Resolution of Static Clones in Dynamic Web Pages

Nikita Synytskyy, James R. Cordy, Thomas Dean
School of Computing, Queen's University
Kingston, Ontario, Canada K7L 3N6
nikita@mondenet.com, cordy@cs.queensu.ca, thomas.dean@ece.queensu.ca

Abstract

Cloning is extremely likely to occur in web sites, much more so than in other software. While some clones exist for valid reasons, or are too small to eliminate, cloning percentages of 30% or higher—not uncommon in web sites—suggest that some improvements can be made. Finding and resolving the clones in web documents is rather challenging, however: syntax errors and routine use of multiple languages complicate parsing the documents and finding clones, while lack of native code reuse tools forces the analyst to rely on other technologies for resolution.

Here we present a way to find clones in multilingual web documents, and resolve them using one of several code reuse techniques that are available in a dynamic web site. Rather than picking a single resolution technique and relying on it exclusively, we pick it based on the clone in question, to minimize disruption to the structure of original documents.

1 Removing Clones From Web Pages

Previous research indicates that all software contains cloned code[1, 4]. Web sites and web applications contain a higher proportion of cloned code than other software—on average duplicated code amounts to 5-15% of the total amount of code in an application[9, 10], whereas cloning ratios of 30% and higher are not uncommon for web sites[3]. The main reason for high cloning ratios in web documents is HTML's lack of code reuse tools. Boldyreff and Kewish point out[3] that HTML lacks even an "include" directive, which is available in many programming languages. Therefore developers are forced to reuse code by cloning simply because no other alternative is available—if a piece of content must appear on several pages, the only way to do it is to place a copy in every page.

As Brereton *et al.*[11] point out, HTML lacks many abstractions employed by other languages to ease maintenance and promote code reuse. The concept of a library does not exist. While links to other sites may serve as a surrogate, the absence of flow control tools and use of parameters inhibits the ability to reuse code. The concept equivalent of encapsulation is absent, and separation of page tags from page text is difficult to achieve. Since both encapsulation and use of parameters are absent, HTML lacks any equivalent of a subroutine i.e. a potentially parameterized code unit that can be reused as needed. Emergence and popularization of Cascading Style Sheets (CSS) has contributed towards a solution of these problems, but by no means eliminated them.

Reducing the number of clones in a web site has undeniable benefits for its maintainability: the size of software to maintain is reduced, and update anomalies that clones necessarily cause are eliminated. Furthermore, related work suggests[8] that as purely static sites are migrated to dynamic platforms, both the opportunities and the needs to identify and eliminate duplicated content are present. At the same time, removing clones has the potential to make software worse by disrupting locality and linearity, which are important for code readability and understandability[12]. Overzealous clone elimination has the potential to turn software into spaghetti code. Clone resolution must be done with extreme care to be successful, and clones must be resolved with proper methods for software to remain understandable afterwards.

To attack this problem, we have developed a method of automated clone resolution that is suitable for web sites. In the absence of native code reuse methods, we rely on a number of tools that are provided either by languages that are used alongside HTML—typically server-side scripts—or by software that handles HTML on its way from source to destination—web servers and browsers. Rather than relying on one method exclusively to resolve all clones, we attempt to match the clones being resolved to the method in use, so that each clone is resolved using the method most suited for it. This matching minimizes the disruption made to the structure of the original web site, so that the resulting code is still familiar to its maintainers. After the clone res-

olution process is done, it is still be possible to maintain the site by hand.

Structure of the paper. Section 2 presents the background for this work. Section 3 briefly discusses our approach to robustly parsing multilingual web documents. Section 4 describes the clone detection method we use. Section 5 explains our method of clone resolution and shows a walk-through of a resolution process applied to a simple example. Section 6 discusses practicality and scalability of our work. Finally, Section 7 shows some future work directions, and Section 8 concludes the paper.

2 Background

A large amount of work has already been done in the area of clone research in general, and in web clones in particular.

The work that is most closely related to ours is by Boldyreff and Kewish[3], who introduce a system for reducing duplication in web sites. They propose to store components of web site pages in a relational database, with each component being stored only once, and instrumenting original pages with scripts to retrieve appropriate content from the database. This approach has the potential to remove the highest proportion of clones, at the expense of disrupting its structure. Modifying the pages by hand after it has been processed is unadvisable, and all changes have to be made to the elements stored in the database.

Ricca and Tonella[8] present a related system for identifying web pages with similar structure for creation of a page template for later dynamic generation of individual pages. They rely on clustering for identifying similar pages, and store details of individual pages in a database, relying on server-side scripts to assemble the final page from its components.

Baxter *et al.*[1] present a way to locate clones and sequences of clones in a C program, and eliminate them with preprocessor macros. Their method of clone detection relies on detecting similarities in parse trees built from the code, and performing additional analysis on them to find sequences of clones. The paper also presents solutions to the problem of finding and removing sub-clones, which is often encountered in clone detection efforts, and some optimization techniques to make the task of finding and resolving clones tractable in a large system. Some of the techniques described could be used to improve our clone detection method.

Ducasse *et al.*[2] present an approach to clone detection that does not rely in parsing, and is therefore easily adapted to any language. Similarities in code are detected by string-matching with some initial preprocessing to remove comments and whitespace. Meaningful clone resolution, however, is difficult to achieve in a language-independent way because it is hard to guarantee that clones found represent a cohesive unit in the language being analyzed. The process of resolution itself also depends on the language in question.

3 Parsing Web Applications

Any effort directed at automated maintenance or modification of software must be preceded by its understanding by the computer; this generally means parsing of the source code. The code that makes up web applications presents several challenges to parsing. First of all, the code is often multilingual. While applications written in several languages are certainly nothing new, multilingualism is taken to a new level by web applications. A single file can contain code written in several languages. These languages are intertwined, sometimes to the point where statements in one language appear as part of statements of another language. Second parsing obstacle that web documents routinely present is the presence of syntax errors. HTML frequently contains errors, and does not usually conform to any of the published specifications for the language. Because web browsers nevertheless consider such code valid and process it, any analysis effort directed at web documents must do the same.

To analyze web documents despite these difficulties, we have developed an approach to robustly parsing several languages simultaneously into a single parse tree. The approach is based on the use of island grammars to describe multiple languages while at the same time using their features to deal with errors in the code. Island grammars have been used for partial source code analysis in the past [5, 7] and are described extensively by Moonen[6]. Island grammars view input as belonging to one of two categories—interesting “islands”, and uninteresting “water”; these can be nested, i.e. an island might contain a lake, which will in turn contain another island, and so on. Definitions of islands, which are the focus of the analysis, are usually quite restrictive. Definitions of water productions, on the other hand, are liberal enough to describe any uninteresting content the input might contain. By tuning individual island and water definitions, it is possible to select from the input only the productions of interest, while ignoring everything else—including syntax errors, if the definition of water is liberal enough.

The island parsing paradigm lends itself well to simultaneous parsing of multiple languages. Our grammar contains one island type for each language. The concept of nesting of islands and water can be used to process instances of language nesting, and since the islands can be arbitrarily small, even the most closely intertwined languages can be processed. Finally, syntax errors in languages, together with any productions which are of no interest in current analysis can be treated as water. Water is not acted upon during the analysis, but preserved in the final output, so that content

which is not understood by the analyzer is not changed by it.

4 Clone Detection

Figure 1 shows the overall conceptual architecture of our system. This section describes the functioning of the clone detector. Since the primary focus of this work is on clone resolution rather than detection, we used a simplistic detection algorithm. The issues of finding the clones and of doing something with them once they are found—such as resolving them—are for the most part orthogonal. A more sophisticated clone detection algorithm, such as the one proposed by Baxter *et al.*[1] may be inserted later with minimal disruption to other parts of our system.

A single island is the smallest clone we are willing to consider. All islands in our grammar correspond to sizeable HTML constructs, such as tables and forms, which minimizes the likelihood that insignificant clones only a few characters long will be resolved. HTML `<script>` tags are one type of island the grammar recognizes, which allows us to find and resolve JavaScript clones.

The clone detection algorithm performs its task in three major steps—list building, comparisons, and list trimming.

List Building. All islands contained in the files to be analyzed for cloning are extracted into a master island list. The list contains each island found in the files exactly once, except for islands that are nested inside other islands (e.g. a table nested inside another table). These are listed several times—once on their own, and once as part of their parent island. This permits us to find the clones of both the parent and the child islands.

Comparisons. To find the clones, all files under analysis are searched for every item that occurs in the list. Whenever an island is found in a file that is different from the file of its origin, the conclusion is made that the island is cloned across several files, and is annotated as such.

List Trimming. Finally, the resulting clone list is trimmed to remove unnecessary items. These items can be of two kinds:

- Items that occurred only once, and are therefore not clones;
- Items that are children of other clones in the list, i.e. sub-clones.

The result of algorithm execution is a list of cloned islands, each one annotated with a list of files in which the clone has been found. This list serves as a starting point for our clone resolution efforts.

5 Clone Resolution

HTML provides no native code reuse facilities, making clone resolution in HTML extremely difficult. It is however possible to reuse code by relying on tools provided by other languages, or by servers which are used to serve HTML pages to the browsers.

5.1 Resolution methods

The three techniques which are employed in this paper—all commonly used—are described below.

Server side includes. The simplest, and the most generally applicable method of code reuse on a web site relies on a feature called server-side includes. The includes are not part of HTML; rather, they are a feature provided by web servers that deliver the site to the browsers. The includes are very similar to C-style `#include` statement. A server side include statement can appear anywhere within a web page, and looks like this:

```
<!--#include file="foo.html" -->
```

The functionality of the command is rather straightforward: the content of `foo.html` is read by the server and inserted in the place of the include statement before the file is sent to the web browser. Nearly all web servers support the server side include feature. Because server side includes are processed before any other instructions a web page might contain, they are the most general of all clone resolution techniques—they can be used to resolve any clone.

Executable subroutines. When HTML is being used in a dynamic web page, it usually appears in conjunction with a programming language, such as Visual Basic (VB) or Java. In this case it is possible to rely on code reuse features provided by these languages. All discussion that follows talks about using Visual Basic, but the concepts apply to most other programming languages as well. It is possible, for example, to define a Visual Basic subroutine which prints the cloned HTML and does nothing else. All occurrences of the clone are then replaced by a call to that subroutine. Server-side executable code is entirely transparent to browsers, so VB subroutines can be used to resolve any static clone.

This method may seem almost identical to using server includes, but this is not really the case—the use of VB subroutines for clone resolution has several advantages over server-side includes.

When using the includes, every clone has to be housed in a separate file, which is the file that is included by the server. There is no such restriction on clone-resolving VB subroutines—they can be all contained in one file, or organized into an arbitrary number of files according to some criterion. This affords greater flexibility in performing the

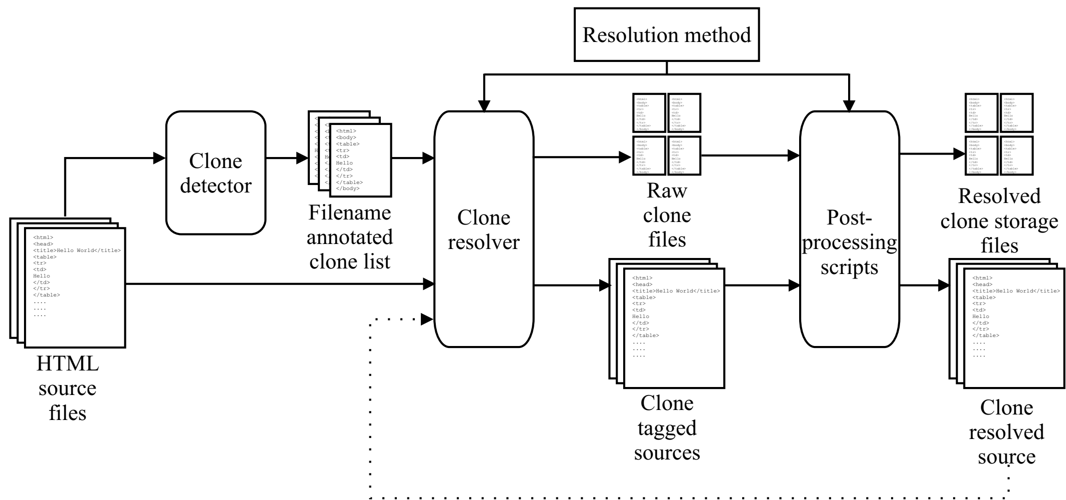

Figure 1. Conceptual architecture of the clone resolution system.

clone resolution, since the clones may be organized by function, size, content, or other criterion.

VB subroutines are the only resolution method discussed here that can resolve near-miss clones, i.e. clones that are the same except for some small difference. A clone-resolution subroutine can be parameterized, so that the needed clone will be generated from a generic template depending on the value of the parameter. The template can be held in the routine itself, or read from a file on the server. Server-side includes, on the other hand, can not resolve near miss clones because they do not support parametrization.

The price for this versatility is extra work that has to be done during page serving. Unlike inclusion of a file, which is a relatively simple operation, the routines have to be executed by the server before being sent to the browsers. The precise mechanics involved in execution differ from language to language, but all are more complex and time-consuming than the handling of a server-side include.

External script files. HTML does provide the tools to reuse existing client-side scripts—including JavaScript—which means that clones containing nothing but JavaScript can be resolved in yet another way, not available for HTML in general.

One way to include JavaScript code in an HTML page is to embed it in that page. In that case the code is available for execution only to the page it is embedded in. An equivalent, but superior from the point of view of code reuse, method is to house the script in a separate file, and include a reference to it in the HTML file. The reference takes the form of an empty `<script>` tag, with a parameter to indicate where to take the source code of the script from. The functionality remains the same, but since the script now exists as a separate entity, it is possible to re-use a single instance of it across multiple HTML pages, thus reducing the amount of duplication in a web site. Unsurprisingly, this clone resolution method only works for cloned client-side scripts.

Use of resolution methods. In principle, either the server-side includes or the VB subroutines can resolve all static clones found. It would be better, however, to match up resolution methods and clones, so that for each clone resolved, the most appropriate method is used to resolve it. This will minimize the inevitable disruption to code locality and linearity that clone resolution must make to do its work.

5.2 Clone resolution algorithm

The key to achieving the best match between the resolution method and the clone it is used to resolve is to perform several passes on the files being analyzed, resolving some clones during each pass. During each pass, only the clones that match the resolution method are resolved, while the other ones are skipped over.

We employ a two-pass approach. We use clone resolution via external script files as our method of resolution for the first pass, and resolution via Visual Basic subroutines for the second pass. The external script files method is the least general of the two—it is able to resolve only JavaScript clones; it is also the most appropriate method for resolving such clones, because that is its express goal. By employing it first, we can make sure that all JavaScript clones are resolved using only this method—after the first pass there will be no JavaScript clones left for other methods to resolve.

In this application, we limit ourselves to only two passes, because Visual Basic subroutines can resolve all remaining clones; in fact, server side includes could have been used

advertisement.html

```
<html>
<script>
  dNow = new Date()
  document.write(dNow.toString())
</script>
<table border=1>
  <tr>
    <td>
      Welcome to SnowStorm, Inc.
    </td>
  </tr>
</table>
Buy Our Products!!
</html>
```

contact.html

```
<html>
<script>
  dNow = new Date()
  document.write(dNow.toString())
</script>
<table border=1>
  <tr>
   <td>
      Welcome to SnowStorm, Inc.
    </td>
 </tr>
</table>
Contact us by e-mail!
</html>
```

Sample input files to the clone resolver system. A simple JavaScript program and a table with a welcome message are cloned across these two files.

Figure 2. HTML files with cloned content.

in their place—both are appropriate enough. Depending on the situation, more than two passes could be used; for example, if near-miss clones were being resolved, three passes could be employed. The first, relying on external script file resolution, would handle JavaScript clones; the second, using server-side includes, would resolve the identical clones; finally, VB subroutines could be used to resolve near-miss clones.

The algorithm used to resolve the clones is the same for every pass. As its input, the algorithm takes an annotated clone list generated by the clone detection algorithm, a list of files to resolve the clones in, and a parameter that specifies in what way the clones will be resolved—via server-side includes, Visual Basic subroutines, or JavaScript importing.

Consider two files shown in Figure 2. The files contain two clones: a JavaScript program for printing the date, and an HTML table with a welcome message. Similar (although considerably larger and more complex) files can be found on web sites of many companies. In analyzing these two files, the first pass of our clone resolution scheme would receive as input the filename annotated clone list, consisting of two clones, *annotated* with the files they occur in:

```
<script>
  dNow = new Date()
  document.write(dNow.toString())
</script>
```

advertisement.html, contact.html

and

```
<table border=1>
  <tr>
    <td>
      Welcome to SnowStorm, Inc.
    </td>
  </tr>
</table>
```

advertisement.html, contact.html

The resolver algorithm will also receive the list of HTML source files under analysis: *advertisement.html, contact.html*, and a parameter telling it to employ external script files for clone resolution.

As a first step, clones which are not worth resolving, or which can not be resolved via the chosen method are removed from the clone list. In this case, the second clone in the list (i.e. the table with the welcome message) will be removed from the list and not considered further, because it is impossible to eliminate this clone with the chosen resolution method. The only kind of clone we consider not worth resolution at the moment is a cloned `<script>` tag without a body. These tags are in fact references to external scripts, and can hardly be treated as clones.

Each clone is then given a unique identifier, a "name" to distinguish it from another clones. Currently the names carry no further meaning and take the form of `unique_id_1`, `unique_id_2` and so on. The input files are then searched for occurrences of the cloned islands, and each occurrence is replaced with an HTML-like clone identifier tag, with the name of the clone it replaced as a parameter. The resulting files are shown as "Clone tagged sources" in Figure 1. In our example the entire script tag will be replaced by the following:

advertisement.html	*contact.html*
<html>	<html>
<script src="unique_id_1.js">	**<script src="unique_id_1.js">**
</script>	**</script>**
<table border=1>	<table border=1>
<tr>	<tr>
<td>	<td>
Welcome to SnowStorm, Inc.	Welcome to SnowStorm, Inc.
</td>	</td>
</tr>	</tr>
</table>	</table>
Buy Our Products!!	Contact us by e-mail!
</html>	</html>

The analyzed files after the first pass through the clone resolution system. Cloned JavaScript code has been resolved. The newly inserted lines used for resolution are highlighted in bold.

Figure 3. HTML files with JavaScript clone resolved.

```
<clone_identifier "unique_id_1">
```

Each clone in the clone list is then written to a file; these correspond to "Raw clone files", as pictured in Figure 1. In case of clone resolution by Visual Basic subroutines, all clones are written to one file, and the necessary text is later added to make clones into separate subroutines. The clone names are then re-used as the names of the subroutines. For resolution by server-side includes or—as in our example—JavaScript importing, each clone is written out to a separate file; in this case clone names are used to name the files that hold the clones. JavaScript files are further modified to remove the unnecessary

```
<script> ... </script>
```

tags. In our example, only one file will be generated. It will be named `unique_id_1.js` and will contain the code of the JavaScript program the cloned `<script>` tag originally contained:

```
dNow = new Date()
document.write(dNow.toString())
```

Finally, the analyzed files are searched for the clone identifier tags like the one above, and each tag is replaced by a call to a subroutine, a server-side include statement, or—as in this case—a script tag importing the appropriate script file.

Ease of implementation is the reason for using intermediate clone identifier tags and not replacing clones with their replacements directly. The replacement step is done with `sed`, a Unix tool for automated text editing. These scripts are lexical in nature and do not have to follow the grammar being used to parse and analyze the input, and therefore encounter fewer restrictions in modifying the files.

After the first pass of the algorithm, the JavaScript clone has been resolved, while the "welcome table" has remained. Figure 3 shows the results of this initial algorithm application.

The second application of the algorithm follows the same general execution pattern, except that the algorithm is instructed to use VB subroutines as its resolution tool. As Figure 1 shows, the clone resolved source from the first application is used as the new input to the clone resolver. The clone list received from the clone detection algorithm this time contains the "welcome table" and the body-less script tag that replaced the original cloned script.

```
<script src="unique_id_1.js">
</script>
```

The script tag is eliminated from the clone list as being too small to resolve, and the algorithm proceeds with resolution of the cloned table. The table is again issued a unique name, and then eliminated from all the files in which it occurs, replaced by a pseudo-HTML tag with a reference to the clone name. All the clones—in this case, only one—are written out to a file `subroutines.vb`. The file is modified to change it from a simple clone listing to a sequence of Visual Basic subroutines; the clone names are re-used as names of generated VB subroutines.

```
<% sub unique_id_1  %>
<table border=1 >
  <tr>
    <td>
      Welcome to SnowStorm, Inc.
    </td>
  </tr>
</table>
<%  end sub %>
```

advertisement.html	*contact.html*
<!--#include file="subroutines.vb"-->	**<!--#include file="subroutines.vb"-->**
<html>	<html>
<script src="unique_id_1.js">	<script src="unique_id_1.js">
</script>	</script>
<% unique_id_1 %>	**<% unique_id_1 %>**
Buy Our Products!!	Contact us by e-mail!
</html>	</html>

The final output files with all clones resolved. The newly inserted code that implements the resolution of the "welcome table" is highlighted in bold.

Figure 4. HTML files with all clones resolved.

Finally, all the files in which the clone has occurred have the temporary clone ID tags replaced by calls to the appropriate subroutines, and an include statement is added to the beginning of each file to make the subroutines available for use. Figure 4 shows the final result of the second, and final, iteration of the algorithm; all the found in the originals were removed.

6 Practicality and Scalability

Our system performs well for small inputs, taking approximately one minute to complete the analysis and clone resolution of a 5–7 page subset of a student pre-registration web site. Analysis of smaller examples, like the one shown in this paper, is nearly instantaneous. Because our naive detection algorithm compares every potential clone to every other, the performance is quadratic in the number of potential clones. The performance can be significantly improved by using performance optimizations such as pre-sorting or hashing of potential clones. The resolution algorithm on the other hand is linear in the number of detected clones, and should scale well to larger tasks.

7 Future Work

Improved clone detection. The most promising way to improve the quality and usefulness of our program is to switch to a better clone detection algorithm. While the algorithm in use now works, it is not efficient enough to work on sites that consist of more than a dozen or so pages. Optimizing the steps of clone detection and deletion of redundant sub-clones can significantly boost the performance of the algorithm, thus putting larger, more realistic web sites within its reach.

Near-miss clones. Our program only finds and resolves identical clones. A good number of clones are in fact not identical, but similar, differing by only a few features. A good example of such a near-miss clone would be a navigation bar of a web site that displays the link to the current page in a different color. The navigation bar is substantially the same across all the pages of a site, with the color of individual links being the only difference. Currently, our program would not consider the navigation bar a clone. An interesting and useful addition to our program would be an ability to resolve such a near-miss clone, perhaps by storing a generic template for it in a file, and replacing all instances of the clone with a call to a VB subroutine, which would recreate the needed version of the clone based on the template and the place it was called from.

Clone evaluation. Not all clones in a web site are worth resolving. A substantial number of the clones are too small, or too insignificant to be of interest, and their resolution would only complicate the existing structure of the site unnecessarily. Size a good first indicator of whether a clone is worth resolving, but it is by no means perfect. Large clones may not be worth replacing if they are only cloned a few times; and clones only a few lines long are worth resolving if they are cloned extensively and are important to the site's functionality.

The difficulty in finding the answer to the question "What is a worthwhile clone to resolve?" is that it is to a certain degree subjective, and varies from site to site and from developer to developer. And yet, answering this question even vaguely would benefit clone resolution in two important ways. The first and most important benefit would be the ability to detect, and remove from consideration, inconsequential clones—the clones that for some reason are not worthwhile to resolve. This will reduce the amount of computational effort required, because uninteresting clones won't be analyzed, and minimize the disruption that clone resolution causes, because they will not be resolved.

An interesting aspect of clone evaluation is the task of clone naming. Our present clone naming technique only ensures that names given to clones are unique—they serve as identifiers only, and carry no information about the clone. This technique can be improved by letting a human analyst view and name the clones after they are found but before

any resolution efforts begin. As the size of web site under analysis grows, however, the amount of clones becomes too large for a human to name and review, so automated evaluation remains a necessary first step.

8 Conclusion

This paper presents a system for resolution of static clones in web sites. Unlike most web-oriented clone resolution efforts to date, our system does not rely on one method of clone resolution exclusively. Instead, it uses a multi-pass approach to resolve clones incrementally, using several different resolution methods, resolving each clone encountered with the most appropriate resolution method available. The aim of the matching efforts is to minimize disruptions to the structure of the original HTML files being analyzed, thus reducing the negative effects of clone resolution as much as possible.

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] Stephane Ducasse, Matthias Rieger, Serge Demeyer, "A Language Independent Approach For Detecting Duplicated Code", Proceedings of International Conference on Software Maintenance, August-September 1999.

[3] Cornelia Boldyreff, Richard Kewish, "Reverse Engineering To Achieve Maintainable WWW Sites", Proceedings of Eighth Working Conference on Reverse Engineering, October 2001.

[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-December 2001.

[5] A. van Deursen, T. Kuipers, "Building Documentation Generators", Proceedings of International Conference on Software Maintenance, August-September 1999.

[6] Leon Moonen, "Generating Robust Parsers Using Island Grammars", Proceedings of Eighth Working Conference On Reverse Engineering. October 2001.

[7] Leon Moonen, "Lightweight Impact Analysis Using Island Grammars", Proceedings of Tenth International Workshop On Program Comprehension. June 2002.

[8] F. Ricca, P. Tonella, "Using Clustering to Support the Migration from Static to Dynamic Web Pages", Proceedings of International Workshop on Program Comprehension, May 2003.

[9] 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-December 2001.

[10] B.S. Baker, "On finding duplication and near-duplication in large software systems", Proceedings of Second Working Conference on Reverse Engineering, July 1995.

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

[12] Gerald M. Weinberg, "The Psychology of Computer Programming", pp. 229-232, Van Nostrand Reinhold Ltd. New York, New York. 1971.