible, and therefore useful.

Figure 5 illustrates the clone detection process, as applied to five potential clones, named A through E. In the first comparison round, A is compared to each potential clone in turn, as illustrated by the dotted arrows. The outcome of the comparison is indicated by the type of the arrow head—pointed arrow heads indicate that a clone was detected, X-shaped arrowheads show that the two code snippets being compared were not clones of each other. A was found to have two clones—C and E.

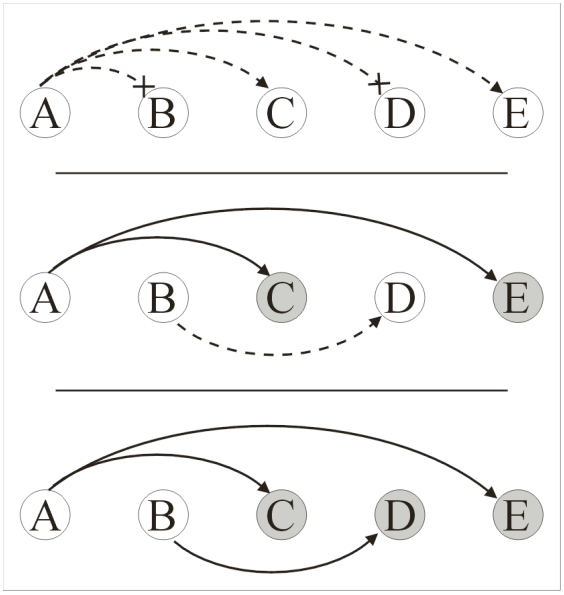

Figure 5: The comparison process as applied to five potential clones.

For the second round of comparisons, C and E are removed from the potential clone list, and added to A's clone class, as shown by the solid arrows. During the second comparison round, B is compared to all potential clones that remain in the list—that is, to D; C and E are removed from further comparisons, as indicated by shading. B and D are found to be clones, and the comparisons stop, because there are no more potential clones left to compare. In larger examples, the comparisons go on until there are at least two clones of comparable sizes that haven't been compared to each other. In the case depicted in Figure 4, three clone pairs, grouped into two clone classes, were detected.

3.4 Clone Report Generation

Deciding what to do with the clones, once they are found, is a task that is at least as difficult as the clone detection itself. Cloning ratios (the percentage of code that is copied at least once) range from 10% and 20% in large applications, and can be as high as 30% or more in web sites and web applications [11]. These numbers mean that even looking for clones in relatively modest-sized application of 100K lines, one will be faced with at least 10K lines of results upon task completion. That is a large amount of information for a human to understand, and presentation of the results determines, to a large extent, whether analysts will find the whole clone detection process useful.

Clone report structure. Since our clone-detection work has been focused primarily on web sites, we present the results to the users using the same artifact they're analyzing—a web site. The generated web site contains one index page, and a number of secondary clone report pages. On the index page, we display the exemplars that were used to create our clone classes. Next to an exemplar is a link leading to the clone report for the class, and a number showing how many clones the class contains. Information on the origin of every exemplar (the file it was taken from) is also shown. A clone report page for a given clone class lists all the clones in that class in the order they were found during processing, starting with the exemplar. Naturally, origins of all clones are also shown. Figure 6 shows both the index page and a secondary clone report page for our toy example. The page on the left is the index page which contains a link to the page on the right.

When dealing with source code of an executable language, the choices in code presentation are few—one is limited to examining the code itself. It is impossible to look at what the code does, because it is not functional on its own, and the execution results are usually impossible to visualize. Fortunately, this is not true for HTML. Markup languages can be examined in two ways—by looking directly at source code, and looking at "rendered" markup. Even taken out of context, HTML retains all, or most of its meaning. Some functionality may be missing or un

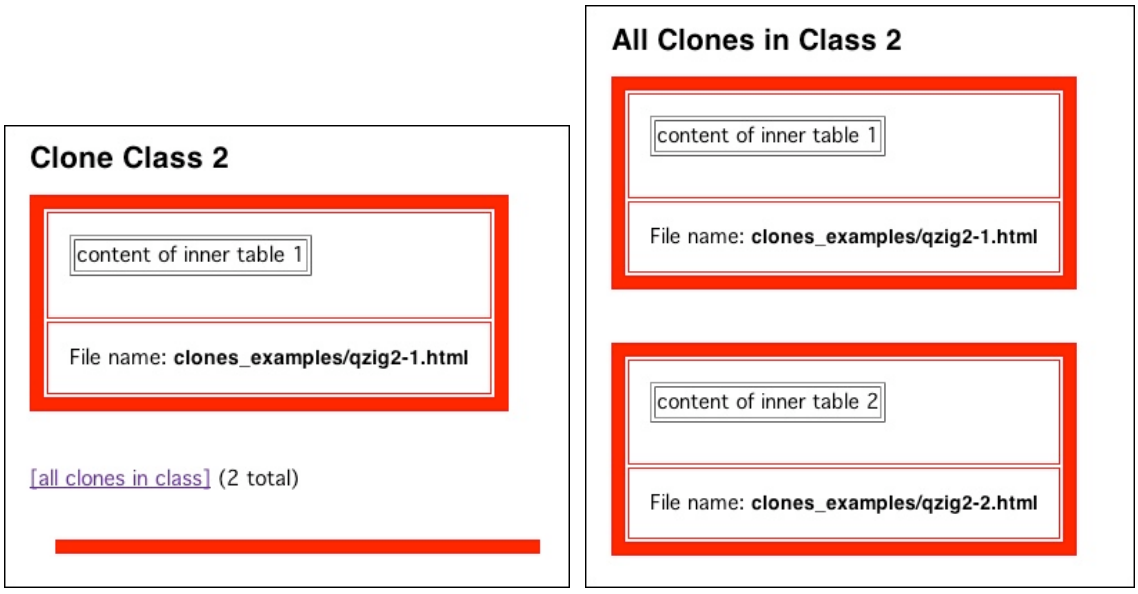

Figure 6: Interactive Clone Report Interface.

available, but more often than not HTML can still be rendered and the results of its "execution" examined.

Our result generation step takes advantage of this fact, and presents HTML clones not as source code, but as "executed" HTML. In effect, the analyst is shown what is cloned on the web site in terms of the web site itself, and not in terms of source code. This feature is intended to help relate the results of cloning analysis back to the original web site.

3.5 Implementation tools and portability

Our clone detection system is implemented using software that is either open source or available free of charge. Parsing of source code, as well as extraction and pretty-printing of potential clones is done with TXL [6,7]. Code comparisons and result filtering are performed with *diff* and *sed*, utilities that come standard with most implementations of Unix. The bulk of the source code is written in Perl, a freely available language that is also standard on most Unix distributions today.

The system will run correctly on any Unix machine that has access to the tools mentioned above.

4 Clone Analysis Results

We have applied our clone detection tool to two medium-sized web sites: the homepage of the TXL programming language (http://www.txl.ca) and the home page of the Society of Graduate and Professional Students (SGPS) of Queen's University (http://www.queensu.ca/sgps/). The size of both sites is approximately the same—about 10,000 lines of HTML code each.

Statistics and performance. Even though the two sites are similar in size, the results of clone analysis show that they are different in their structure. Table 1 lists the basic statistics on the analysis of the two web sites. Both sites were analyzed on a 1000MHz Pentium 3 with 512MB of RAM, running Red Hat Linux.

The figures in Table 1 show that a substantial amount of cloning is present in both sites; in the case of the SGPS web site, almost every potential clone (649 out of 686) had a "cousin" somewhere else on the site—only 37 potential clones (5.4% of the total number) were actually unique. The TXL web site had somewhat fewer clones: out of 170 potential clones, 130 were cloned somewhere on the site. 23.5% of potential clones were unique. This suggests that HTML features we've designated as potential clones (form and table tags) are indeed heavily cloned. These numbers,

however, don't represent the overall cloning ratios for these web sites. The mismatch occurs because potential clones we select are more likely to be cloned than a random line in a web site. We also don't look at the entire web site, but focus exclusively on the potential clones. Overall cloning ratios for these web sites are likely to be less dramatic.

The processing times for the two web sites show that the number of potential clones, rather than the number of lines of code, determines the time it takes for analysis to run. Even though the sizes of the web sites are approximately the same, the SGPS web site took 25% longer to analyze. These results also demonstrate the effectiveness of our clone-class optimization: while the number of potential clones in the SGPS web site is four times the number of potential clones in the TXL web site, the analysis time only increases by a minute, instead of quadrupling. Because of the high cloning percentage in the SGPS web site, the potential clone list is trimmed very quickly after the first few rounds of comparisons, and the overall number of comparisons required remains low.

Clone groups. The clones found on both sites fall into several distinct groups. The first, and most prominent, consists of interface clones—HTML code that implements the layout and "look-and-feel" of the site. The clone classes of such clones contain many members, usually one per every web site page. The clones themselves tend to be medium-to large-sized, typically exceeding 20 lines of code. These are particularly prevalent in the SGPS web site, which has a left-hand navigation bar that is essentially the same on all the pages.

Clones that implement a certain specific layout or formatting style make up another sizeable group. Both web sites have several such "style groups". For example, the TXL web site employs a consistent way of presenting download links all across the site, and the code for formatting a link is re-used throughout the site. These clones also tend to be numerous—there can be literally dozens of these on some pages—and relatively small in terms of line count. Style clones are concentrated (at least in the web sites we analyzed) on one or a few pages, e.g. the downloads page.

Cloned pages makeup the third clone group of note. These are entire HTML pages that have been reused by the webmaster. They typically share both interface and page structure; only the content has been altered. Clones in this group are typically large or very large, but each clone class has only two or three members in it—perhaps because the web masters of both sites we analyzed don't tend to clone entire pages extensively.

The fact that clones seem to fall into several distinct groups suggests that they can be successfully resolved, and that resolving them will improve site maintainability and structure. Style clones, for example, could be re-implemented as Cascading Style Sheets (CSS). This will allow site maintainers to apply the style by adding a single attribute to a particular tag, rather than copying over all formatting code. As an added benefit, the style will be stored in one place (the CSS file), which will ensure that it is consistent throughout the web site, and can be changed throughout the web site by changing the CSS file.

Statistic	SGPS Website	TXL Website
Lines of code	10378	9436
Potential Clones	686	170
Comparisons made	2543	883
Clones found	649	130
Clone classes found	37	24
Analysis time	5 min. 2 sec.	3 min. 42 sec.

Table 1: Cloning and performance statistics for the SGPS and TXL web sites.

5 RELATED WORK

There have been several attempts to deal with inexact or near miss clones. Lexical clone detection tools for near-miss clones [2,3] treat them as any sequence of strings that are the same with respect to a certain parameter, or parameters. These tools differ from our work because of their purely lexical approach, which often detects clones that do not correspond to the structural elements of the language.

Davey [13] uses a radically different method to detect near-miss clones of procedures. Davey identifies four classes of clones, I. identical, II. identical except for formatting, III. identical except for slight changes to specialize, and IV. independently semantically equivalent. Like Davey's, our method does an excellent job on the first three classes but has little to say about the fourth. Our much simpler method avoids the long training times and complex methodology

described by Davey while efficiently yielding comparable results. Like ours, Davey's method can also be said to be language independent, but requires new training and tailoring of the method for each new language.

Kontogiannis et al [14] approach clone detection as a concept analysis problem, using dynamic programming techniques to pattern-match similar code fragments given either exemplar source code or abstract concept descriptions. Markov models are used to compute similarity measures based on the the likelihood that a given code fragement can implement an abstract description. Kontogiannis reports good results using this method to identify clones at the procedural and abstract data type level in large scale software systems. Our method differs in being much simpler and in handling clones at many levels of abstraction, including those well below procedure level, while obtaining similar results. Kontogiannis can in theory be language independent but in practice requires significant change from language to language because of its dependence on source code understanding.

Code metrics have also been used as a method to find approximate clones [15]. In Mayrand's method, metrics extracted from source code using Datrix [19] are used to characterize functions to determine their similarity. By experimenting with thresholds in the metrics, many near clones in a large telecommunications software system were identified. Our work differs in that it is not restricted to functions, that it is language independent, and that the method is based on direct similarity of the code rather than indirect metrics.

Komondoor and Horwitz [16] have used slicing methods to identify cloned code sequences and extract procedures for them ("code compaction"). Using a slicing technique, their method can find non-contiguous code sequences, including those in which statements have been reordered or intertwined. Unlike our method, their method does not depend on any structural similarity. However, it also does not address structural clones, and does not detect near-miss clones.

Both Krinke [17] and Chen et al. [18] have described dependence-graph based techniques for the related problem of code compaction that take into account both syntactic structure and data flow. These methods have many advantages, especially for embedded systems, but can be expensive for large programs. Neither method offers good near-miss clones.

6 FUTURE WORK

The clone detection system described in this paper, as it exists now, is a proof of concept for our approach to clone detection. It demonstrates that the approach is able to find meaningful clones in real-world applications with reasonable speed. The system is not finished however; here are several major research directions we intend to pursue in the future.

Near-miss cloning test. As we mentioned in the beginning of this paper, a near-miss clone is an imprecise concept, and one needs to give it a concrete definition before building a detector tool. In this paper, we defined near-miss clones as code structures that share 70% or more of their code on a line-by-line basis. While this definition is a good first approximation and produces useful results, it is not a good long-term answer to the question of "What is a near-miss clone?" Therefore, one major direction for our future research is working towards a better test for near-miss cloning.

Measuring the quality of tests for near-miss clones is also a difficult and rather subjective task. We will define the quality metric in terms of the analyst using the clone detection system. A good metric is a metric that, when used, finds and groups the clones in a way the analyst would expect them to be grouped, and does not detect spurious clones, i.e. does not mark as clones items which are not clones in the opinion of the analyst. Because the opinions of different analysts will likely differ, the metric will have to be tunable, i.e. incorporate a number of parameters to alter or adjust its performance. The metric we currently use can be tuned to a limited degree by changing the cloning threshold used; this is a feature we want to keep and extend further.

Clone resolution. We want to implement user-guided clone resolution as an extension of our tool. Fully automated resolution of near-miss clones is unlikely to be beneficial in our applications, because the resolution method will have to be parameterized to handle significant differences between clones; it will therefore be quite complex itself. If performed without the guidance of an experienced user, such resolution is likely to do more harm than good, because it will introduce more complexity than it will remove.

Instead, we want to expand the functionality of the result-display web site we currently generate, to enable it to serve as an interface to the clone resolution routines. The web site will let the user select the clones that need to be resolved and pick

various options related to the resolution. The resolution itself will then be done automatically, without further user involvement.

An issue closely related to clone resolution is the addition of search functionality to the generated web site. The sheer number of clones present in an average system makes searching abilities necessary, because the user will be expected to examine a significant number of clones to select the ones in need of resolution. To enable users to select clones effectively, the web site will have to provide tools to view clones selectively, by origin, size, type (near-miss or exact), and other criteria.

7 CONCLUSION

This paper presents an approach to detection of near-miss clones. Through a multi-stage process involving extraction of potential clones, comparisons of the potential clones, and report generation, the system described here can quickly and reliably detect exact and near-miss clones in the source code being processed.

The approach combines both compiler-based and lexical techniques in its clone searching. The extraction of potential clones is the only language-dependent stage in the system. By extracting potential clones from analyzed source, it reduces the workload for future stages, and ensures that all clones found are meaningful constructs in the language currently under analysis. Switching between extractors is easy, and the approach can therefore be adapted to various languages.

Comparisons between potential clones are performed with the Unix *diff* utility. The use of lexical comparison tools means that the bulk of the system is language-independent, fast, and simple to maintain and change.

About the Authors

Nikita Synytskyy is a research associate at the Computer Science Department at the University of Waterloo, Ontario, Canada. His research focuses on analysis, design recovery and transformation of web application source code. He holds a Masters degree from Queen's University and a Bachelor of Commerce degree with Honors from the University of Ottawa. Before starting his graduate studies, he worked as a software developer at a variety of organizations, including International Datacasting Corporation, Corel Corp. and Environment Canada.

James Cordy is the Director of the School of Computing and Professor of Computing and Electrical and Computer Engineering at Queen's University. From 1995 to 2000 he was Vice President and Chief Research Scientist at Legasys Corporation, a software technology company specializing in legacy software system analysis and renovation. Dr. Cordy is a founding member of the Software Technology Laboratory at Queens's University and winner of the 1994 ITRC Innovation Excellence award and the 1995 ITRC Chair's Award for Entrepreneurship in Technology Innovation for his work there. He serves as program committee member and chair for international conferences and workshops in system analysis and maintenance, such as ICCL'92, ICSM'02, SCAM'02, ICSM'04 and IWPC'05

Thomas Dean is an Assistant Professor of Electrical and Computer Engineering at Queen's University and an Adjunct Associate Professor at the Royal Military College of Canada. His background includes research in air traffic control systems with the RMC and 5 1/2 years of experience in source code analysis and transformation as a Senior Research Scientist at Legasys Corp.

References

[1] Ira D. Baxter, Andrew Yahin, Leonardo Moura, Marcelo SantAnna, Lorraine Bier, "Clone Detection Using Abstract Syntax Trees", *Proceedings of International Conference on Software Maintenance*, November 1998.

[2] Brenda S. Baker, "On Finding Duplication and Near-Duplication in Large Software Systems", *Working Conference on Reverse Engineering*, 1995.

[3] Toshihiro Kamiya, Shinji Kusumoto, Katsuro Inoue, "CCFinder: A Multilinguistic Token-Based Code Clone Detection System for Large Scale Source Code", *IEEE Transactions On Software Engineering,* Vol. 28, No. 7, pp 654-670.

[4] G. Antoniol, U. Villano, M. DiPenta, G. Casazza, E. Merlo, "Identifying Clones in the Linux Kernel", *Proceedings of International Workshop on Source Code Analysis and Manipulation*, November 2001.

[5] Pearl Brereton, David Budgen and Geoff Hamilton, "Hypertext: The Next Maintenance Mountain", *IEEE Computer*, Vol. 31 No. 12, pp 49-55, 1998.

[6] T.R. Dean, J.R. Cordy, A.J. Malton and K.A. Schneider, "Agile Parsing in TXL",

Journal of Automated Software Engineering 10,4 (October 2003), pp. 311-336.

[7] James R. Cordy, Ian H. Carmichel, Russel Halliday, *The TXL Programming Language, Version 10*. TXL Software Research Inc. January 2000.

[8] N. Synytskyy, J.R. Cordy and T.R.Dean, "Resolution of Static Clones in Dynamic Web Pages", *Proc. WSE 2003, IEEE 5th International Workshop on Web Site Evolution*, Amsterdam, September 2003, pp. 49-58.

[9] J. W. Hunt, M. D. McIlroy, *An algorithm for differential file comparison*, Technical Report #41, Computing Science, Bell Laboratories, 1976

[10] J. W. Hunt, T. G. Szymanski, "A fast algorithm for computing longest common subsequences," *Comm. ACM* 20,5, May 1977, pp. 350-353.

[11] Cornelia Boldyreff, Richard Kewish, "Reverse Engineering to Achieve Maintainable WWW sites", *Proceedings of Eighth Working Conference On Reverse Engineering*, October 2001.

[12] M. Synytskyy, J.R. Cordy, T.R. Dean, "Robust Mutilingual Parsing Using Island Grammars", *Proceedins CASCON 03, IBM Centre for Advanced Studies 2003 Conference*, Toronto, Ontario, November 2003, pp. 149-161,

[13] N. Davey, P. Barson, S. Field, R. Frank and D. Tansley, "The development of a software clone detector", *International Journal of Applied Software Technology* 1,3-4, pp. 219-236, 1995

[14] K. Kontogiannis, R. DeMori, E. Merlo, M. Galler, M. Bernstein, "Pattern Matching for Clone and Concept Detection", *J. Automated Software Engineering* 3,1-2, pp. 77-108, 1996.

[15] J. Mayrand, C. Leblanc and E. Merlo, "Experiment on the automatic detection of function clones in a software system using metrics", *Proc. ICSM'96, International Conference on Software Maintenance*, pp. 244-254, 1996.

[16] R. Komondoor and S. Horwitz, "Using slicing to identify duplication in source code", *Proc. 8th International Symposium on Static Analysis (SAS),* pp. 40-56, 2001.

[17] J. Krinke: "Identifying Similar Code with Program Dependence Graphs", *Proc. 8th Working Conference on Reverse Engineering (WCRE 2001)*, Stuttgart, Germany, pp. 301-309, 2001.

[18] W-K. Chen, B. Li, and R. Gupta, Code Compaction of Matching Single-Entry Multiple-Exit Regions, *Proc. 10th Annual International Static Analysis Symposium*, San Diego, pp. 401-417, 2003.

[19] Bell Canada Inc., *DATRIX Abstract semantic graph reference manual*, version 1.2, Montreal, Canada, July 1999.