Reuse Measurement with Line and Word Runs

Bart Childs
Department of Computer Science
Texas A&M University, USA
bart@cs.tamu.edu

Johannes Sametinger
Department of Computer Science
Brown University, USA
js@cs.brown.edu
and *CD Lab* for Software Engineering
Johannes Kepler University, Austria

Abstract

Software reuse provides several advantages, e.g., increased productivity and software quality, decreased development time and costs. Installing reuse programs requires up-front investments. Empirical data showing that a potential for software reuse exists in a certain environment will help managers to decide on such investments. In order to determine the potential productivity gain it is necessary to know the amount of similarities in one's systems.

Systematic black-box reuse increases productivity more than white-box reuse. However, white-box reuse is the usual means of dealing with common parts in different systems. We will demonstrate that word and line runs provide an effective means for measuring ad-hoc reuse and determining reuse potential. The suggested measurement can be used to determine candidates for reusable components and, thus, help in focusing reuse investments.

Line and word run measurement can be used to find similarities in any text and can be used for other purposes as well. We will demonstrate other applications like spotting locations of possible redesign in object-oriented programs.

Key Words:
software reuse, reuse potential, reuse measurement, line and word runs, literate programs, TEX, C++

1 Introduction

Software reuse is the process of creating software systems from existing software rather than building them from scratch [Krueger92]. Reusable software has many benefits, the most common ones of which are reduction of development time and redundant work; ease of documentation, maintenance, and modification; improvement of performance and quality of software; expertise sharing and intercommunication among designers; and construction of complex software systems with smaller teams. For more details on reuse see [Braun94, Krueger92, Mili95].

Despite advantages of software reuse, installing corporate-wide reuse programs is not an easy task and requires up-front investment. Reuse is typically measured after it has been done and is a gauge for the success or failure of reuse programs. We want to show beforehand that reuse potentials are present and where they are. In order to determine the potential productivity gain it is necessary to know the amount of similarities in one's applications.

We will demonstrate a way of determining the amount of white-box reuse and the potential for explicit reuse in source code and documentation. For this purpose we will use literate programs–TEX and METAFONT–and a C++ class library, i.e., ET++, as input.

TEX is a book quality formatting system [Knuth86a, Knuth86b]. METAFONT is a system that enables a programmer/artist to create a family of fonts for TEX's use [Knuth86c, Knuth86d]. Both systems were implemented by Donald Knuth as literate programs. ET++ is an extensive application framework for C++ [Weinand89]. For the literate programs we measure reuse in the WEB sources containing source code and documentation. The WEB source is input to the literate programming system, which subsequently generates code and high-quality documentation.

The structure of the paper is as follows: Section 2 contains some general information about the measurement of reuse. In Section 3 we discuss our first approach that is simply based on lines and words. Then, 'line runs' and 'word runs' are presented in Section 4. Results of line and word run measures for TEX and ET++ are presented in Sections 5 and 6, respectively. Some implementation aspects for the evaluation of our measures follow in Section 7. Finally, we present our conclusions in Section 8.

2 Reuse Measurement

Software reuse is no exception to the rule "we cannot manage what we cannot measure." Reuse spans multiple projects and has an influence even on organizational structures of companies. To manage such enterprise-wide activities requires monitoring. Software metrics can be used to estimate costs, cost savings, and the value of software practices [Poulin92]. The amount of software reuse (the reuse level) in a certain software system can easily be determined by the ratio of reused components (or their number of lines of code) to the total number of components of the system (or total amount of code), see Fig. 1. This measure does not consider more subtle aspects like adaptation costs but it is objective.

$$R = (1 - \frac{New}{Total}) \times 100$$

R	reuse percentage
New	number of new components
$Total$	total number of components

Figure 1: Reuse Measurement based on Components

Banker *et al* present a reuse percentage which focuses on the total benefit that is attributable to reuse [Banker93]. The measure represents the reuse of existing, unmodified objects and is defined as the proportion of calls of such objects.

Metrics based on generalization costs and reuse savings are proposed by Henderson-Sellers [Henderson-Sellers93]. It is important to consider the whole spectrum of life-cycle costs, like costs for testing, verification, and maintenance. On the one hand, these costs will hopefully be reduced by software reuse. On the other hand, despite cost reductions during development, the integration of a reused code may have an impact on the overall system design and a negative impact on software maintenance [Mittermeir90].

Measuring reuse effectiveness includes measuring costs, savings, and improvements in quality. For example, a model for measurement and metrics for object-oriented systems is presented in [Vaishnavi96].

In this paper we will concentrate on a measure based on any text, i.e., arbitrary source code and documentation text. We will provide an objective measure for white-box reuse, but will not consider aspects like savings or costs of reuse.

Reuse measures presented in the literature indicate reuse in a new software system, e.g.,40 percent of system A had been reused from other systems.

We are more interested in how much a system has been reused in other systems than one specific reuse. The former indicates 'reuse potential' while the latter is a measure of a particular reuse.

We want to know how much of `system A` had been reused in `system B`, or at a finer granularity, how much of `component A` had been reused in `component B`. This is measuring white-box reuse and is interesting for an indication of whether it is worthwhile to make a component more reusable and, thus enable its reuse as a black box. This is the case when the results indicate a high reuse in many other components.

$$R_{l(ab)} = (1 - \frac{C_{l(ab)}}{T_{l(a)}}) \times 100$$

$R_{l(ab)}$	line reuse percentage of `a` in `b` 0 means no reuse, 100 means everything (all lines) of `file a` have been reused in `file b`. (word reuse percentage is determined accordingly.)
$C_{l(ab)}$	number of lines to be changed in `file a` in order to get contents of `file b`
$T_{l(a)}$	total number of lines of `file a`

Figure 2: Reuse Measurement based on Lines

3 Lines and Words

We started our first approach in reuse measurement by comparing lines and words of files. In order to get a first hint for reuse we compare each file of a system with each file of another system. The number of lines that have to be changed in `file a` to transform its contents to the contents of `file b` as opposed to the total number of lines (of `file a`) gives an indication of how much of `file a` had been reused in `file b`. Fig. 2 shows how the line reuse percentage (of `file a`) can be determined. This formula can be used in both directions, i.e., to determine how much of `file a` is being reused in `file b` or how much of `file b` is being reused in `file a`, by using the total number of lines of `file a` or `file b`, respectively.

As empty lines are considered to be equal, the reuse percentage, naturally, is greater than zero, if empty lines appear in both files. Thus, it is crucial that empty lines be eliminated before the reuse percentage is determined. Of course, for two equal files the reuse percentage is 100. Without empty lines it should be zero for nominally different files. But this is not always true. We will discuss this later.

Comparing lines does not give a reliable indication of white-box reuse, as modifications are often made on a word basis. Subtle differences are often found by investigating white-box reuse. Examples include: single words had often been changed, line breaks were changed, the order of chapters and/or sections was different, user supplemented index entries were different, and sentences were changed in syntax (e.g., word ordering) without changes in the semantics.

We can determine word reuse by replacing blanks with newlines. A word in this context is a sequence of characters separated by blanks or newlines. But again, two different files can have a reuse percentage that is greater than zero, because it is not improbable that single words appear in both files, even though they really have nothing in common. We will denote R_l and R_w as reuse percentages considering lines and words, respectively.

Comparing lines and words gives a good indication about reuse. Usually, R_l and R_w do not differ much, with R_w slightly higher than R_l. If both R_l and R_w are high, then obviously reuse had been done. If R_l is low, but

R_w is high, then reuse had been done, but the reused text had been modified on a more local basis. This leads to many different lines and a lower value for R_l, but still leaves many identical words resulting in a higher value for R_w. Finally, if both R_l and R_w are low, then apparently there is no reuse at all.

We will demonstrate R_l and R_w on some small examples. For that purpose we take a paragraph from the chapter "Introduction to the syntactic routines" of TEX and METAFONT, see Fig. 3. Identical lines are marked with an '=' at the beginning. Despite its high similarity there are only three identical lines in this paragraph. Word (runs) that appear in both systems are $\boxed{\text{boxed}}$.

The text of TEX in Fig. 3 contains 12 lines and 128 words. The text of METAFONT contains 13 lines and 135 words. 9 lines or 30 words have to be changed to transform the text of TEX to the text of METAFONT. This results in a line and word reuse of $R_l = 25.0\%$ and $R_w = 76.6\%$. The high difference between R_l and R_w indicates the modification and polishing of the text that had been done.

The paragraph in Fig. 4 has been taken from the chapter "File Names" and demonstrate reuse of source code also. In this particular case the documentation had been left unchanged, but modifications to the source code had been made. In this example line and word reuse result in $R_l = 69.2\%$ and $R_w = 79.7\%$, indicating high reuse and less wordsmithing.

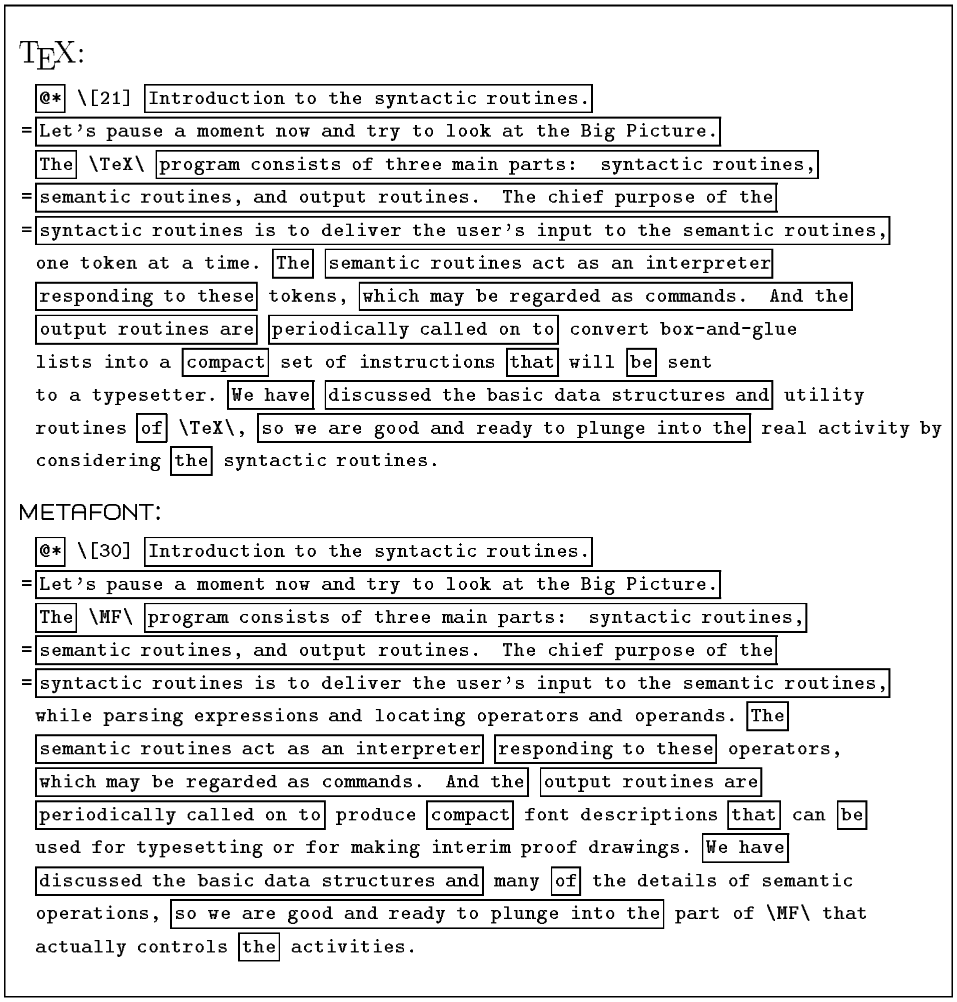
We include two additional samples from arbitrary chapters of TEX and METAFONT, as well as two arbitrary files of the ET++ application framework. We take TEX_{14} and MF_{42}, i.e., chapters number 14 and 42 of TEX and METAFONT, respectively. Even though these chapters are dissimilar, their resulting values for $R_l = 14.8\%$ and $R_w = 29.9\%$ are quite high. Similarly, the C++ files `Application.C` and `Document.C` have values of $R_l = 23.3\%$ and $R_w = 24.2\%$. The resulting reuse percentages of the four samples are summarized in Table 1.

The unexpectedly high results for R_l and R_w of samples 3 and 4 in Table 1 indicate high reuse. This needs further explanation. As it turns out some of the high reuse percentages of TEX and METAFONT chapters are due to their short lengths. For example, the chapter "System-dependent changes" consists only of 10 lines in TEX. Reusing a single line results in a reuse percentage of 10%. This is actually the case for some METAFONT chapters which contain an index entry for *system dependencies*. Some longer chapters also achieve a rather high reuse percentage. Take TEX's chapter "Copying boxes" as an example. It exhibits a reuse of more than 10 percent in 25 (!) METAFONT chapters. Looking for the reasons of this effect we found out that source code lines containing '`end;`' mostly contributed to this result.

C++ code has many lines containing only curly braces (often at the beginning or end of classes, functions, methods, and loops). It also turns out that preprocessor statements, like `#include` or `#pragma`, can distort the result. ET++ files compared with each other yield an average reuse of about 20 percent. This is caused soley by preprocessor statements and lines that contain only braces. This distortion can be eliminated by removing such lines before doing the measure. Nonetheless, determining the runs of identical lines is an appropriate and effective way to get information on actual reuse.

4 Line and Word Runs

If 10 consecutive lines are identical in two chapters/files it is likely that they were reused. If there is only one line it may have been reused, but it also may having nothing to do with reuse at all. This is the case with '`end;`' lines, even though in some cases such lines may be regarded as being reused in a certain context.



TEX:

```
 @* \[21] Introduction to the syntactic routines.
=Let's pause a moment now and try to look at the Big Picture.
 The \TeX\ program consists of three main parts:  syntactic routines,
=semantic routines, and output routines.  The chief purpose of the
=syntactic routines is to deliver the user's input to the semantic routines,
 one token at a time. The semantic routines act as an interpreter
 responding to these tokens, which may be regarded as commands.  And the
 output routines are periodically called on to convert box-and-glue
 lists into a compact set of instructions that will be sent
 to a typesetter. We have discussed the basic data structures and utility
 routines of \TeX\, so we are good and ready to plunge into the real activity by
 considering the syntactic routines.
```

METAFONT:

```
 @* \[30] Introduction to the syntactic routines.
=Let's pause a moment now and try to look at the Big Picture.
 The \MF\ program consists of three main parts:  syntactic routines,
=semantic routines, and output routines.  The chief purpose of the
=syntactic routines is to deliver the user's input to the semantic routines,
 while parsing expressions and locating operators and operands. The
 semantic routines act as an interpreter responding to these operators,
 which may be regarded as commands.  And the output routines are
 periodically called on to produce compact font descriptions that can be
 used for typesetting or for making interim proof drawings. We have
 discussed the basic data structures and many of the details of semantic
 operations, so we are good and ready to plunge into the part of \MF\ that
 actually controls the activities.
```

Figure 3: Sample 1

TEX:

```
=@ Here we have to remember to tell the |input_ln| routine not to
=start with a |get|.  If the file is empty, it is considered to
=contain a single blank line.
=@^ system dependencies@>
 @^ empty line at end of file@>

=@<Read the first line...@>=
=begin line:=1;
=if input_ln(cur_file,false) then do_nothing;
=firm_up_the_line;
 if end_line_char_inactive then decr(limit)
 else buffer[limit]:=end_line_char;
 first:=limit+1; loc:=start;
=end
```

METAFONT:

```
=@ Here we have to remember to tell the |input_ln| routine not to
=start with a |get|.  If the file is empty, it is considered to
=contain a single blank line.
=@^ system dependencies@>

=@<Read the first line...@>=
=begin line:=1;
=if input_ln(cur_file,false) then do_nothing;
=firm_up_the_line;
 buffer[limit]:="%"; first:=limit+1; loc:=start;
=end
```

Figure 4: Sample 2

The “Copying boxes” chapter mentioned above has only runs with length 1. Of course the chapters with a high reuse percentage also have runs with length 1. However, there were many which could be considered for real reuse, e.g., when a paragraph is reused with some lines being modified leaving identical lines without an identical predecessor or successor. Consider the example in Fig. 3 where the paragraph has apparently been reused but it has only two runs of identical lines with lengths 1 and 2.

Table 2 contains the run lengths for lines and words for the four samples. The third sample in Table 2 demonstrates a high reuse percentage resulting from single lines and words. (The maximum length for line runs is 1, see M_l.) In the C++ code of the fourth sample we have 80 line runs. However, the average length is only 1.1, and the maximum length is 3. It seems appropriate to consider only lines and words that are part of a run of certain length. When we consider only runs with a minimum length of 2, the picture looks quite different. The numbers for samples 1 and 2 drop only slightly (indicating reuse), whereas the numbers for samples 3 and 4 drop drastically. The remaining four line runs for the C++ sample originate from `#include` and `return` statements. Please note that including a file is some sort of reuse, too. With our reuse measurement we do not take this into account, but rather make simple text comparisons without considering any semantic information.

Thus, the determination of the reuse percentages by considering line runs and word runs is justified rather than just using lines and words. Table 3 shows the resulting reuse percentages for line runs of lengths 1, 2, and 3 and for word runs of lengths 1, 2, and 5.

The results show that in sample 1 there are no long line runs (no one longer than 2), but there are many word runs with at least a length of 5. This indicates high reuse with modifications. Sample 2 shows a slight drop both in line runs and in word runs, indicating high reuse also. However, samples 3 and 4 show a rapid drop of runs, indicating that the originally high numbers cannot be contributed to reuse.

$$R_{l,len(ab)} = (1 - \frac{C_{l(ab)}}{T_{l(a)}}) \times 100$$

$R_{l,len(ab)}$	line reuse percentage of `a` in `b` considering runs with minimum length *len*. (word reuse percentage is determined accordingly.)
$C_{l(ab)}$	number of lines to be changed in `file a` in order to get contents of `file b` (equal lines in runs with length $<$ *len* are considered as being different)
$T_{l(a)}$	total number of lines of `file a`

Figure 5: Reuse Measurement based on Line Runs

Figure 5 contains the run based measurement for reuse. In order to evaluate reuse, R_l and R_w have to be determined for various lengths. (The lengths may vary depending on the input data.) We will demonstrate this on two more extensive examples in the next two sections.

5 Example 1: TEX System

TEX’s sources contain about 21,500 lines and 122,000 words. METAFONT’s sources consist of about 20,500 lines and more than 110,000 words. TEX and METAFONT are subdivided into 55 and 52 chapters, respectively.

First, we compare each chapter of TEX with each chapter of METAFONT and determine the values for R_l and R_w for various lengths (measuring how much of TEX had been reused in METAFONT.

Sample	R_l	R_w	T_l	T_w
1. Fig. 3	25.0	76.6	12	128
2. Fig. 4	69.2	79.7	13	59
3. $\TeX_{14} \rightarrow \mathsf{MF}_{42}$	14.8	29.9	81	428
4. Appl.C→Doc.C	23.3	24.2	365	901

R_l:	line reuse percentage
R_w:	word reuse percentage
T_l:	total number of lines
T_w:	total number of words

Table 1: Line and Word Reuse in Samples

Sample	*len*	n_l	avg_l	M_l	Σ_l	n_w	avg_w	M_w	Σ_w
1. Fig. 3	1	2	1.5	2	3	12	8.2	30	98
	2	1	2.0	2	2	6	15.3	30	92
2. Fig. 4	1	3	3.0	4	9	3	15.7	33	47
	2	2	4.0	4	8	3	15.7	33	47
3. $\TeX_{14} \rightarrow \mathsf{MF}_{42}$	1	12	1.0	1	12	123	1.0	2	128
	2	0	0.0	0	0	5	2.0	2	10
4. Appl.C→Doc.C	1	80	1.1	3	85	165	1.3	5	218
	2	4	2.2	3	9	35	2.5	5	88

len:	minimum length for runs to be considered
n_l, n_w:	number of line/word runs
avg_l, avg_w:	average length of line/word runs
M_l, M_w:	maximum length of line/word runs
Σ_l, Σ_w:	sum of lengths of line/word runs

Table 2: Line and Word Runs in Samples

Sample	$R_{l,1}$	$R_{l,2}$	$R_{l,3}$	$R_{w,1}$	$R_{w,2}$	$R_{w,5}$
1. Fig. 3	25.0	16.7	0.0	76.6	71.9	71.9
2. Fig. 4	69.2	61.5	61.5	79.7	79.7	74.6
3. $\TeX_{14} \rightarrow \mathsf{MF}_{42}$	14.8	0.0	0.0	29.9	2.3	0.0
4. Appl.C→Doc.C	23.3	2.5	0.8	24.2	9.8	1.7

$R_{l,i}$:	reuse percentage for line runs with minimum length i
$R_{w,i}$:	reuse percentage for word runs with minimum length i

Table 3: Run based Reuse in Samples

The results are shown in Table 4, sorted in descending order by $R_{l,1}$. Not surprisingly pairs of chapters with identical or similar titles show up at the top. Some chapters with identical titles have a rather low reuse percentage, e.g., "Saving and restoring equivalents" ($\TeX_{19} \rightarrow \mathsf{MF}_{16}$), "Expanding the next token" ($\TeX_{15} \rightarrow \mathsf{MF}_{12}$), or "The command codes" ($\TeX_{25} \rightarrow \mathsf{MF}_{35}$). The results clearly show reuse in the top chapters with similar or equal titles. They also show that despite the high values for $R_{l,1}$ there is no reuse below the entries of $\TeX_{14}$, because reuse percentages drop to zero quickly with increased run lengths.

Having a closer look at the investigated literate programs we found, that software reuse had been (successfully) applied to both source code and documentation. This was done primarily by text scavenging.

We had expected a certain degree of reuse in the results. However, our expectations have been exceeded and we were surprised about the high differences between line and word reuse. More details about reuse in the TEX systems are given in [Childs96].

6 Example 2: ET++

We also tested our reuse measuring method by applying it to the files of ET++, an application framework implemented in C++ [Weinand89].

We have mentioned earlier that C++ code contains many curly braces and preprocessor statements. In order to get rid of such similarities in the result of the comparison we evaluated the reuse percentages by considering only line runs with minimum lengths 5 and 10, and word runs with minimum lengths 10 and 20.

Version 3.2.2a0 of ET++ has 169 `.h` and 146 `.C` files. We compared all `.h` and `.C` file with each other, which results in $169^2 + 146^2$, i.e., almost $50,000$ file comparisons. Out of these 50,000 comparisons, 260 resulted in at least one value greater than zero for $R_{l,5}$ or $R_{w,10}$. However, only a few have considerable values for $R_{l,10}$ and/or $R_{w,20}$. These are shown in Table 5. Most entries show up twice in reverse order, because comparisons had been done in both directions. Even without knowing the particular class library, the numbers reveal some interesting facts.

Some files, e.g., `Date.C`, `Time.C`, and `DateTime.C` share a high percentage of code. `DateTime.C` is apparently a combination of `Date.C` and `Time.C`. The corresponding `.h` files do not show up with high similarities. Inspecting the code reveals that–surprisingly–both files `Date.h` and `Time.h` simply include `DateTime.h`.

The files `Math.C` and `MathUtils.C` contain the same implementation of class `Math`. File `Math.h` is empty and includes file `MathUtils.h` which contains the definition of class `Math`.

The classes `CollView` and `CollView1` contain slightly different definitions and implementations of a *collection view*. Class `CollView` is based on a class `View`, whereas class `CollView1` is based on class `GridView`.

Similar observations can be made for the other files showing high similarities. We draw the following conclusions from these results:

- *Redesign*
  High similarities may indicate potential locations for redesign.
- *Unfinished work*
  High similarities may indicate locations of unfinished code.

Naturally, it is not possible to automatically determine locations that need redesign. This is a subjective matter. However, high similarities suggest to have a look at the code and to determine the reason for these similarities. They may have good reasons. But they may also suggest a redesign cycle or, for example, indicate temporary files.

Chapters	$R_{l,1}$	$R_{l,2}$	$R_{l,3}$	$R_{w,1}$	$R_{w,2}$	$R_{w,5}$
$\TeX_6 \rightarrow \mathsf{MF}_6$	82.7	80.2	77.4	93.3	92.9	91.5
$\TeX_2 \rightarrow \mathsf{MF}_2$	81.6	79.6	77.7	76.9	76.1	75.8
$\TeX_3 \rightarrow \mathsf{MF}_3$	81.1	78.7	76.1	93.4	93.2	92.5
$\TeX_{54} \rightarrow \mathsf{MF}_{51}$	80.0	80.0	60.0	97.6	96.4	96.4
$\TeX_9 \rightarrow \mathsf{MF}_{10}$	72.1	67.9	65.7	85.1	84.6	82.9
$\TeX_{52} \rightarrow \mathsf{MF}_{50}$	71.9	68.8	62.5	89.2	87.8	86.6
$\TeX_4 \rightarrow \mathsf{MF}_4$	71.5	65.9	62.6	90.2	88.9	87.0
...	...	...	...	...	...	...
$\TeX_{18} \rightarrow \mathsf{MF}_{13}$	19.4	14.2	9.7	53.2	47.7	44.7
$\TeX_{28} \rightarrow \mathsf{MF}_{36}$	17.1	13.9	9.9	38.1	35.5	28.7
$\TeX_{20} \rightarrow \mathsf{MF}_{14}$	16.8	9.9	7.5	34.7	24.1	18.3
$\TeX_{14} \rightarrow \mathsf{MF}_{42}$	14.8	0.0	0.0	29.9	1.9	0.0
$\TeX_{14} \rightarrow \mathsf{MF}_{35}$	14.8	0.0	0.0	25.7	3.5	0.0
$\TeX_{14} \rightarrow \mathsf{MF}_{43}$	14.8	0.0	0.0	23.4	0.9	0.0
...	...	...	...	...	...	...
$\TeX_{25} \rightarrow \mathsf{MF}_{35}$	7.3	0.8	0.0	18.3	9.4	5.0
$\TeX_{19} \rightarrow \mathsf{MF}_{16}$	3.1	0.0	0.0	8.5	3.1	0.5
$\TeX_{15} \rightarrow \mathsf{MF}_{12}$	0.0	0.0	0.0	19.9	5.6	5.6
...	...	...	...	...	...	...

Table 4: Line and Word Runs of TEX in METAFONT

Files	$R_{l,5}$	$R_{l,10}$	T_l	$R_{w,10}$	$R_{w,20}$	T_w
PopupTearOff.h→Menu.h	58.3	30.0	60	69.1	57.6	191
CollView1.h→CollView.h	34.8	29.2	89	47.6	35.3	275
CollView.h→CollView1.h	31.3	26.3	99	42.8	31.7	306
Menu.h→PopupTearOff.h	25.7	13.2	136	26.8	22.3	493
ObjFloat.h→ObjInt.h	21.4	0.0	42	19.6	19.6	107
ObjInt.h→ObjFloat.h	19.1	0.0	47	16.3	16.3	129
CheapText.h→GapText.h	7.2	0.0	69	11.3	11.3	291
MathUtils.C, Math.C	96.6	89.1	119	96.1	96.1	333
Date.C→DateTime.C	95.7	95.7	47	97.1	97.1	140
Math.C→MathUtils.C	95.0	87.6	121	96.1	96.1	333
Time.C→DateTime.C	82.6	82.6	23	82.4	82.4	51
DateTime.C→Date.C	68.2	68.2	66	74.3	74.3	183
CollView1.C→CollView.C	44.3	25.9	375	52.1	37.0	924
PopupTearOff.C→Menu.C	37.0	19.3	254	55.6	23.8	626
CollView.C→CollView1.C	34.6	20.2	480	38.6	27.4	247
DateTime.C→Time.C	28.8	28.8	66	23.0	23.0	183

Table 5: Line and Word Runs in ET++

Of course, reuse had been done in ET++ by means of inheritance. Inheritance is natural in object-oriented systems and provides a means of systematic reuse. We do not measure this kind of reuse. Instead we measure reuse that had been done by means of text scavenging, i.e., ad-hoc reuse. Thus, we discover locations where reuse should be done in a more systematic manner. For example, the results of our measures could be used to locate similarities of classes. Extracting these similarities and designing abstract base classes is a way to transform discovered ad-hoc reuse into systematic reuse.

```
c1
@* \[21] Introduction to the ...
.
c3
The \MF\ program consists of ...
.
c6 12
while parsing expressions and ...
semantic routines act as an ...
which may be regarded as commands.  ...
periodically called on to produce ...
used for typesetting or for making ...
discussed the basic data structures ...
operations, so we are good and ...
actually controls the activities.
.
```

Figure 6: Sample Output of *diff*

7 Implementation

We will demonstrate on the small examples of Fig. 3 and Fig. 4 how we determine the reuse percentages based on line runs. Word runs are evaluated similarly, but first blanks are being replaced with newlines.

As first step we eliminate blank lines and use the UNIX *diff* tool to compare the two sources. *diff* outputs edit commands (*add*, *change*, and *delete*) that can be applied to change the contents of one file to another. Fig. 6 shows the output of *diff* when comparing the two paragraphs of Fig. 3. The output means that the first line should be changed to the contents shown in the next line. The period ends the data for the change command.

We simply skip the data and grab only the pure edit commands (by using the UNIX *grep* tool). The result for Fig. 3 is `c1`, `c3`, and `c6 12`. This means that line 1, line 3 and lines 6 to 12 have to be changed in order to change the paragraph from the TEX to the METAFONT version.

Thus, we know that 9 lines out of 12 have to be changed and that there are two line runs, i.e., line 2 and lines 4 to 5. We do these calculations with a C++ program that reads the edit commands line by line, adds up the line runs, and determines the reuse percentages.

The example shown in Fig. 4 results in the edit commands `d5` and `c10 12`, indicating three runs. The runs are from line 1 to 4 (length 4), from line 6 to 9 (length 4), and at line 13 (length 1). In order to determine a run at the end of the text we have to know the text's length. We use a little trick for that purpose by appending two different lines to both texts. This guarantees that we get edit commands for the last line as well, giving us the information about the length of the text. With this extra line we get the commands `c1`, `c3`, and `c6 13` for Fig. 3 and `d5`, `c10 12`, and `c14` for Fig. 4. Thus, the lengths of the texts are 12 and 13.

8 Conclusions

We have presented a reuse measure that is based on runs of lines and words. The measure is effective in determining the amount of white-box reuse in any kind of software systems. It is language-independent and can be applied to source code and documentation. Additionally, it can be fully automated.

Reuse measurement based on line and word runs can be used for many different purposes. White-box reuse evaluation presented in this paper is just one example. Other applications are finding legal or illegal reuse in technical and scientific papers, determining (the amount of) modifications from one version of software to another, finding potential locations for redesign, or finding the amount of "reuse" in programs written by students for classes.

Using the proposed measure we have investigated some TEX systems and C++ code. C++ code generally contains many similar lines. Using runs of appropriate length can eliminate such inherent similarities. However, care must be taken not to increase the length too much and possibly overlook some reuse that can be seen only with smaller run lengths.

TEX and its related systems have been implemented in a literate manner. Thus, reuse in their context not only means reuse of source code but also of documentation.

The investigated TEX systems have been implemented from scratch by scavenging existing documentation and code. The amount of adhoc reuse turned out to be surprisingly high. The different results achieved by comparing lines and words indicate, that reused text had been carefully adapted. Thus, successful adhoc reuse had been done, but no attention had been directed to providing and reusing blackbox components. With the results of our measurement we are in a position of knowing the amount of reuse and its exact locations. This enables us to make statements about where reuse investments are most effective.

We expect to make similar statements for source code as well. The application framework we have used for our experiments is well designed and does not indicate further reuse potential in itself. We would have to make comparisons with various application systems to possibly find areas that might be extracted and included in the framework.

References

[Banker93] Rajiv D. Banker, Robert J. Kauffman, D. Zweig: "Repository Evaluation of Software Reuse," *IEEE Transactions on Software Engineering*, Vol. 19, No. 4, pp. 379–389, April 1993.

[Braun94] Christine Braun: "Reuse," in John J. Marciniak (Ed.): *Encyclopedia of Software Engineering*, Vol. 1, John Wiley & Sons, pp. 1055–1069, 1994.

[Childs96] Bart Childs, Johannes Sametinger: "Analysis of Literate Programs from the Viewpoint of Reuse," *Software – Concepts and Tools*, to be published, 1997.

[Henderson-Sellers93] Brian Henderson-Sellers: "The Economics of Reusing Library Classes," *JOOP*, Vol. 6, No. 4, pp. 43–50, July-August 1993.

[Knuth86a] Donald E. Knuth: "The TEX Book," Volume A of Computer & Typesetting, Addison-Wesley, 1986.

[Knuth86b] Donald E. Knuth: "TEX: The Program," Volume B of Computer & Typesetting, Addison-Wesley, 1986.

[Knuth86c] Donald E. Knuth: "The METAFONT Book," Volume C of Computer & Typesetting, Addison-Wesley, 1986.

[Knuth86d] Donald E. Knuth: "METAFONT: The Program," Volume D of Computer & Typesetting, Addison-Wesley, 1986.

[Krueger92] Charles W. Krueger: "Software Reuse," *Computing Surveys*, Vol. 24, pp. 131-183, June 1992.

[Mili95] Hafedh Mili, Fatma Mili, Ali Mili: "Reusing Software: Issues and Research Directions," *IEEE Transactions on Software Engineering*, Vol. 21, No. 6, pp. 528–562, June 1995.

[Mittermeir90] Roland T. Mittermeir, Wilhelm Rossak: "Reusability," in Peter A. Ng, Raymond T. Yeh (Eds.): *Modern Software Engineering: Foundations and Current Perspectives*, Van Nostrand Reinhold, Chapter 7, pp. 205–235, 1990.

[Poulin92] Jeffrey S. Poulin: "Measuring Reuse," *5th Annual Workshop on Software Reuse*, WISR5, 1992.

[Vaishnavi96] Vijay Vaishnavi, Rajendra Bandi: "Measuring Reuse," *Object Magazine*, Vol. 6, No. 2, pp. 53–57, April 1996.

[Weinand89] Andre Weinand, Erich Gamma, R. Marty: "Design and Implementation of ET++, a Seamless Object-Oriented Application Framework," *Structured Programming*, Vol. 10, No. 2, 1989.