Quantification of Code Regularity Using Preprocessing and Compression

Ahmad Jbara[1,2] Dror G. Feitelson[2]
[1]School of Mathematics and Computer Science
Netanya Academic College, 42100 Netanya, Israel
[2]School of Computer Science and Engineering
The Hebrew University of Jerusalem, 91904 Jerusalem, Israel

***Abstract*—Complexity metrics are useful to identify potentially problematic code. But there is little agreement regarding which complexity metrics should be used and exactly what should be measured, partly because many different factors influence code complexity and comprehension. Code regularity has recently been identified as another such factor, which may compensate for other complexity factors, especially in long functions with high cyclomatic complexity. Given that regularity consists of repeated code structures, it has been suggested to measure regularity by compressing the code with standard text compression tools. But such compression can be done in many ways. We compare five widely available compression tools (*LZ77*, *gzip*, *LZMA*, *bzip2*, and *bicom*) and four levels of preprocessing the code (using the code as is, or reducing it to a skeleton of keywords and possibly some formatting). The comparison is done in terms of how the different combinations discriminate between different functions, how they correlate with human perceptions of complexity, and how well they handle relatively short functions. The results show that different combinations of compression tool and code preprocessing lead to significantly different levels of discrimination and correlation with human perceptions, and in addition some combinations are extremely bad in handling many functions and should be avoided. Our recommendation is to use *gzip* or *bicom* on a code skeleton containing keywords and formatting.**

***Index Terms*—Software complexity metrics, Code regularity, Compression.**

I. Introduction

Program comprehension is a vital preliminary step of software maintenance. The ability to comprehend a given program naturally depends on the programmer's experience, his or her knowledge of the problem domain, and the complexity of the program itself [21]. Our focus is on the measurement of program complexity, and in particular of one specific factor that has an influence on this complexity, namely the code's regularity.

Measuring code complexity is difficult because there are so many different factors that have an effect on developers who are trying to comprehend, correct, or modify the code. As a result there is no single metric of complexity, and in fact, any given metric will fail to match human perceptions of complexity in some cases [3], [14], [12]. Myriad metrics are therefore used to measure distinct aspects of complexity: McCabe's cyclomatic complexity (MCC) and nesting measure control flow complexity [17], [7], Halstead's metrics measure vocabulary and operator use [6], fan-in and fan-out measure data flow [8], and other metrics measure elements of style and formatting [9], [5]. Regularity was recently introduced as yet another code attribute that may affect comprehension [12], [13], [10], [22]. Specifically, it was demonstrated that developers faced with long regular functions perceive them as less complex than the conventional metrics (e.g. LoC and MCC) suggest, and also perform cognitive tasks better than when faced with shorter non-regular versions of the same functions. An example of such a regular function from the Linux kernel is shown in Figure 1.

In order to further investigate the importance and effects of code regularity we need an objective metric that can quantify the degree to which given code is regular. Intuitively, regularity means that the same structures in the code repeat themselves over and over again. It has therefore been suggested that regularity may be quantified by compressing the code, and noting the compression ratio [13]. This indirect methodology is based on the mechanisms used in compression algorithms, where repeated segments are replaced with pointers to earlier instances in order to derive a shorter representation.

Still, this basic idea may be implemented in many different ways. First, there is the question of which compression scheme to use. In the following we compare common tools such as *gzip* and *bzip2* and more exotic ones like *LZMA* and *bicom*. Then there is the question of possible preprocessing of the code, to better express the regularities in the control structure. We therefore compare compression of the raw code with compression of a skeleton containing only the keywords, possibly with some of the formatting.

Our goal in the present work is to find the best combination of compression scheme and preprocessing, within the framework of using compression to quantify regularity. Naturally, this does not go to say that there are no other ways to quantify regularity. However, finding the best parameters is enough to support continued work on code regularity, and is also important for future comparisons with competing approaches.

In order to identify the best combination, we use all of the available combinations to compress 18755 functions taken from seven systems in different domains. These functions have an MCC of 20 or more to exclude short and simple functions where regularity is not expected to play a part. Our results show that different combinations indeed lead to very different results, so it is important to select the compression methodology carefully. In particular, some of the combinations

```
static int amd8111e_calc_coalesce(struct net_device *dev)
{
  struct amd8111e_priv *lp = netdev_priv(dev);
  struct amd8111e_coalesce_conf * coal_conf = &lp->coal_conf;
  int tx_pkt_rate;
  int rx_pkt_rate;
  int tx_data_rate;
  int rx_data_rate;
  int rx_pkt_size;
  int tx_pkt_size;

  tx_pkt_rate = coal_conf->tx_packets - coal_conf->tx_prev_packets;
  coal_conf->tx_prev_packets =  coal_conf->tx_packets;

  tx_data_rate = coal_conf->tx_bytes - coal_conf->tx_prev_bytes;
  coal_conf->tx_prev_bytes =  coal_conf->tx_bytes;

  rx_pkt_rate = coal_conf->rx_packets - coal_conf->rx_prev_packets;
  coal_conf->rx_prev_packets =  coal_conf->rx_packets;

  rx_data_rate = coal_conf->rx_bytes - coal_conf->rx_prev_bytes;
  coal_conf->rx_prev_bytes =  coal_conf->rx_bytes;

  if(rx_pkt_rate < 800){
    if(coal_conf->rx_coal_type != NO_COALESCE){

      coal_conf->rx_timeout = 0x0;
      coal_conf->rx_event_count = 0;
      amd8111e_set_coalesce(dev,RX_INTR_COAL);
      coal_conf->rx_coal_type = NO_COALESCE;
      }
    }
  else{

    rx_pkt_size = rx_data_rate/rx_pkt_rate;
    if (rx_pkt_size < 128){
      if(coal_conf->rx_coal_type != NO_COALESCE){

        coal_conf->rx_timeout = 0;
        coal_conf->rx_event_count = 0;
        amd8111e_set_coalesce(dev,RX_INTR_COAL);
        coal_conf->rx_coal_type = NO_COALESCE;
        }

      }
    else if ( (rx_pkt_size >= 128) && (rx_pkt_size < 512) ){

      if(coal_conf->rx_coal_type !=  LOW_COALESCE){
        coal_conf->rx_timeout = 1;
        coal_conf->rx_event_count = 4;
        amd8111e_set_coalesce(dev,RX_INTR_COAL);
        coal_conf->rx_coal_type = LOW_COALESCE;
        }
      }
    else if ((rx_pkt_size >= 512) && (rx_pkt_size < 1024)){

      if(coal_conf->rx_coal_type !=  MEDIUM_COALESCE){
        coal_conf->rx_timeout = 1;
        coal_conf->rx_event_count = 4;
        amd8111e_set_coalesce(dev,RX_INTR_COAL);
        coal_conf->rx_coal_type = MEDIUM_COALESCE;
        }

      }
    else if(rx_pkt_size >= 1024){
      if(coal_conf->rx_coal_type !=  HIGH_COALESCE){
        coal_conf->rx_timeout = 2;
        coal_conf->rx_event_count = 3;
        amd8111e_set_coalesce(dev,RX_INTR_COAL);
        coal_conf->rx_coal_type = HIGH_COALESCE;
        }
      }
    }
  /* NOW FOR TX INTR COALESC */
  if(tx_pkt_rate < 800){
    if(coal_conf->tx_coal_type != NO_COALESCE){

      coal_conf->tx_timeout = 0x0;
      coal_conf->tx_event_count = 0;
      amd8111e_set_coalesce(dev,TX_INTR_COAL);
      coal_conf->tx_coal_type = NO_COALESCE;
      }
    }
  else{

    tx_pkt_size = tx_data_rate/tx_pkt_rate;
    if (tx_pkt_size < 128){

      if(coal_conf->tx_coal_type != NO_COALESCE){

        coal_conf->tx_timeout = 0;
        coal_conf->tx_event_count = 0;
        amd8111e_set_coalesce(dev,TX_INTR_COAL);
        coal_conf->tx_coal_type = NO_COALESCE;
        }

      }
    else if ( (tx_pkt_size >= 128) && (tx_pkt_size < 512) ){

      if(coal_conf->tx_coal_type !=  LOW_COALESCE){
        coal_conf->tx_timeout = 1;
        coal_conf->tx_event_count = 2;
        amd8111e_set_coalesce(dev,TX_INTR_COAL);
        coal_conf->tx_coal_type = LOW_COALESCE;

        }
      }
    else if ((tx_pkt_size >= 512) && (tx_pkt_size < 1024)){

      if(coal_conf->tx_coal_type !=  MEDIUM_COALESCE){
        coal_conf->tx_timeout = 2;
        coal_conf->tx_event_count = 5;
        amd8111e_set_coalesce(dev,TX_INTR_COAL);
        coal_conf->tx_coal_type = MEDIUM_COALESCE;
        }

      }
    else if(tx_pkt_size >= 1024){
      if (tx_pkt_size >= 1024){
        if(coal_conf->tx_coal_type !=  HIGH_COALESCE){
          coal_conf->tx_timeout = 4;
          coal_conf->tx_event_count = 8;
          amd8111e_set_coalesce(dev,TX_INTR_COAL);
          coal_conf->tx_coal_type = HIGH_COALESCE;
          }
        }
      }
    }
  return 0;

}
```

Fig. 1. Example of a regular function from the Linux kernel.

fail to effectively compress thousands of functions, because they (or some of their preprocessed versions) are too short. As being able to handle functions of modest length is important, these combinations should be avoided.

The remainder of this paper is structured as follows. In the next section we motivate our work and present its research questions. Our methodological approach, including a description of the compression schemes and preprocessing levels, is presented in section III. We present the results and analyze them in section IV, also showing the correlation of regularity with perceived complexity and documentation of the code. Finally, we discuss the results and conclude in section VI.

II. Motivation and Research Questions

In view of the large number of metrics that have been defined for measuring code complexity, it is now accepted that there is no one metric or factor that fully reflects the complexity of source code [4], [18].

In previous work we have suggested *regularity* as an additional factor that affects code comprehension, especially in long functions, and provided experimental evidence for its significance [13], [10]. Specifically, we conducted several experiments where developers with different levels of experience were required to understand functions and to perform maintenance tasks on functions, where different subjects were actually working on different versions of the same function. Thus we could evaluate the dependency between performance and the style in which the function was coded. Additional experiments required subjects to evaluate and grade a set of functions. The produced rankings provide us with "ground truth" regarding how human developers perceive code complexity. Using this information we can now ensure that our metrics reflect human perception, a quality that is missing in many metrics that were proposed on theoretical grounds.

The preliminary operational definition of regularity we used in that study was based on compression. We applied the *gzip* tool to compress 30 functions and used the compression ratio as a metric for regularity. This was actually done twice: first with the full function code and then using only the control structure while removing formatting, layout, and expressions.

Comparing the compression ratios with the human grading, we found a weak correlation between the grades and the compression ratios achieved on the whole function. We found a moderate correlation between the grades and the compression ratios of the control structure, and even better correlation when using the grades given based on the visual representation of the functions.

These results show that the methodology of calculating the compression has an effect on the results. Consequently, a systematic investigation of the methodology is needed.

Our ultimate goal is to define an objective metric for code regularity that reflects perceived complexity. Based on the framework of using compression ratios to quantify regularity, this may be itemized into the following research questions:

1) Does it matter what compression scheme is used?

2) What elements of the source code should be compressed?
3) What is the best combination of compression scheme and code preprocessing level that would reliably reflect code regularity?

To answer these questions, we need a way to evaluate the available compression schemes and combinations. We use the following three criteria:

1) Good discrimination. We expect functions with different levels of regularity to exhibit different compression ratios. A good compression algorithm that compresses all the functions to the same degree would be useless for us, even if the compression ratios are all very high. To check this we use thousands of functions from multiple sources, and look at the distributions of compression ratios produced by the different compression-preprocessing combinations.
2) The compression ratio should negatively correlate with perceived complexity: a higher compression ratio means higher regularity which should yield better comprehension. To verify this we use the same 30 functions we used in the previous work [13], and check the correlation of complexity scores we have with the compression ratios achieved by the different compression-preprocessing combinations.
3) Success on as many functions as possible. Some compression schemes fail to compress some functions, especially when only a minimal skeleton is used, probably because they are too short. We obviously prefer metrics that can work on any function.

III. Methodological Approach

A. Compression Schemes

As explained above our operational quantification of regularity is based on compression. However, there are many compressing schemes available. The compression schemes that we examine in this work are three that are based on the Lempel-Ziv algorithm: *LZ77*, *gzip*, and *LZMA*. Furthermore, we also examine *bzip2* and *bicom*. Table I summarizes these schemes and indicates the versions used.

TABLE I
Compression schemes used in this study.

Tool	*Version*	*Description*
LZ77	N/A	The basic Lempel-Ziv dictionary-based compression algorithm as implemented by Marcus Geelnard.
gzip	1.4	a variant of the Lempel-Ziv algorithm as included in the GNU project.
LZMA	4.32.0.beta3	Lempel-Ziv Markov-chain Algorithm, another improved version of the Lempel-Ziv algorithm.
bzip2	1.0.5	Compression algorithm using the Burrows-Wheeler block sorting transformation and Huffman coding.
bicom	1.01	Bijective compression based on prediction by partial matching.

Text compression usually works on complete files: an input file is compressed to create an output file. To work on functions, we create temporary files that include only the function of interest.

The most basic compression scheme we use is the original Lempel-Ziv algorithm *LZ77* [25]. This is a dictionary-based compression scheme, where repeated occurrences of a string are replaced with a pointer to the original occurrence. The key point is that the pointer consumes less space than the string itself assuming the matched string is long enough. Literals that are not matched are output verbatim. The dictionary need not be stored, as it can be reconstructed during the decompression.

A very popular version of the Lempel-Ziv algorithm is implemented in the *gzip* tool, which is part of the common GNU software distribution. It combines LZ77 with Huffman coding. The output file format includes some static overhead (magic number, version number, timestamp, original file name, and CRC check) so in some cases, especially with small input files, the output may be larger than the input.

Another compression scheme based on the Lempel-Ziv algorithm that we use is *LZMA*. This is based on LZ77 followed by a range encoder. The dictionary size is huge relative to previous implementations, with special support for repeatedly used match distances. The encoding is done using context-based prediction.

Another compression scheme we use is *bzip2*. This uses, at its core, the *Burrows-Wheeler* block sorting transformation, which treats blocks of input to create sequences of repetitions of the same symbol. This is then put through run-length encoding and Huffman coding. Similar to the *gzip* tool, *bzip2* always performs the compression even if the compressed file is larger than the input.

The last compression scheme we use is *bicom*. This is a bijective compressor from the PPM family. Bijective means that it can always operate both ways: any file can be both compressed and decompressed. In other words, it does not produce any specified file format. PPM means prediction by partial matching. This is an adaptive statistical compression scheme, where the last n symbols are used to predict what will come next. Arithmetic coding is used to represent the output. This compressor is efficient even for very short input sequences. It was developed for a Windows platform, but we easily compiled and used it on a Linux system.

B. Code Preprocessing Levels

Regularity in code may occur in different forms, such as repeated block structures, formatting, identifier names, and operators usage. We believe that regularity in structure, which is dominated by the control-flow constructs, has a large effect on the overall understanding of the code. When using compression to quantify regularity, the question is then what parts of the code should be compressed to best reflect regularity and provide a good correlation with humans' opinions.

We define four levels of code preprocessing. The first level is the *raw code*, where we take the source code as is (including

TABLE II
KEYWORDS IN THE C LANGUAGE AND THEIR LETTER-CODE MAPPINGS.

Keyword	*Mapping*
`if`	A
`else`	B
`while`	C
`for`	D
`switch`	E
`case`	F
`do`	G
`?`	H

TABLE III
THE SYSTEMS FROM WHICH FUNCTIONS WERE TAKEN.

Name	*Version*	*Domain*	*# functions*
Windows	WRK-v1.2	Op. syst.	420
FreeBSD	9 (stable)	Op. syst.	1413
OpenSolaris	8	Op. syst.	864
Linux	2.6.37.5	Op. syst.	2819
Firefox	9 (stable)	Browser	681
GCC	4.8.0	Compiler	1391
OpenSSL	1.0.0k	Library	194

comments and blank lines) and compress it. The other extreme is the *unformatted control flow skeleton*, where we remove all expressions, layout, and comments. Thus we are left with just the sequence of control flow constructs (keywords) and braces (to preserve the nesting), with no linebreaks. In between are two levels where we retain the formatting (linebreaks and indentation), in order to better reflect the block structure. The difference between them is that one contains only the *formatting*, while the other also indicates the existence of individual *statements* (by retaining each statement's semicolon).

A potential problem with preserving keywords from a given programming languages is that keywords have different lengths, and there may be common substrings that cause keywords to overlap. These characteristics may be expected to affect the compression without reflecting any regularity. For example, multiple repetitions of `switch` may be compressed more than a similar sequence of the shorter `if`. To prevent such bias in the compression process we replace all the occurrences of each keyword with a single letter. For example, the keyword `if` is replaced by the letter code `A`. Table II shows the mapping between keywords and letter codes. (It should be noted that such a replacement was not used in our previous study [13].) The results of applying the different transformations are exemplified in Figure 2.

C. Data Collection

We have 5 compression schemes combined with 4 code preprocessing levels yielding 20 different combinations. We examine these combinations on 18755 C functions from different systems taken from different domains. The different systems, their domains, and the number of functions extracted (filtered) from each system are summarized in Table III.

Initially, our scripts extracted all functions of all systems. However, there is an intrinsic problem in C source code that is caused by the C preprocessor (CPP) conditional compilation directives. This interweaving causes problems in particular due

1 Raw
```
is_prime(int n)
{
   int i, flag=0;
   for (i=2; i<=n/2; ++i) {
      if (n%i==0) {
         flag=1;
         break;
      }
   }
   if (flag==0)
      printf("%d is prime",n);
   else
      printf("%d not prime",n);
}
```

2 Skeleton
```
{
   ;
   D;;{
      A{
         ;
         ;
      }
   }
   A
      ;
   B
      ;
}
```

3 Format
```
{
   D{
      A{
      }
   }
   A
   B
}
```

4 Keywords
```
{D{A{}}AB}
```

Fig. 2. Example of the four levels of preprocessing the code.

to unbalanced braces. To avoid this we dropped each source file that has such problems.

Functions that passed the first step were filtered by their cyclomatic complexity value. We took functions with MCC 20 or higher to ensure a minimum size of the function's structure, as regularity is especially meaningful for long functions and compression may fail on very small functions. We use the *pmccabe* [1] tool to calculate the MCC values of the different functions.

After these two filtering steps we had 18755 different functions. However, for many of these functions some of the different compression schemes yielded negative reduction percentages. Removing all these problematic cases led to a reduced set of 7744 functions.

Looking at the 11011 "bad" functions we found that almost all of them (about 95%) have MCC lower than 40, which means that they are relatively small functions. As compression schemes perform better on large inputs, this might explain the negative values they received. To support this conjecture we looked at all combinations to see where the negative values come from, and found them all in the most extreme preprocessing level (keywords and braces only) compressed by the *bzip2* scheme and in some cases also the *LZMA* scheme. In this level each function is in its shortest form, as we remove all its content including formatting and layout and preserve only control flow keywords which are then replaced with a single letter. Thus the functions may be reduced to a few dozen characters. The problems with *bzip2* and *LZMA* do not mean that such behavior does not occur in other schemes, but it is not as prevalent. Table VII shows the different combinations

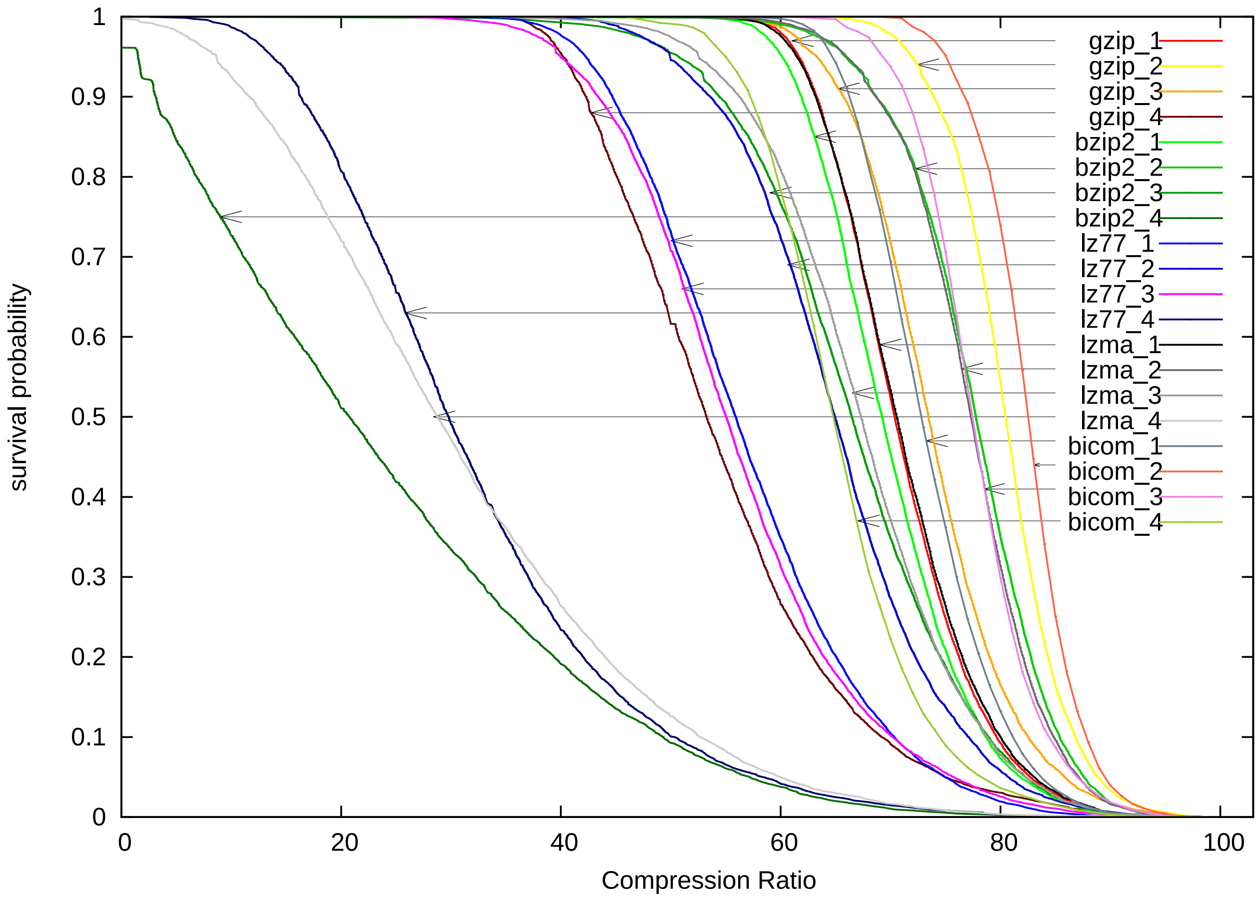

Fig. 3. Complementary cumulative distribution plots of compression ratios for 7744 functions using 20 combinations of preprocessing level and compression scheme.

and the number of functions out of the total 18755 that each combination failed to compress. Apparently the other schemes handle small inputs better. We show later that, for schemes that do not fail often, using the full 18775 functions or the reduced set of 7744 functions does not lead to significant changes in the results.

IV. Results and Analysis

A. Discrimination

As described above we have 20 candidate combinations of compression scheme and preprocessing level. We applied these to 7744 functions taken from 7 different systems that belong to 4 domains. Each function was preprocessed at 4 different levels, and each of the results was then compressed by 5 different compression schemes.

Figure 3 shows the distribution of the results using the complementary cumulative distribution function (survival function) of the obtained compression ratios. This distribution function shows the probability to observe a sample that is bigger than a given value.

According to this figure both the compression scheme and the preprocessing level have a significant effect on the achieved compression. The different schemes and the different preprocessing levels lead to different distributions. This means that selecting the best compression scheme and preprocessing level is indeed important. Arbitrarily selecting a popular compression scheme with some or no preprocessing is inappropriate.

When comparing compression schemes, the figure shows that some compression schemes consistently compress better than others, relatively independently of the preprocessing level. For example, the *gzip*, and *bicom* schemes compress very well at all preprocessing levels, so all their distributions are concentrated between moderate and high compression ratios. There is even a slight advantage for the *bicom* scheme (compresses better than *gzip*).

The *bzip2* and *LZMA* schemes exhibit similar behavior for three of the preprocessing levels. But with the keywords only preprocessing level (level 4) the distributions also include low compression ratios. Interestingly, the *LZ77* scheme has more diverse distributions: one is relatively high, two distribute over moderate up to high values, and one concentrates at rather low values.

When using the results to compare preprocessing levels, we observe that the *raw code* preprocessing level (level 1)

TABLE IV
DISCRIMINATION ABILITY OF THE DIFFERENT COMBINATIONS, AS MEASURED BY THE DIFFERENCE IN COMPRESSION RATIOS AT THE 15TH AND 85TH PERCENTILES OF THE DISTRIBUTIONS OF FIGURE 3.

	7744 functions		*18215 functions*	
Combination	*Width*	*per func.*	*Width*	*per func.*
lz77_1	20.7	0.0038	21.2	0.0016
lz77_2	18.0	0.0033	21.3	0.0016
lz77_3	20.7	0.0038	24.7	0.0019
lz77_4	26.9	0.0049	29.9	0.0023
gzip_1	13.2	0.0024	13.7	0.0010
gzip_2	9.7	0.0017	12.9	0.0010
gzip_3	13.1	0.0024	17.8	0.0013
gzip_4	21.8	0.0040	31.3	0.0024
lzma_1	13.6	0.0025		
lzma_2	12.1	0.0022		
lzma_3	17.9	0.0032		
lzma_4	33.3	0.0061		
bzip2_1	13.6	0.0025		
bzip2_2	12.8	0.0023		
bzip2_3	19.3	0.0035		
bzip2_4	38.6	0.0071		
bicom_1	11.0	0.0020	11.0	0.0009
bicom_2	7.0	0.0013	10.0	0.0007
bicom_3	9.0	0.0016	12.0	0.0009
bicom_4	13.0	0.0023	18.0	0.0013

compresses relatively highly across the different schemes. One explanation is that the code is the longest at this level when compared with others, and therefore has more potential for compression. Moreover, the input content at this level is real source code and English text, which are what most compression schemes are optimized to handle (as stated explicitly in the *gzip* manuals for example).

High compression ratios are obviously a desirable trait for compression schemes. But in the context of using compression to measure regularity, uniformly high compression ratios may be counterproductive. Instead, what we want is a good discrimination between input functions that have different degrees of regularity. (In the next section we add to this the requirement that this discrimination also corresponds to complexity as perceived by human developers.)

To assess the discrimination provided by the different combinations of compression and preprocessing, we focus on the central 70% of each distribution. This is the steepest part of the graph, excluding the bottom 15% which are always considerably lower and those above the 85th percentile which are always considerably higher. A large difference between the 15th and 85th percentiles of the compression ratio distribution indicate that good discrimination is possible. A small difference runs the risk that small changes in the code may lead to large and inappropriate changes in the placement in the distribution. In addition, we also divide this span of compression ratios (*width* column in Table IV) by the number of functions, to see the average difference per function (*per function* column in Table IV).

Table IV shows the results for the different combinations, both for the common set of 7744 functions and for the larger set of 18215 functions (out of a total of 18755) that are handled successfully by *LZ77*, *gzip*, and *bicom*.

According to this table combinations at level 4 (keywords and braces only, with no formatting) exhibit the best discrimination, sometimes by a wide margin. One explanation for this is that because it is the shortest representation of the code, compression ratios are necessarily lower, and every little difference in length or regularity has an effect.

Levels 1 (raw) or 3 (keywords with formatting) vie for second place. With *bicom* raw code provides a bit more discrimination, whereas with *LZMA* and *bzip2* the formatted skeleton appears a bit better. With *LZ77* and *gzip* they are essentially the same. Level 2 (including also semicolons for statements) is nearly always the least discriminative.

The results are not changed when looking at different sets of functions. Obviously, in order to achieve a fair comparison, all the combinations should be evaluated on the same set of functions. However, some of the combinations turn out to mishandle a large fraction of the original 18755 functions (this is discussed further below). The problem is that maybe limiting the evaluation to the subset of 7744 functions that all combinations can handle may distort the results regarding the better schemes, which can actually handle many more functions. We therefore also checked the distributions for a much wider set of functions, which are well handled by only three compression schemes. The results for this set are also presented in Table IV. They are consistent with those discussed above for the smaller set of functions.

B. Correlation of Regularity with Human Perception

In this section we examine the different combinations of compression schemes and preprocessing levels against other factors that are supposed to be related to regularity: perceived complexity and documentation in the source code.

1) Regularity and Perceived Complexity: In one of the experiments conducted in our previous work 30 diverse functions with high MCC (McCabe's cyclomatic complexity) values were presented to 15 experienced programmers [13]. The experiment was conducted in two phases, on different days. In one phase each subject was presented with listings of the functions and in the other phase he was presented with code structure diagrams (CSD: a visual representation of the code structure [12], [11]). Which representation came first was randomized across subjects. The subjects were asked to assign a perceived complexity score to each function in each representation. The result was a moderate negative correlation between perceived complexity and regularity, where regularity was measured by compression using *gzip* of a keywords plus braces representation of the code.

We can now compute the correlations between the rankings, from our previous work [13], and all 20 combinations of compression and preprocessing, to see which combination best matches the rankings of the human programmers. The two correlation methods (Spearman and Pearson) yielded very close results but we preferred the Spearman rank correlation because the data tends to be non linear. The results are presented in Table V and Figure 4, for both modes of presenting the functions (visual and listing).

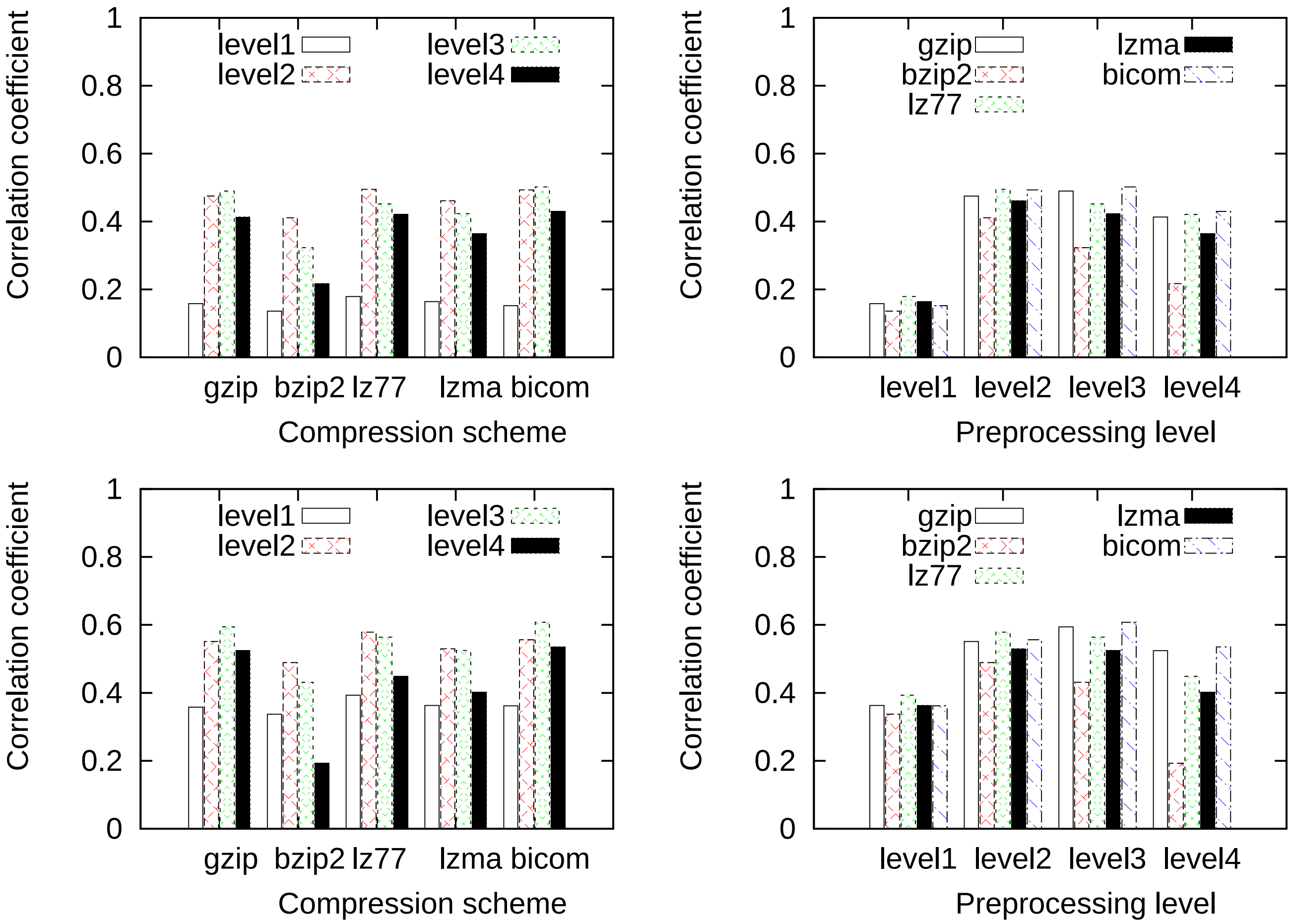

Fig. 4. Correlation between perceived complexity and compression ratios. Top row shows results when using code listings and second row when using CSD visualizations.

TABLE V
CORRELATIONS BETWEEN PERCEIVED COMPLEXITY AND COMPRESSION RATIO FOR DIFFERENT COMBINATIONS OF COMPRESSION SCHEMES AND PREPROCESSING LEVELS.

Combination	*Code*	*CSD*
lz77_1	-0.179	-0.393
lz77_2	-0.495	-0.579
lz77_3	-0.452	-0.564
lz77_4	-0.421	-0.449
gzip_1	-0.158	-0.363
gzip_2	-0.475	-0.551
gzip_3	-0.490	-0.594
gzip_4	-0.413	-0.524
lzma_1	-0.164	-0.363
lzma_2	-0.461	-0.530
lzma_3	-0.423	-0.525
lzma_4	-0.364	-0.402
bzip2_1	-0.136	-0.337
bzip2_2	-0.411	-0.489
bzip2_3	-0.323	-0.431
bzip2_4	-0.217	-0.193
bicom_1	-0.152	-0.362
bicom_2	-0.493	-0.556
bicom_3	-0.502	-0.608
bicom_4	-0.430	-0.535

According to these results, using the raw code (preprocessing level 1) has very low correlation across all schemes. The other three preprocessing levels achieve reasonable correlations for all compression schemes except *bzip2*. When using the *code listing representation*, levels 2 and 3 achieve the best correlation for all schemes and level 4 achieves slightly lower results. Similar results are achieved when using *CSD representation*. As for the highest correlation overall, it occurs at level 3 of *bicom* for both the visual mode and code listing representations. *gzip* also achieved high correlations which are not far from those of *bicom*.

2) Regularity and Comments: Regularity is characterized by repeated code; similar code segments may consecutively occur within a function. It seems reasonable to assume that programmers document such regular functions less than other non-regular ones. The rationale is that once the first instance of some repeated code segment is documented the programmer fairly believes that following instances of that pattern are understandable by implication, so he would provide less comments for these instances or even not provide comments at all.

TABLE VI
CORRELATION BETWEEN COMMENTS AND COMPRESSION RATIOS FOR DIFFERENT COMBINATIONS OF COMPRESSION SCHEMES AND PREPROCESSING LEVELS.

Combination	*Correlation coefficient*
lz77_1	-0.471
lz77_2	-0.298
lz77_3	-0.240
lz77_4	-0.224
gzip_1	-0.428
gzip_2	-0.253
gzip_3	-0.177
gzip_4	-0.201
lzma_1	-0.358
lzma_2	-0.209
lzma_3	-0.122
lzma_4	-0.142
bzip2_1	-0.298
bzip2_2	-0.118
bzip2_3	-0.034
bzip2_4	-0.034
bicom_1	-0.392
bicom_2	-0.284
bicom_3	-0.217
bicom_4	-0.197

TABLE VII
NUMBER OF FUNCTIONS EACH COMBINATION FAILS TO COMPRESS, PRODUCING NEGATIVE REDUCTION.

Combination	*# Bad functions*
lz77_1	0
lz77_2	4
lz77_3	15
lz77_4	534
gzip_1	0
gzip_2	4
gzip_3	15
gzip_4	59
lzma_1	0
lzma_2	8
lzma_3	96
lzma_4	7994
bzip2_1	0
bzip2_2	45
bzip2_3	287
bzip2_4	10942
bicom_1	0
bicom_2	0
bicom_3	0
bicom_4	0

Thus we conjecture that the more the function is regular the less likely it is to be documented. To examine this idea we measure the comments of each function of this study and check if there is any correlation between this measure and the regularity as quantified by the 20 different combinations of compression scheme and preprocessing.

There are many ways to measure comments: character-based length, word-based length, non-stop-word length, or just the number of comments. However, no matter which metric one chooses, it should be normalized relative to the function length (LOC). In other words, we are interested more in the density of commenting than in their absolute number. This point of view is required to reliably reflect situations where we have equal regularity measures for functions with different lengths. In this work we calculated the character-based length of all comments divided by the logical lines of code of that function.

We applied the Spearman nonparametric rank correlation coefficient between the comments ratio and the compression ratio for all 20 combinations of compression schemes and preprocessing levels of the 7744 set of functions. The results are shown in Table VI. Generally, a moderate correlation is achieved when the raw version of the functions is used. The best correlation is achieved in level 1 for the *gzip* and *LZ77* schemes, with a slight advantage of the latter one. Many other combinations also showed some correlation but it was weaker. *bzip2* showed essentially no correlation in its non-raw versions.

These results show that this direction is promising and apparently programmers indeed document regular code less than non-regular code. However, more work should be done in the direction of the best way of measuring comments and the ways developers document regular code.

C. Handling Small Functions

Generally, compression schemes are good with large inputs. However, most of the functions in any system are not considered large. For example, in this work we collected 18755 different functions from different systems where more than half of them have a cyclomatic complexity below 40. Many more functions have MCC below 20 and were not included in our sample to begin with. (The cyclomatic complexity is a relevant threshold criterion as we look at the control structure of each function).

Functions with relatively low MCC values lead to small input files that might cause the compression algorithms to create compressed files that are larger than the original ones. In particular, one should remember that most of the compression schemes have some headers that enlarge the output files without reflecting real compression.

The problem is critical in levels 2, 3, and especially 4, as in these levels much of the functions' contents are removed and the resulting input files are very small. This greatly reduces the effectiveness of the compression schemes in identifying and quantifying regular code as they fail to compress these files by shortening them.

We have already seen that more than half of the functions checked failed to be compressed in level 4 of *bzip2*, and more than 40% failed in level 4 of *LZMA*. It is important to mention that other schemes and levels fail also, but not as massively as *bzip2* and *LZMA*. Table VII shows the numbers of the "bad" functions under the different combinations. Note that in this work we considered only functions with MCC above 20. We expect that functions in the range between 10 and 20 would cause many more failures for the different compression schemes.

While *gzip* fails for a relatively small number of functions, *bicom* stands out for its ability to compress small files — it did not fail for a single function, regardless of preprocessing level. We believe that these differences and the inability of the current compression schemes to deal with small files indicate that there is room for considering other compression schemes, especially ones that are good at small files and maybe even irreversible (lossy) compression schemes. This is permissible in our context because we are not concerned with storing the information, just with measuring the regularity.

At the same time, we note that the shorter the function, the smaller the scope it has for regularity. Moreover, short functions are typically considered easier to understand, so the question of regularity is less pressing for short functions.

V. Related Work

To the best of our knowledge we are the first to study regularity in the context of code comprehension. In [12] we examined more than 1000 versions of the Linux kernel where we identified functions with very high cyclomatic complexity values. We found that these functions are not really as complex as their MCC complexity metric suggests. In particular, many turned out to be well structured and very regular. We then suggested the use of compression (using *gzip*) as an operational metric to enable the quantification of regularity. A survey we conducted provided empirical evidence for correlation between the measured regularity scores and perceived complexity by developers. In [13] we extended this work to encompass more systems and more domains, finding that regular functions also occur in systems and domains other than Linux.

Based on these results, we set out to verify the conjecture that regularity is one of the factors that allows developers to handle long high-MCC functions successfully. In a controlled experiment we compared the performance of subjects in terms of time and correctness when working on different implementations of the same specification, where one of the implementations adopted a regular style [10]. We found that the subjects working with the regular version achieved better results than others. The tasks that were used to assess comprehension in this experiment were *feature adding*, *bug fixing*, and *functionality description*.

Similar observations and results were reported in [22]. They introduced the idea of Control Flow Pattern (CFP) and Compressed Control Flow Pattern (CCFP). They used CCFPs to eliminate some repetitive structure from flow graphs. They concluded that methods with high cyclomatic complexity have very low entropy and are easy to understand.

Sasaki et al. also ascribed the large cyclomatic values that some modules exhibit to the presence of repeated structures such as consecutive if-else structures [20]. They claimed that it would not be so difficult to understand such source code. They proposed to preprocess the code to make complexity measurement more efficient.

Regularity has been noticed before, but not quantified. Chaudhary et al. conducted an experiment to study the effect of control and execution structures on program comprehension [2]. One result that contradicted their intuitive expectation was the positive correlation between the subjects' score and the control structure complexity. They attributed this result to the existence of syntactic and semantic regularities in the code. They claimed that these regularities reduced the efforts in the learning process and yielded a higher score.

Regularity has also been considered in other areas. Lipson has defined structural regularity as the compressibility of the description of the structure [15]. In addition to the regularity definition, a metric for quantifying the amount of regularity was suggested. It was defined by the inverse of the description length or Kolmogorov complexity.

Recently, Zhao et al. have shown that regularity leads to spontaneous attention [24]. This may be part of the explanation of why regular code is easier to understand.

There are also works that have used the term "regularity" with different meanings. For example, Lozano et al. use regularity in the context of naming conventions, complementary methods, and interface definitions [16]. Zhang suggested a revised version of Halstead's length equation. He based it on the fact that the distribution of lexical tokens in the studied systems follow Zipf's law [23]. Similar results, regarding the distribution of lexical tokens, were presented by [19].

VI. Discussion and Conclusions

We have already shown in a previous work that *regularity* is yet another factor that may have a substantial effect on code comprehension. We also suggested to measure it using compression. In this study we have performed a methodological investigation of this idea, and considered 20 different combinations of compression scheme and code preprocessing level. We used 5 compression schemes, namely *LZ77*, *LZMA*, *gzip*, *bzip2*, and *bicom*. We used 4 levels of preprocessing which are based on control-flow structure, formatting, and statement awareness. The effectiveness of the 20 combinations was evaluated by how well they discriminate between functions, how well their compression ratios correlate with perceived complexity, and how well they handle small functions.

The results show that *bzip2* and *LZMA* are problematic even with not-so-small functions, so they are less useful and should not be used. *bicom* is best on small files, but has somewhat lower discrimination than *gzip*. *gzip* is also very good, except with the most extreme preprocessing.

The *bicom* scheme achieved the highest correlations with perceived complexity so it best reflects effect on humans. *gzip*'s performance was very close to that of *bicom*. Similar results were also achieved by *LZ77*, with an advantage of being more discriminative.

As for preprocessing levels, level 1 (using the raw code) leads to very low correlations, so this should not be considered and preprocessing should definitely be used. Several interactions occur between the correlations and other attributes. The most extreme preprocessing, level 4, led to the best discrimination, but had somewhat lower correlations than levels 2 and 3. Preprocessing level 2 gives the highest correlations for *LZ77*, *LZMA*, and *bzip2*, but level 3 was better for *gzip* and *bicom*.

Our conclusion is that *gzip* or *bicom* combined with preprocessing level 3 (retain keywords, braces, and formatting, but not statements) are the best combination. *bicom* may be better at handling small functions, and has the advantage of not adding a header that distorts the compression ratio (due to its bijective nature). But *gzip* is more widely available. Luckily, the combination we used in previous work turns out to be near optimal, and therefore the results are valid.

These results indicate that compression is a promising way to measure regularity, but it is important to choose an adequate scheme. Not all schemes correlate with perceived complexity and not all of them have the same discrimination ability. Furthermore, the whole code of the functions is not representative and a preprocessing step on the code should be performed prior to compressing.

VII. Threats to Validity

Our work suffers from several threats to validity. We preprocessed the code in various levels by removing different things and retaining others. It might be that some of the removed stuff has an effect on regularity and we missed that. For example, Green et al. present a set of coding guidelines that are partially based on formatting. They suggest considering a program as a table by using vertical alignments, and considering the use of white space to show structure [5]. These guidelines represent factors that are part of regularity. Our work considers indentation and structure but not all these factors.

Another threat is that most compression schemes add headers to the compressed output, which distorts the compression ratio. This can be avoided by careful parsing of the output files to better reflect the true representation of the compressed data.

A third threat is that we evaluated the effectiveness of different combinations based on their correlation with a previous study in which complexity grades were given to 30 functions. Comparisons with larger sets of functions, and with multiple methods of assessing their complexity, would increase confidence in the results and allow for more general conclusions.

For future work, an interesting issue is measuring regularity of small functions. Indeed, we have shown a scheme that is capable of compressing small functions, but it has somewhat lower discrimination ability. Moreover, in this work we examined only functions with cyclomatic complexity of 20 or more, but a significant fraction of functions is below that.

Another avenue is to examine other families of compression schemes which were not examined in this study. For example, lossy compression scheme may be adequate as humans do not really read repeated code line by line and allow themselves to skip predictable parts.

Acknowledgments

this research was supported by the ISRAEL SCIENCE FOUNDATION (grant no. 407/13).

References

[1] P. Bame, "*pmccabe*". URL http://parisc-linux.org/~bame/pmccabe/overview.html. (Visited 18 Sep 2011).

[2] B. Chaudhary and H. Sahasrabuddhe, "*Two dimensions of program comprehension*". *Intl. J. Man-Machine Studies* **18(5)**, pp. 505–511, 1983.

[3] G. Denaro and M. Pezzè, "*An empirical evaluation of fault-proneness models*". In 24th *Intl. Conf. Softw. Eng.*, pp. 241–251, May 2002, doi:10.1145/581339.581371.

[4] N. E. Fenton and S. L. Pfleeger, *Software Metrics: A Rigorous and Practical Approach*. Course Technology, 2nd ed., 1998.

[5] R. Green and H. Ledgard, "*Coding guidelines: Finding the art in science*". *Comm. ACM* **54(12)**, pp. 57–63, Dec 2011, doi:10.1145/2043174.2043191.

[6] M. Halstead, *Elements of Software Science*. Elsevier Science Inc., 1977.

[7] W. Harrison, K. Magel, R. Kluczny, and A. DeKock, "*Applying software complexity metrics to program maintenance*". *Computer* **15(9)**, pp. 65–79, Sep 1982.

[8] S. Henry and D. Kafura, "*Software structure metrics based on information flow*". *IEEE Trans. Softw. Eng.* **SE-7(5)**, pp. 510–518, Sep 1981, doi:10.1109/TSE.1981.231113.

[9] A. Hindle, M. W. Godfrey, and R. C. Holt, "*Reading beside the lines: Indentation as a proxy for complexity metrics*". In 16th *IEEE Intl. Conf. Program Comprehension*, Jun 2008.

[10] A. Jbara and D. G. Feitelson, "*On the effect of code regularity on comprehension*". In *Proceedings of the 22Nd International Conference on Program Comprehension*, pp. 189–200, ACM, New York, NY, USA, 2014, ISBN 978-1-4503-2879-1, doi:10.1145/2597008.2597140.

[11] A. Jbara and D. G. Feitelson, "*JCSD: Visual support for understanding code control structure*". In *Proceedings of the 22Nd International Conference on Program Comprehension*, pp. 300–303, ACM, New York, NY, USA, 2014, ISBN 978-1-4503-2879-1, doi:10.1145/2597008.2597801.

[12] A. Jbara, A. Matan, and D. G. Feitelson, "*High-MCC functions in the Linux kernel*". In *Proceedings of the 20th IEEE International Conference on Program Comprehension, ICPC 2012.*, Jun 2012.

[13] A. Jbara, A. Matan, and D. G. Feitelson, "*High-MCC functions in the Linux kernel*". *Empirical Softw. Eng.* 2013, doi:10.1007/s10664-013-9275-7. Accepted for publication.

[14] B. Katzmarski and R. Koschke, "*Program complexity metrics and programmer opinions*". In 20th *IEEE Intl. Conf. Program Comprehension*, Jun 2012.

[15] H. Lipson, "*Principles of modularity, regularity, and hierarchy for scalable systems*". *Journal of Biological Physics and Chemistry* **7(4)**, pp. 125–128, 2007.

[16] A. Lozano, A. Kellens, K. Mens, and G. Arevalo, "*Mining source code for structural regularities*". In 17th *Working Conf. Reverse Engineering*, pp. 22–31, Washington, DC, USA, 2010, doi:10.1109/WCRE.2010.12.

[17] T. McCabe, "*A complexity measure*". *IEEE Trans. Softw. Eng.* **2(4)**, pp. 308–320, Dec 1976, doi:10.1109/TSE.1976.233837.

[18] N. Nagappan, T. Ball, and A. Zeller, "*Mining metrics to predict component failures*". In 28th *Intl. Conf. Softw. Eng.*, pp. 452–461, May 2006, doi:10.1145/1134285.1134349.

[19] D. Pierret and D. Poshyvanyk, "*An empirical exploration of regularities in open-source software lexicons*". In 17th *IEEE Intl. Conf. Program Comprehension*, pp. 228–232, 2009, doi:10.1109/ICPC.2009.5090047.

[20] Y. Sasaki, T. Ishihara, K. Hotta, H. Hata, Y. Higo, H. Igaki, and S. Kusumoto, "*Preprocessing of metrics measurement based on simplifying program structures*". In *Software Engineering Conference (APSEC), 2012 19th Asia-Pacific*, vol. 2, pp. 120–127, 2012, doi:10.1109/APSEC.2012.59.

[21] S. Tilley, S. Paul, and D. Smith, "*Towards a framework for program understanding*". In *Program Comprehension, 1996, Proceedings., Fourth Workshop on*, pp. 19–28, Mar 1996, doi:10.1109/WPC.1996.501117.

[22] J. J. Vinju and M. W. Godfrey, "*What does control flow really look like? Eyeballing the cyclomatic complexity metric*". In 12th *Working Conf. Source Code Analysis and Manipulation*, Sep 2012.

[23] H. Zhang, "*Exploring regularity in source code: Software science and Zipf's law*". In 15th *Working Conf. Reverse Engineering*, pp. 101–110, 2008, doi:10.1109/WCRE.2008.37.

[24] J. Zhao, N. Al-Aidroos, and N. B. Turk-Browne, "*Attention is spontaneously biased toward regularities*". *Psychological Sci.* **24(5)**, pp. 667–677, May 2013, doi:10.1177/0956797612460407.

[25] J. Ziv and A. Lempel, "*A universal algorithm for sequential data compression*". *IEEE Trans. Information Theory* **IT-23(3)**, pp. 337–343, May 1977, doi:10.1109/TIT.1977.1055714.