A Language Independent Approach for Detecting Duplicated Code

Stéphane Ducasse, Matthias Rieger, Serge Demeyer
Software Composition Group, University of Berne
ducasse,rieger,demeyer@iam.unibe.ch
http://www.iam.unibe.ch/∼scg/
Neubrückstrasse 10, CH-3012 Bern
Tel. +41 31 631 49 03, Fax +41 31 631 33 55

Abstract

Code duplication is one of the factors that severely complicates the maintenance and evolution of large software systems. Techniques for detecting duplicated code exist but rely mostly on parsers, technology that has proven to be brittle in the face of different languages and dialects. In this paper we show that is possible to circumvent this hindrance by applying a language independent and visual approach, i.e. a tool that requires no parsing, yet is able to detect a significant amount of code duplication. We validate our approach on a number of case studies, involving four different implementation languages and ranging from 256 K up to 13Mb of source code size.

Keywords: *Software maintenance, code duplication detection, code visualization*

1. Code Duplication Detection

Duplicated code is a phenomenon that occurs frequently in large systems. The reasons why programmers duplicate code are manifold (see [9, 2] for a thorough discussion) and include the following reasons: (a) Making a copy of a code fragment is simpler and faster than writing the code from scratch. In addition, the fragment may already be tested so the introduction of a bug seems less likely. (b) Evaluating the performance of a programmer by the amount of code he or she produces gives a natural incentive for copying code. (c) Efficiency considerations may make the cost of a procedure call or method invocation seem too high a price. In industrial software development contexts, time pressure together with points (a) and (b) lead to plenty of opportunities for code duplication.

Although code duplication can have its justifications, it is considered bad practice. Especially during maintenance (estimated at 70% of the overall effort for producing a software system [16]) unjustified duplicated code gives rise to severe problems: (a) If one repairs a bug in a system with duplicated code, all possible duplications of that bug must be checked. (b) Code duplication increases the size of the code, extending compile time and expanding the size of the executable. (c) Code duplication often indicates design problems like missing inheritance or missing procedural abstraction. In turn, such a lack of abstraction hampers the addition of functionality.

Techniques and tools for detecting duplicated code are thus a highly desired commodity especially in the software maintenance community and research has proposed a number of approaches ([1, 9, 14, 10, 2]) with promising results. However, the application of these techniques in an industrial context is hindered by one major obstacle: the need for parsing. This is clearly stated in the following quote:

> *"Parsing the program suite of interest requires a parser for the language* dialect *of interest. While this is nominally an easy task, in practice one must acquire a tested grammar for the dialect of the language at hand. Often for legacy codes, the dialect is unique and the developing organization will need to build their own parser. Worse, legacy systems often have a number of languages and a parser is needed for each. Standard tools such as Lex and Yacc are rather a disappointment for this purpose, as they deal poorly with lexical hiccups and language ambiguities."* [2].

Most of the approaches [9, 14, 10, 2] are based on parsing techniques and thus rely on having the *right* parser for the right dialect for *every* language that is used within an organization.

To circumvent this problem, we try to answer the question "How can we build a language independent duplication detector and what will it detect?"

Structure of the Paper. In Section 2, we propose a language independent approach based on (a) *simple* string matching, (b) *textual reports* that synthesize the duplication information found, and (c) *scatter plot* visualizations as a helpful means in analyzing duplication.

To validate our claim of language independence, we present in Section 3 the results we obtained with four case studies written in four different languages having diverse syntaxes: C, Smalltalk, Python and Cobol. To show that our approach is able to detect significant duplication, we present the overall percentages of duplication that we found in the different case studies. Then, by visually inspecting some samples of the identified files, we exhibit anecdotal evidence of duplication in cloned files, in files that underwent evolutionary change, and other duplication artefacts. Finally, in Section 7, we discuss related work and future plans.

Note that proposals for duplication removal are not in the scope of this paper. Remedies for the duplication problem are not a trivial topic and deserve attention on their own account.

2. A Language Independent Approach

As we have stressed with the introductory quote, language dependency is a big obstacle when it comes to the practical applicability of duplication detection. We have thus chosen to employ a technique that is as simple as possible and prove that it is effective in finding duplication.

In this section, we detail the principles behind our approach under the three aspects *algorithms* used to compute the comparisons data, *visualization* of the comparison data, and *pattern matching* to condense the data. Figure 1 shows an overview of the steps that take us from source code to duplication data.

2.1. Algorithmic Aspects

Clone detection is always a two-step process. First, source code is transformed into an internal format. Second, a more or less sophisticated comparison algorithm is then performed on the internal data. In our case, the code is only slightly transformed using string manipulation operations. To compare the transformed lines, we use basic string matching.

Source Code Transformation

Two decisions must be made: the nature of the transformation and the size of the source code fragment that will be the entity of the incidental comparison. We choose one line of source code as code fragment entity on which we base our algorithm. The choice is on the one hand motivated by the consideration that the important copy and paste performed by programmers include one or more lines, and on the other hand that prepocessing can be kept simple (see [9] for an approach that uses multiple lines as fragment size). To stay language independent, we refrain from code transformation to more abstract formats like AST's [2] which have to employ parsing, or *parameterized strings* [1] which need at least a lexer. The transformation we apply to a code fragment is minimal and stays in the realm of string manipulation: We remove comments and all white space until we get a condensed form of the line.[1] As an example, the C line

```
if( code & pcObjType ) { /* print type */
```

is condensed to

```
if(code&pcObjType){
```

As a consequence, the code reader which does the transformation is adapted to any new language in a few minutes.

The transformation reduces the entire file to an ordered collection of *effective* lines (see Figure 1) that will be compared against itself and line collections from other files.

[1]The UNIX `diff` utility uses the same condensation technique upon request.

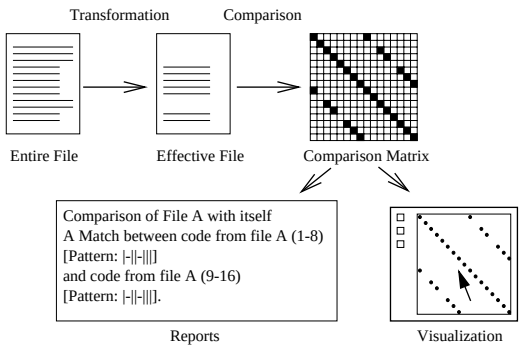

Figure 1. Overview of the approach.

Comparison Algorithm

Since we do not know what to look for, we cannot apply `grep`-like pattern matching algorithm but have to compare every entity (transformed source line) with every other entity. The comparison of two entities is done by string matching. The result is a boolean `true` for an exact match and a `false` otherwise. This value is stored in a matrix (see Figure 1), taking the coordinates that the two compared entities have in their respective ordered collections as the matrix coordinates for the comparison result. Note that after the the comparison process, the matrix only contains matches of individual lines and not yet of whole sequences. They will have to be extracted from the matrix in a separate pass (see section 2.3).

Optimization. The search space spanned by an input of n lines is quite uncomfortable ($\Omega(n^2)$). We thus reduce it by preprocessing the transformed lines a second time: the lines are hashed into B buckets. The string matching is then applied on all possible pairs of one hash bucket. Equal lines have the same hash value and are thus thrown into the same hash bucket, so no false negatives occur. This procedure cuts the processing time by the factor B (the same optimization is used in [2]).

2.2. Visualization

The matrix created by the comparison can be visualized using scatter-plots [6], which were first used by geneticists looking for similar strings of DNA. Such "dot drawings" (see Figure 2) allow immediate recognition of typical situations.

Some interesting configurations formed by the dots in the matrices are the following [6]:

a) diagonals of dots indicate copied sequences of source code (see Figure 2 *a)*). Later in the paper, Figure 5 shows some instances of copied sequences in a Cobol system.

b) sequences that have holes in them indicate that a portion of a copied sequence has been changed (see Figure 2 *b)*).

c) broken sequences with lower parts shifted indicate that a new portion of code has been inserted (see Figure 2 *c)*, *above* the main diagonal), or removed, respectively (Figure 2 *c)*, *below* the main diagonal). Figure 7 on page 8 will be showing an example of how evolutionary change shows up in real code.

d) rectangular configurations indicate periodic occurrences of the same code (see Figure 2 *d)*). An example is the break; at the end of the individual cases in a C/C++ switch statement or recurring preprocessor commands.

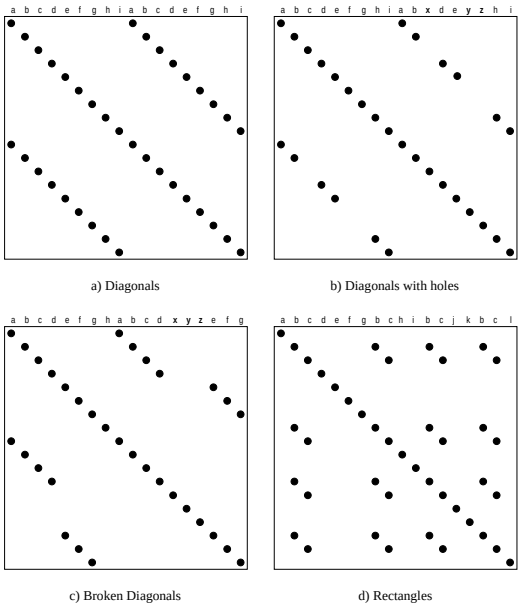

Figure 2. Different Configurations of Dots.

Note that due to the line-based comparison, repeated matches of either structural code elements that occur alone on one line (e.g. the break;) or minimal "idioms" (e.g. the frequent C-line int i;) spread spurious dots all over the matrix. To get rid of this *noise* rather than duplication, we have two possibilities: (a) Remove such lines up front by running a filter over the input before the comparison process. (b) Use a pattern matcher to sweep over the matrix and remove single dots. We do both.

2.3. Pattern Matching to Extract Copied Sequences

The algorithm as stated above does not catch duplicated code that was changed inside one line of code. In a sequence of copied code that is compared with the original sequence, a changed line shows up as a hole in the diagonal match pattern (see Figure 2 *b)*). To cope for this weakness when extracting whole copied sequences, a pattern matcher is run over the matrix which captures diagonal lines and allows holes up to a certain size in the middle of the line.

The sequence extraction is run automatically

and produces textual reports containing the detailed locations of the duplication.

2.4. Textual Reports

The report extract shown below presents matched sequences that were found in two comparisons. First, the participants of the comparison are identified. Then, a summary of all the sequences found is presented. Finally, all matched sequences are listed in detail. In the `Pattern`-string of a match, a vertical bar stands for two lines that matched, and a horizontal bar marks two lines that did not match. A dot in the pattern string represents a source line that contained comments or white space and was removed from the input before the comparison. Such lines are not taken into account during the pattern matching and are printed so that reported source line numbers correspond to the length of the pattern.

```
Comparison of
~/gcc/gcc-2.8.1/cp/pt.c
with itself

Sequence length: 13 Number of Sequences: 1

A Match between code from file 'pt.c'
(from line 2603 to line 2615)
[Pattern: ||-||-||-||||]
and code from file 'pt.c'
(from line 68 to line 80)
[Pattern: ||-||-||-||||].
(Stretch = 13 , RelevantLines = 10)


Comparison of
~/gcc/gcc-2.8.1/config/i386/i386.c
with
~/gcc/gcc-2.8.1/config/h8300/h8300.c

Sequence length: 17 Number of Sequences: 1

A Match between code from file 'i386.c'
(from line 3060 to line 3092)
[Pattern: |-..|-..|-..|-..|-..|-..|-..|-..|]
and code from file 'h8300.c'
(from line 1071 to line 1087)
[Pattern: |-|-|-|-|-|-|-|-|].
(Stretch = 17 , RelevantLines = 9)
```

Reports are important since they provide the exact location of the duplicated code. This information is handy for the maintainer that has to work with the code. Reports are also the basis for computing more abstract numbers like duplication percentages (see Section 4).

2.5. Visualization vs. Automated Detection

The advantage of the visualization over the automated pattern matching is twofold: First, images of duplication can be striking and allow to grasp situations immediately. A prime example is the image of the code evolution in Figure 7. Second, visualization allows an *exploratory* approach to the investigation of the duplication situation in a system. Exploratory in the sense that unknown configurations attract the eye of the user and lead to unexpected findings, whereas pattern matching only catches preprogrammed, known configurations.

Tool Support. We have implemented the presented approach in a tool called DUPLOC,[2] running under VISUALWORKS 2.5. DUPLOC offers a click-able matrix display which allows the user to look at the source code that produced the match. DUPLOC includes an information mural algorithm that enables the tool to present a matrix of 100'000 lines per side in its entirety on a 600x800 screen.

3. Selecting the Case Studies

The goal of our case studies is to stress the language independent aspects and to prove the potential of our approach. In order to choose the case studies we took four criteria into account: first the *implementation language*, second the *potential of duplication*, third the *size* of the system and fourth the possibility of *reproduction of the experiment* by other researchers.

Implementation Languages. We selected languages that have clearly different syntaxes: C, Smalltalk, Python and Cobol. Smalltalk is well-known to have a simple and uniform, keyword-based syntax. The syntax of Python is based on indentation which replaces block delimiters. In the overly verbose Cobol syntax, line numbers exist and identifiers that are attached at the end of each line. It is obvious that writing a parser for these diverse languages would be a entire new endeavor in each case [15].

Potential Duplication. We used case studies from different sources to maximize the potential range of the duplication. We took (a) two industrial case studies for which it was known that they contained a lot of duplication, (b) one case study where the duplication of code was suspected to be low, and (c) a small application from the public domain for which we had no knowledge about the duplication situation.

Size. The scalability of our approach has to be considered under two aspects: First, does it scale

[2] DUPLOC is available under a GNU license at http://www.iam.unibe.ch/∼rieger/duploc/.

given the size of the source code? Second, does it scale given the amount of duplication that is found? To prove that our approach is scalable under the first aspect, we took the full GNU gcc source code whose size is 13Mb. That our approach also scales regarding the second aspect was proved when one of the industrial applications happened to contain an enormous amount of duplication.

Reproducibility of the Experiment. We chose one well-known and freely available case study: the Free Software Foundation C compiler gcc [3] to make our experiments reproducible by others.

The Case Study Material The chosen case studies are: the implementation files of the GNU gcc source (written in C), a web-based message board (Python), parts of a payroll application (Cobol) and a database server (Smalltalk). The statistics of the case studies are given in the table below. Note that in the database server case one file contained one class.

Case	*Language*	*Size*	*# Files*	*LOC*
gcc	C	13.4 Mb	221	460000
Database Server	Smalltalk	7.1 Mb	593	245000
Payroll	Cobol	3 Mb	13	40000
Message Board	Python	265 K	36	6500

Initial Set-Up. We have performed the case studies according to this procedure: (1) We took source code about which we did not know anything. (2) Since the code was written in a number of different languages, we had to adapt our tool to the language at hand so it could normalize the lines (white space and comment removal). (3) We ran the tool off-line to produce the reports. (4) We extracted overall duplication percentages from the reports and analyzed them (see section 4). (5) We browsed the visual representations of the matrices to evaluate our findings (see section 5).

4. Code Duplication Overview

In this section, we will present the overall percentages of duplication which we extracted from the reports produced by our tool (see section 2.4). We take these numbers to be nothing more than very general indicators of duplication occurring in a system. We will not go into a more detailed analysis of the reports, since our aim in this section is only to prove that our approach detects a significant amount of duplication.

Constraining the Results. The results we present here have been obtained with the following constraints: First, to remove accidental duplication of small fragments, we limited the detection to sequences of lines having 10 or more lines. Second, to avoid missing duplicated sequences with changed parts, we allowed holes in the copied sequences up to 20%[4] of the total length of the sequence. Third, since reengineers are looking for the duplication of *functional* code elements, we chose to present the percentage of duplication in terms of *effective* lines of code. This means that we computed the percentage over the set of lines from which comments and white space have been removed (see section 2.1). This way we minimize the impact of comments in the percentage computation. As a consequence, a file can be integrally copied into a second, but if this second file contains a lot of comments the percentage will not reflect this situation.

4.1. Overview of the Duplication

Having a global percentage of duplication per application is the first indication of the state of an application.

Average Percentage of Duplication. The following table presents the average percentage of duplication per file. We also include the percentage in terms of entire code (i.e. files including comments) so that readers can have their own ideas about the relevance of the duplication detection. The third line shows the number of files that effectively contain duplicated code under the constraints we fixed (see above).

Note that inferior percentages for the entire code is normal because comments and white space can make up for a lot of lines.

Average percentage of duplication found per file				
Case	gcc	Datab. S.	Payroll	Mess. B.
effective LOC	8.7%	36.4%	59.3%	29.4%
entire LOC	5.9%	23.3%	25.4%	17.4%
# of files with duplication	143	464	13	24
Total # of Files	170	593	13	36

The quite high average percentage found for the two industrial case studies (Cobol payroll system and Smalltalk database server) is not totally surprising considering fact that these were given to us because it was suspected that they contained a lot of duplication. Nevertheless we were astounded by their overall duplication ratio. The web message board system shows some duplication elements that are result from evolutionary *clones*, since the system was given to us as a snapshot in

[3] ftp://prepr.ai.mit.edu/pub/gnu/

[4] These thresholds come from our experiences with the case studies.

the middle of an extension, thus containing old as well as new code side by side. The `gcc` source code has the lowest ratio. This is not surprising because `gcc` is known to be software of a good quality.

Now we refine our analysis by looking at the duplication percentage per file. We present the payroll system and `gcc` because they cover the extremes in the range of our case studies. Note that the tables in Figures 3 and 4 only display the files containing the effective duplication.

Percentage per File: the Payroll Case. For the payroll system, the overview (Figure 3) immediately identifies three main groups according to the degree of duplication: (a) few duplication (around 5% in file F), (2) some duplication (from 25% to 50% in files A, B, D, E and J) and (3) mostly duplicated (up to 70% in files C, G, H, I, K, L and M).

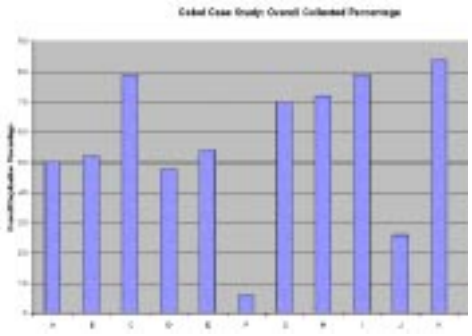

Figure 3. Duplication Percentage per Files in the payroll case study.

Percentage per File: the `gcc` Case. Even if the average percentage showed that `gcc` has the lowest percentage of duplication, looking at the percentage per file (Figure 4) gives another view. We see that two files have more than 60% of duplication, that 6 files have more than 50% of duplication and that a number of files have more than 20% of duplication.

The data from the reports that the analysis of this section was made with also serves the software maintainer in the process of eliminating duplication. What we want to do in the next section is to look at line-based comparison data from the angle of its representation in scatter-plots.

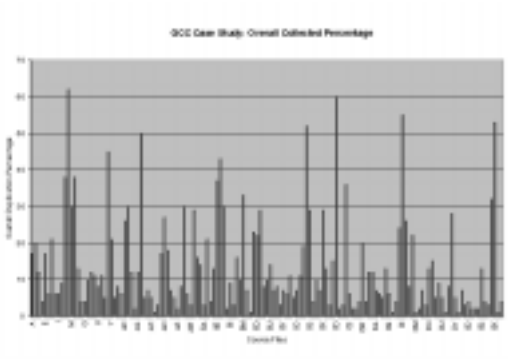

Figure 4. Duplication Percentage per Files in the `gcc` case study.

5. Visually Analyzing the Duplication

In this section, we want to give credibility to our claim, from section 2.2, that visualizations of duplication help the maintainer. First the graphical visualizations of the duplication gives a quick idea in terms of the frequencies and the size of the duplicated elements. Second, they support understanding of the nature of the duplication, e.g. if the file has been cloned or if one big chunk or multiple small chunks have been copied. The images presented in this section contain all the found matches, i.e. we did not remove spurious dots from the plots to "clean" them.

Besides the copy of small code fragments occurring inside a same file or between different files, we present clones of entire files and evolutionary changes that we found in the case studies..

Cloned Files

By *cloned files* we designate files that have a very high duplication ratio between each other. Both, in the payroll case as well as in the database server case, we found distinctive files that were almost identical copies of each other. In Figure 5 we see that file K is mostly a copy of file I and we see where the few changes have been applied.

In Figure 6 we see five classes that are identical copies (except class E which exhibits some minor changes). The spurious dots and rectangular patterns found in the plots act as a kind of visual fingerprint, which would help to find the members of this "club" in a larger matrix even if they were not clustered together like in Figure 6.

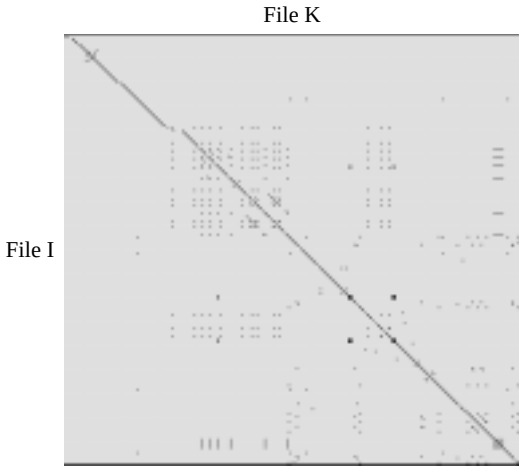

Figure 5. Comparison of the two Cobol files I and K from Figure 3.

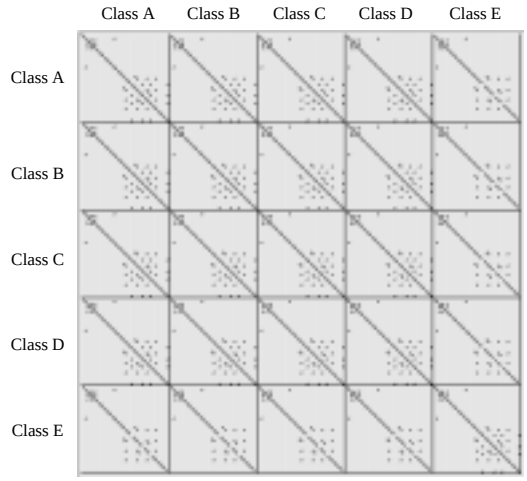

Figure 6. Five sibling classes in the database server application (Smalltalk). The five rectangles containing the main diagonal represent the comparisons of each class with itself.

Evolutionary Change

Knowing that the message board code contained old and new versions, we can observe how code evolution appears in scatter-plots (the following explanations refer to the right rectangle in Figure 7). Most of the changes that were applied to the code consisted in adding lines. This shows up as broken diagonals that are progressively shifted to the right. In the middle of the diagram, however, we see a down-shift of the diagonal, telling us that a chunk of code has been removed. Since one matrix coordinate corresponds to one source line, we are able to estimate the size of the changed code chunks easily, which facilitates understanding further. Note that the same information could be obtained using the `diff` tool, but it is only via the image that the user understands the changes quickly.

Note that Seesoft [3] that interactively displays line oriented statistics like the age of the line and its programmer is a better suited tool for this kind of analysis.

Additional Comments About the Visualization

The plain line-based comparison produces sometimes highly redundant comparison matrices like the one in Figure 8. The fact that we have horizontal as well as vertical repetitions of the diagonal sequences is due to the fact that the programmer copied the same sequence in *both* files *multiple* times (up to 16 times in the example). To be useful to the maintainer, the visual redundancy could be removed from the picture. This is left to be done in future work.

We conclude that visualization emphasizes duplication situations, sometimes to the point of shouting at the maintainer that the code should be fixed. Moreover, visualization helps to understand where and how code has changed between versions.

6. Technical Remarks

We add some technical remarks about the tool we implemented.

Scalability. The scalability issue is a crucial question when striving for applicability in the industrial context where system sizes of hundreds of KLOCs are normal. The table below summarizes some performance statistics gathered during our case studies. We have run the tool on a single G3 MacIntosh with 230Mhz and 100Mb RAM. It checked for all matching code sequences which where longer than 10 matching lines (including holes). Note that the task of computing the comparisons of a set of files can be easily parallelized. In the table below, he row entitled *parallel tasks* reports how many parallel runs we used to compute the data for this case study. The row entitled *comparisons* reports the number of file-to-file

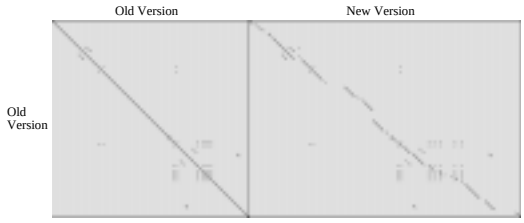

Figure 7. Comparisons of two versions of a Python-file. In the left square the old file is compared to itself, in the right rectangle, the old file is compared to the new.

comparisons that contained one or more matches of the required length.

The long running time for the database server case is due to the high number of files and the exceptionally high number of duplication detected in the system.

Performance Statistics				
Case	`gcc`	Database S.	Payroll	Message B.
# of files	221	593	13	36
LOC	460000	245000	40000	6500
parallel tasks	1	2	1	1
running time	6h30m	5h/7h	8m	1m
comparisons	1670	22939	51	50

We experienced that the performance of the current implementation tool is sufficient for systems sizes below 1MLOC. We think that in a maintenance project, duplication data is something that does not change frequently and can be computed once over night and then be interpreted afterwards Should performance turn out to be a critical factor, a re-implementation of the tool in C++ would certainly amend the problem.

7. Related Work

The analysis of code to identify copy and paste and plagiarism [5, 4, 11, 7] is broad. Various techniques are used: structural comparison using pattern matching [14], metrics [13, 10] or statistical analysis of the code, code fingerprints [12, 8, 9].

[5, 4] detect student plagiarism using statistical comparisons of style characteristics such as the use of operators, use of special symbols, frequency of occurrences of references to variables or the order in which procedures are called. [7] uses the static execution tree (the call graph) of a program to determine a fingerprint of the program.

In [14], a regular language is proposed to identify programming patterns. Cloning can be detected if we assume that if two code fragments can be generated by the same patterns then they could be clones.

Figure 8. Extract of a comparison between two different files from the database server (Smalltalk). It presents rich but confusing patterns of duplication.

Johnson[8] uses a specific heuristic, using constraints for the number of characters as well as the number of lines, to gather a number of lines a *snip* of source code on which he applies the fingerprint algorithm. `Sif` [12] that is also based on the same idea. However no graphical support is provided and the reports only present an overall similarity percentage between two files.

[10] evaluates the use of five data and control flow related metrics for identifying similar code fragments. The metrics are used as signatures for a code fragment. The technique supports change in the copied code. However it is not language independent because it is based on Abstract Syntax Tree Annotation.

The visual display used in DUPLOC is not new, DOTPLOT [6] uses the same principle. DOTPLOT has been used to compare source code, but also filenames in a file system and literary and techni-

cal texts. However it does not support source code browsing and the report facilities.

DUP [1] is a program that detects parameterized matches and generates reports on the found matches. This work is however mostly focused on the algorithmic aspects of detecting parameterized duplication and not on the application of the technique in an actual software maintenance and reengineering context. In particular code browsing is not supported.

The tool of [2] transforms source code into abstract syntax trees and detects clones and near miss clones among trees. It reports similar code sequences and proposes unifying macros to replace the found clones. Their approach requires, however, a full-fledged parser.

8. Conclusion and Future Work

In this paper, we presented a language independent approach for detecting duplicated code. The approach is based on (1) *simple* line-based string matching, (2) *visual presentation* of the duplicated code and (3) detailed *textual reports* from which overview data can be synthesized.

We presented results from a number of case studies and we showed that we can easily identify (1) duplicated code between several files, (2) within the same file, (3) cloned files and (4) evolution files. We claim that the data that was generated is useful information for practical software maintenance and reengineering tasks.

Moreover, the results we found were in certain instances beyond our expectations. We are referring in particular to the two industrial case studies. The provider of the source code suspected code duplication in the system, but we found much more than expected. The high amount of duplication, especially in the database server case study, suggests that the simple algorithm does not miss too much of the duplication that is actually going on in the system. Evaluating the impact of a parameterized match is however one of our future goals.

Future Plans. We plan to (a) Experiment with different algorithms for fuzzy matching, (b) Evaluate the impact of a parameterized comparison algorithm on this approach. We want to qualify how much of duplication we are missing and compare the preprocessing time, space tradeoff and languages independent lose between the two approaches. (c) Evaluate the gain in time that `Sif` [12] as a first filter of similar files can provide. (d) Develop a methodology for finding essential duplicated code in a system. This includes especially the fine tuning of the report facility. We will work in close contact with industrial software maintainers. (e) Investigate if textual comparison is a useful approach for exploring the structure of data other than source code, like for example program execution traces, where it could be interesting to find recurring patterns of execution sequences.

8.1. Acknowledgments

This work has been funded by the Swiss Government under Project no. NFS-2000-46947.96 and BBW-96.0015 as well as by the European Union under the ESPRIT program Project no. 21975. We want to thank Kurt Verschaeve and Bart Wydaeghe from the SSEL of Vrije Universiteit Brussels for fruitful discussions.

References

[1] B. S. Baker. A Program for Identifying Duplicated Code. *Computing Science and Statistics*, 24:49–57, 1992.

[2] I. Baxter, A. Yahin, L. Moura, M. S. Anna, and L. Bier. Clone Detection Using Abstract Syntax Trees. In *Proceedings of ICSM*. IEEE, 1998.

[3] S. G. Eick, J. L. Steffen, and E. E. S. Jr. SeeSoft—A Tool for Visualizing Line Oriented Software Statistics. *IEEE Transactions on Software Engineering*, 18(11):957–968, Nov. 1992.

[4] S. Grier. A tool that detects plagiarism in pascal programs. *SIGSCE Bulletin*, 13(1), 1981.

[5] M. Halstead. *Elements of Software Science*. Elsevier North-Holland, 1977.

[6] J. Helfman. Dotplot Patterns: A Literal Look at Pattern Languages. *TAPOS*, 2(1):31 – 41, 1995.

[7] H. Jankowitz. Detecting plagiarism in student pascal programs. *Computer Journal*, 1(31):1–8, 1988.

[8] J. H. Johnson. Identifying Redundancy in Source Code using Fingerprints. In *Proceedings of CASCON 93*, pages 171–183, 1993.

[9] J. H. Johnson. Substring Matching for Clone Detection and Change Tracking. In *Proceedings of the International Conference on Software Maintence (ICSM)*, pages 120–126, 1994.

[10] K. Kontogiannis. Evaluation Experiments on the Detection of Programming Patterns Using Software Metrics. In I. Baxter, A. Quilici, and C. Verhoef, editors, *Proceedings Fourth Working Conference on Reverse Engineering*, pages 44 – 54. IEEE Computer Society, 1997.

[11] N. Madhavji. Compare: A collusion detector for pascal. *Techniques et Sciences Informatiques*, 4(6):489–498, nov 1985.

[12] U. Manber. Finding similar files in a large file system. In *Proc. 1994 Winter Usenix Technical Conference*, pages 1–10, 1994.

[13] J. Mayrand, C. Leblanc, and E. Merlo. Experiment on the automatic detection of function clones in a software system using metrics. In *International Conference on Software System Using Metrics*, pages 244–253, 1996.

[14] S. Paul and A. Prakash. A framework for source code search using program patterns. *IEEE Transactions on Software Engineering*, 20(6):463–475, jun 1994.

[15] H. Reubenstein, R. Piazza, and S. Roberts. Separating Parsing and Analysis in Reverse Engineering Tools. In *First Working Conference on Reverse Engineering*, pages 117–125, 1993.

[16] I. Sommerville. *Software Engineering*. Addison-Wesley, fifth edition, 1996.