A model to predict anti-regressive effort in Open Source Software

Andrea Capiluppi
Centre of Research on Open Source Software
Department of Computing and Informatics
University of Lincoln, UK
acapiluppi@lincoln.ac.uk

Juan Fernández-Ramil
Computing Department
The Open University, UK
j.f.ramil@open.ac.uk

Abstract

Accumulated changes on a software system are not uniformly distributed: some elements are changed more often than others. For optimal impact, the limited time and effort for complexity control, called anti-regressive work, should be applied to the elements of the system which are frequently changed and are complex. Based on this, we propose a maintenance guidance model (MGM) which is tested against real-world data. MGM takes into account several dimensions of complexity: size, structural complexity and coupling. Results show that maintainers of the eight open source systems studied tend, in general, to prioritize their anti-regressive work in line with the predictions given by our MGM, even though, divergences also exist. MGM offers a history-based alternative to existing approaches to the identification of elements for anti-regressive work, most of which use static code characteristics only.

KEYWORDS: Anti-regressive work, Coupling, Empirical Studies, McCabe Cyclomatic Complexity, Maintenance, Metrics, Open Source, Software Evolution.

1. Introduction

Most software systems that are actively used need to be changed and enhanced many times after the first operational release in order, for example, to implement new and changed requirements [Lehman 1974, Lehman and Belady 1985]. A likely consequence of the accumulation of change upon change upon change is a deterioration in the structure of the software. Such deterioration manifests itself in growing complexity [Lehman 1974] and the loss of architectural and conceptual integrity [Parnas 1994]. This phenomenon is likely to make it increasingly difficult to implement new changes to the software.

Some of the drivers for complexity growth are related to people rather than to processes or tools. For instance, as time passes some of the original developers of a software system may distance themselves from a project, and as new developers get involved, there is a chance that tacit knowledge will be irremediably lost. The system's structure will be changed in a haphazard way and the architectural integrity will be damaged. Lack of knowledge about the system or lack of good programming practice may also lead to a sub-optimal implementation of changes, leading to an increase in system complexity greater than necessary. This risk needs to be addressed, in the first place by, for example, promoting developers' stability, clear plans, ensuring spread of good practice, appropriate documentation levels and properly training newcomers [e.g. Humphrey 1997]. This might not be enough to prevent excessive increase in complexity, and sporadic or even continual complexity control work (called anti-regressive effort [Lehman 1974]) will be needed. In this paper we focus on the technical aspect of allocating effort to complexity control work. We assume that developers will have the appropriate management and peer support, knowledge and skills to accomplish such work.

Given pressures to implement changes and new requirements, the amount of effort that can be dedicated to anti-regressive work in order to prevent system's decay is always limited. Developers need to identify and prioritize specific parts of the code to be acted upon first. They locally reduce the complexity of specific parts of the system, and from time to time, seek to improve the architectural integrity of sections of code, rather than globally improving the overall system. Refactoring is a well known technique mainly applied to object oriented (OO) programs. Approaches have been developed in order to identify "hot spots" that will benefit from refactoring [e.g. Demeyer et al 2005].
general, approaches fall into two types: those which detect so called “bad-smells” and those which are metric-based. The first type is based on specific rules or heuristics which can be used for detecting programming at a substandard level (e.g. intraclass duplication or the duplication of methods within the same class). The second type of approaches uses metrics such as size, number of branches, depth of the nesting levels and similar measurements in order to detect within the code elements which are good candidates for refactoring.

By implementing a metric-based approach, this paper reports on a study based on Open Source Software (OSS), with two main aims: first, to investigate empirically how developers of OSS projects perform anti-regressive activity; and second, to provide guidance to OSS developers on how best direct their anti-regressive effort to specific elements of their software systems. In the study reported in this paper, these two aims have been tackled through the design, implementation and test of an Maintenance Guidance Model (MGM).

The MGM focuses on the level of granularity of functions (or methods): it uses data on their accumulated change and complexity to prioritize them for anti-regressive work. Complexity was empirically measured by size, cyclomatic complexity [McCabe 1976] and coupling [Conte et al 1986]. For a system with N releases\(^1\), the model proposes a list of potential candidates for anti-regressive work in the following release (N+1), ordering functions by their complexity and by the accumulated changes. Those ranked at the top become the primary candidates for anti-regressive work at release N+1. At every release, the MGM model is updated using data on the latest release.

In order to test our MGM model empirically, and to assess how OSS developers actually perform anti-regressive work, an experiment was designed and conducted on eight OSS systems. In Table 4 (Appendix), the first five left-hand-side columns present some general characteristics, including domain and size at the latest release. The selection was made from a wider sample of 400 projects that were previously analyzed in order to characterize globally the OSS phenomenon [Capiluppi et al 2003, Capiluppi 2003]. The number of systems to be studied was limited to eight based on the limitations of time and effort to extract the relevant data and perform this study. The eight systems selected include small, medium a large systems (27,000 to 652,000 lines of code or LOC), in order to test the MGM over different-sized software projects. The experiment uses data from existing public releases, where the real anti-regressive work was identified, and compared with the candidates of our model. We call this evaluation “back-to-back” testing of our model.

This paper is structured as follows: Section 2 provides the definitions of the measurements used in this study, together with the rationale and assumptions of our modeling. Section 3 describes the MGM model and the steps followed to implement and validate it. Section 4 presents the results obtained in the eight different OSS projects. Section 5 briefly discusses the related work. Section 6 covers the threats to validity of the results, and Section 7 present the main conclusions and discusses further work.

2. Definitions and rationale

This section covers the justification of the use of the function level as the basis for our modeling, the description of the measurements and the model's rationale, including some assumptions of this work.

2.1. Code elements studied

Source code is organized in files which provide both storage and separate the code into portions that, for example, can be separately compiled. In general, within these files, programmers create functions or procedures that perform operations on the data so that the desired computations are performed. The functions contained in the same file may be diverse in size and complexity. Moreover, in OO refactoring approaches guidance is provided at the class and method level rather than at the file level. When it comes to anti-regressive work guidance, the function level seems to be a reasonable trade-off between too fine and too coarse granularity.

2.2. Complexity

Since long ago, complexity has been identified as one of the key factors that needed to be considered in managing applications [Lehman 1974]. Lehman's second law of software evolution states that as a software system is continually evolved, and more features are added, the complexity of the system is likely to increase unless work is done to control such complexity [Lehman 1974]. Uncontrolled complexity growth is likely to become an obstacle of further evolution and may trigger the need for either substantial software re-engineering or system replacement. Complexity has several dimensions. The three dimensions considered in the present study are:

(a) Size: in general, the larger the source code or its elements, the more effort it will take to produce or to understand it, assuming other characteristics being constant. Size provides an idea of the functional

\(^1\) Releases means here any of the sequence of versions that OSS projects publicly provide.
complexity of a software system and is generally considered a predictor of development effort and of the number of defects. In this study, the number of lines of code (LOC or loc) was measured for each single function, excluding both blank lines and comments.

(b) Cyclomatic Complexity [McCabe 1976] is a measure of structural complexity. It can be calculated from a graph representation of a function, with each executable statement being a node on the graph, and arrows between the nodes showing the execution pathways and decision or branching nodes. Cyclomatic complexity is calculated as the number of decision (or branching) nodes plus one. In this research McCabe complexity (Mc) was evaluated for all the functions of each software system studied.

(c) Coupling [Conte et al 1986, Arisholm et al 2004] measures the degree to which each source element relies on other elements, that is, how interconnected is the code. Since this study is conducted at the function level, the union of all the function calls (and method invocations) form the network of couplings in a system. Each coupling can be uniquely categorized as inbound (c_in) or outbound (c_out) (or fan-in and fan-out), depending on the direction of the relative call. As an example, function 'sign off' (Figure 1) has two inbound (fan-in) and three outbound (fan-out) couplings. In this study we separately measured the number of inbound and outbound couplings of each function.

<table>
<thead>
<tr>
<th>c_out</th>
<th>c_in</th>
<th>Mc</th>
<th>loc</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0000</td>
<td>0.0649</td>
<td>0.2936</td>
<td>0.3249</td>
</tr>
<tr>
<td>1.0000</td>
<td>-0.0269</td>
<td>-0.0471</td>
<td>c_in</td>
</tr>
<tr>
<td>1.0000</td>
<td>0.7135</td>
<td>Mc</td>
<td>loc</td>
</tr>
<tr>
<td>1.0000</td>
<td>loc</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 1: correlations for the four attributes measured at the function level for the XMMS project

2.3. Measuring cumulative change and anti-regressive work at the function level

Different approaches for classifying maintenance and evolution activity have been proposed over the years e.g., [Kemerer and Slaughter 1999, Chapin et al 2001]. The application of the proposed classification schemes in an empirical study like the present one would involve considerable manual work far beyond our available time and effort. For this reason we relied on simple ways of detecting change and anti-regressive work that could be performed by a computer. We extracted data, as said, at the level of granularity of functions. Our measurements of change and anti-regressive work are defined below:

(a) Number of releases with changes (NRwC): this is simply the number of release intervals during which a function has been changed, also called number of release touches. Every instance of a function changed in a given release incremented the NRwC counter for that function. In order to detect a change in a function, we identified changes in its overall size, its cyclomatic complexity, or the inbound or outbound couplings. More precisely, we said that a change on a function is detected and recorded in, at least, one of these cases:

- its amount of inbound couplings (fan-in) has increased or decreased;
- its amount of outbound couplings (fan-out) has increased or decreased;
- its cyclomatic complexity was increased or decreased, or
- its size (in LOC) has changed, upwards or downwards.

When the number of NRwC is zero, it indicates that the function has never been touched again (that is, after its appearance in the first public release) during the evolution of the system. The maximum possible number of NRwC for a function is the total number of releases minus one: in this case, this would mean that a function was already present at release 1 and then changed at every subsequent release so far.

(b) Anti-regressive work: in this study an “instance” of anti-regressive work is identified whenever there is a decrease of any of the four measured attributes (size,

Figure 1: example of 'inbound' and 'outbound' couplings for the function sign off (from Gaim)
McCabe index, fan-in and fan-out) in any of the functions of a system between two consecutive releases. The term anti-regressive was first used in the 1970s [Lehman 1974] to describe complexity control activity and we use it here as it best describes the focus of the work presented in this paper. Refactoring work, a popular term in OO, refers to code transformations that improve structure and preserve the user observed behavior. Refactoring is closely related to anti-regressive work but strictly speaking they are not the same. Anti-regressive work can be “externally” defined using measurements (decrease in complexity), whilst refactoring involves the performance of specific code transformations. Most refactoring work should lead to decrease in measurements of complexity and are anti-regressive in nature, but some others might not be so.

2.4. Assumptions and research questions

In a number of systems we have previously studied we have found that the distribution of accumulated changes in the code elements (files, classes, functions) over releases is not symmetrical [Capiluppi and Ramil 2004, Capiluppi et al 2005]. This suggests that there are loci of high change, that is, parts of a software system which are likely to attract more changes than others. To illustrate this, Figure 2 shows the profile of the distribution of NRwC for one of the systems studied (XMMS). As it is visible, a few functions concentrate the higher values for NRwC (i.e. are changed in most of the releases), giving the distribution a typical skewed shape. A symmetrical distribution like a Gaussian has a skewness coefficient of zero. The skewness of the distribution in Figure 2 is estimated as 3.11. In the Appendix, Table 4 (column 6) gives the NRwC distribution skewness for all the eight systems studied (values between 1.43 to 5.72).

In the absence of additional information, it seems reasonable to assume that functions experiencing frequent changes are more likely to attract changes in the future. If this holds, given two equally complex functions, reducing the complexity of the function which has been changed more frequently is likely to produce a better impact on future productivity, than working on the more stable (less change-prone) parts of the system. We call this the change-rate criterion for anti-regressive work.

Based on this, in the present study we look at the following research questions:

i. In which parts of the system complexity should be reduced, based on the change rate criterion?

ii. To what extent do developers follow the change rate criterion?

iii. Is there any particular dimension of the complexity that developers should rely on more than others when selecting functions for anti-regressive work?

The MGM model, discussed in the next section, helped us address these questions.

3. Maintenance Guidance Model (MGM)

As previously said, our proposed model is based on observations of both accumulated changed and complexity: for optimal use of limited resources, developers should reduce the complexity first on the elements of the system which are both often changed, and that encapsulate at the same time high complexity (assuming that all other factors are constant).

Our approach takes into account a modeled behavior and an observed behavior (top and bottom part of Figure 3, respectively), as follows:

i. The modeled behavior consists of a sorted list of all the functions that have been touched at least once, based on the accumulation of changes over the past releases and their complexity at the most recent release R(n). The list is reverse-ordered (highest first) by a complexity attribute (one of the four mentioned above) first, and then by accumulated changes. Figure 4 displays an example of sorting by (first) LOC and (then) changes. This ordered list contains all the functions at R(n), and provides the basis for the selection of the candidate functions: according to the MGM model developers should focus the anti-regressive work on the top-most subset in the next release (i.e., R(n+1)). Complex functions without any changes are not considered by our approach: for

---

3 In the present study, this assumption led to reasonably good results when used for the MGM purposes. However, if one wishes to be more strict this assumption should be empirically tested [e.g. Girba et al 2004].
example, function “proof-of-concept” in Figure 4 will not be considered in the model, since it has not received any changes.

ii. The observed behavior (bottom part of Figure 3) in turn identifies first the actual quantity (‘q’) of functions whose complexity was decreased by developers between R(n) and R(n+1). Since the complexity, as mentioned above, is composed of various dimensions, a list was generated for each of the four considered factors (inward couplings, outward couplings, cyclomatic complexity and lines of code). In our study we found that this second list is shorter than the modeled list and we interpret this as a consequence of the fact that anti-regressive effort impacts a small part of the total system.

iii. “Back-to-back” testing: from the “modeled-behavior” list, the first ‘q’ functions are selected and compared with the “observed-behavior” list. An evaluation of the prediction is therefore achieved against the real data observed through a hit rate factor, which is the number of hits (functions observed behavior functions which are also present in the first ‘q’ functions of the modeled list) divided by ‘q’.

4. Testing the MGM model

This section describes the results of applying the MGM, and illustrates the outcomes on one of the selected projects (Arla) in more detail. Due to space limitations, a summary table (Table 2) presents an overall view of the results for all the eight projects, obtained through the similar analysis as done for the Arla case study.

In order to test the MGM, the model as presented in Figure 3 was evaluated using real data. As described in the previous sections, the function's complexity and the number of NRwC in the first Nth releases was used to identify candidates for the (N+1)th release. More formally, the candidates were identified based on:

- the number of NRwC on functions in all the previous releases (e.g., from 1 to N);
- the distribution of complex functions at the Nth release.
- Next, the generated candidates were tested against the actual observed anti-regressive work on functions performed between the Nth and the (N+1)th releases.

The minimum amount of releases in order to perform this experiment is 3: the amount of anti-regressive work observed between the first two releases serves as a predictor and creates the candidates for the third release. Therefore, given N releases, a total of (N-2) “predictions” were evaluated for each complexity factor and each project.

4.1. The Arla case study

We studied an overall of 69 public releases of this project: the MGM provided 67 predictions for each of the four complexity characteristics mentioned above. The set of predictions of the model is summarized here with a boxplot for each factor (Figure 5). As it can be appreciated on the figure, for all the factors the hit-ratio ranges between 0 (i.e. a complete inability in some releases to predict the actual functions subject to anti-regressive work) to 1 (i.e. the predictions fully matched with the observed functions experiencing a decreasing complexity). Zero values in the predictions were mostly found whenever anti-regressive work in-between two subsequent releases was small (e.g., only one or two functions were subject to anti-regressive work), and our model was not able to identify the “correct” candidates.

The median value gives an idea of the middle value of the distribution, and is depicted as the horizontal line internal to the rectangle of the boxplot. As visible from Figure 5, the MGM model, applied to the Arla system, performs better when predicting the candidates for anti-regressive work based on a decrease of the cyclomatic complexity (the hit rate distribution has a median of 80 percent). Using this variable, an overall of 662
functions were observed to decrease their cyclomatic complexity during Arla’s evolution. In total, our MGM was able correctly predict an overall of 461 functions (70 percent, not to be confused with the median of the distribution), which developers considered, and performed anti-regressive work upon.

An increasing better performance of the MGM over releases was also observed when using the cyclomatic complexity: when considering the first half of releases, 60 percent of the overall anti-regressive work was correctly predicted. The second half of the evolution experienced 335 reductions in the cyclomatic complexity, while the model pointed 268 candidates, with a hit-ratio of 80 percent.

The other complexity factors (inbound and outbound coupling, lines of code) showed similar patterns, but generally worse medians: the prediction power of the MGM still has better performance in the most recent half of releases of the system’s evolution studied. During this second half:

- the hit-rate predicting the anti-regressive work of inbound couplings moves from 54 to 63 percent;
- for outbound couplings: from 63 to 76 percent;
- for the lines of code: from 63 to 77 percent.

As a general result, we can conclude that the developers of this project applied anti-regressive work for reducing the complexity of elements in the system reasonably well in line with the MGM model for major (large size) releases. (Anti-regressive work for small releases is difficult to predict.) The MGM model assumes that the more changes that a function underwent, together with a measure for its complexity, the more likely this function will be a candidate for anti-regressive work, and developers will reduce its complexity. Also, in the Arla case study, the cyclomatic complexity served in general as a better predictor for anti-regressive work. The model tends also to perform better when comparing the results of the first and the second halves of the evolution: this might be a result of developers accumulated experience that improves their ability to identify parts of the code in order to apply the anti-regressive work.

4.2. The other case studies

Table 2 presents a summary for the results of all the case studied. The highest hit rates are presented in bold. As visible, there is not a single complexity factor which alone makes a best predictor. This implies that for each system one needs to determine individually which measurement is best.

<table>
<thead>
<tr>
<th></th>
<th>c_out</th>
<th>c_in</th>
<th>McCabe</th>
<th>LOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arla</td>
<td>0.71</td>
<td>0.75</td>
<td>0.80</td>
<td>0.72</td>
</tr>
<tr>
<td>Gaim</td>
<td>0.73</td>
<td>0.76</td>
<td>0.77</td>
<td>0.71</td>
</tr>
<tr>
<td>Lcrzo</td>
<td>0.37</td>
<td><strong>0.86</strong></td>
<td>0.33</td>
<td>0.53</td>
</tr>
<tr>
<td>MPlayer</td>
<td>0.84</td>
<td><strong>0.88</strong></td>
<td>0.76</td>
<td>0.77</td>
</tr>
<tr>
<td>Netwib</td>
<td>0.25</td>
<td>0.50</td>
<td>0.44</td>
<td><strong>0.60</strong></td>
</tr>
<tr>
<td>rrdtool</td>
<td>0.93</td>
<td>0.38</td>
<td>0.84</td>
<td>1</td>
</tr>
<tr>
<td>xfce</td>
<td><strong>0.84</strong></td>
<td>0.78</td>
<td>0.80</td>
<td>0.76</td>
</tr>
<tr>
<td>XMMS</td>
<td>0.88</td>
<td>0.79</td>
<td>0.79</td>
<td>0.74</td>
</tr>
</tbody>
</table>

Table 4 in the Appendix (columns 5 and 6) presents the values of ‘total instances of anti-regressive work’ and ‘total correct predictions’ for the eight systems studied.

Overall, the case studies Lcrzo and Netwib showed the poorest results in the predictability of the MGM. The two systems represent the evolution and restructuring of the same application: Lcrzo is, in fact, the predecessor of the newly restructured Netwib. As in the Arla case study, if the set of releases was split into two, the earlier and the later halves, the MGM also gives a better predictability for these two systems during the later half, as seen in Table 3. This can be an indirect evidence of an increased ability of developers to focus their anti-regressive efforts as long as the system evolves: as experience accumulates it appears that developers are able to direct increasingly better their anti-regressive efforts towards sections of code that will benefit most from such work.
5. Related Work

The work presented in this paper is an extension and refinement of work started in [Capiluppi and Ramil 2004, Capiluppi et al 2005] where we looked at a single system and we found highly skewed distributions of accumulated change over releases and complexity, and high correlation between the two variables.

In general, during the last few years, it has been realized that empirical data for OSS systems is more widely available than for proprietary systems. A number of empirical studies of OSS have been published since initial research involving the Apache web-server and Mozilla browser [Mockus et al 2002]. More recent studies include those which examine single OSS projects [e.g. German 2003. Koch and Schneider 2000, Aoki et al 2001, Stamelos et al 2002, Godfrey and Tu 2000], and those which involve several systems [e.g. Capiluppi et al 2003, Capiluppi 2003].

Even though the vast majority of OSS software evolution studies are based on direct trend visualization and curve fitting, interesting new approaches to study the evolution phenomenon have been recently proposed through both quantitative [Antoniades et al 2003], and qualitative [Smith et al 2004] simulation methods. Demeyer et al [2005] presents a summary of recently proposed tools to support various parts of refactoring. These authors distinguish three type of tools: predictive (before refactoring), curative (during refactoring) and retrospective (after refactoring). Our model is predictive, because it permits the identification of candidates for anti-regressive work, before the actual work is done.

There are different types of such predictive tools: “bad-smells” detection tools are exemplified by Duploc, a tool which finds instances of duplicated code [Ducasse et al 1999] based on rules or heuristics. CodeCrawler is a tool to identify inappropriate uses of inheritance and classes which are excessively large [Lanza and Ducasse 2005]. Insider is a metrics-based tool which identifies various possible design flaws [Marinescu 2004]. Some of these tools provide support to detect code duplication. Our approach is unique in the way of trying to link process metrics (e.g. accumulated changed) with product metrics (e.g. complexity). Moreover, our approach can be applied sequentially, as the system evolves and every time a new release is generated our MGM model issues new sets of candidates for anti-regressive work.

Our work is also related to measurement-based research in determining maintainability and complexity indexes. Such research has been going on for many years. One example of this is the work of Oman and Hagemeister [1992], who created a maintainability index for an entire system which was intended to capture three different dimensions: the control and information structures and level of comments by means of a mathematical function (a polynomial). Later, Coleman et al [1994] applied this maintainability index to several software systems at Hewlett Packard. The first step involved the calibration of the maintainability index using several other metrics such an extended version of the cyclomatic complexity measure, the number of lines of code, the number of lines of comments, and the effort measure as defined in Halstead’s “Software Science” [Halstead 1977]. Coleman et al [1994] studied 714 third-party components (equivalent to 236,000 lines of code written in C). The polynomial index was used to rank the components according to the expected maintenance difficulty with the result corresponding to the intuitive or “gut feeling” appreciation by HP personnel. The maintenance index was reported to match the intuition of HP maintainers in subsequent studies. The model presented in this paper is more simple and, we believe, intuitive than other models proposed in the literature, and includes accumulated change as one parameter which is not present in other models.

6. Threats to validity

The following is a list of threats to validity affecting this research:

- Undetected changes: Our NRwC variable is based on changes that impact our four selected measurements (size in LOC, cyclomatic complexity, inbound and outbound couplings). In some cases there could be changes that do not impact on any of these and would pass undetected by our approach.
- Stability of the locus of high change: In these and other systems we have studied [Capiluppi et al 2005], the distribution of accumulated changes in software is in general asymmetrical (not uniform). However, the subset of elements that experience the highest change rate, the locus of high change, may not be stable in all systems, as assumed by our approach. In the current experiments with MGM this did not appear to affect the results, but if the locus (or loci) of high change is (are) changing over releases, we will need to experiment with other approaches to predict which are the elements of

<table>
<thead>
<tr>
<th>Lcrzo</th>
<th>c_out</th>
<th>c_in</th>
<th>Mc</th>
<th>Loc</th>
</tr>
</thead>
<tbody>
<tr>
<td>1st half</td>
<td>0.05</td>
<td>0.35</td>
<td>0.14</td>
<td>0.38</td>
</tr>
<tr>
<td>2nd half</td>
<td>0.5</td>
<td>1</td>
<td>0.75</td>
<td>0.8</td>
</tr>
<tr>
<td>Netwib</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1st half</td>
<td>0.13</td>
<td>0.5</td>
<td>0.18</td>
<td>0.25</td>
</tr>
<tr>
<td>2nd half</td>
<td>0.56</td>
<td>0.75</td>
<td>0.75</td>
<td>0.72</td>
</tr>
</tbody>
</table>

Table 3: increased predictability of the MGM when the set of releases is divided into two
the system which will be most likely to be subject to change.

- Code ownership: Code ownership may play a role in anti-regressive work decisions. For example, if a complex file is owned by a single developer who understands it well, the need for anti-regressive work is less than when the file ownership is transferred to another person, or the file is owned by a group of developers. This was not considered in our study.

- Release tree with multiple branches: Our approach to measuring change and refactoring work assumes a single and linear release sequence. The evolution of a system may have a complex release tree with many points of forking and merging of releases. For example, a function may have been modified during a “parallel” invisible release sequence “branch”. If this is the case, our NRwC metric would have underestimated the accumulated change and other change measurement methods (e.g. based on actual changed lines of code) would have been needed.

- Function renaming: During anti-regressive work some functions are renamed. In the present study we did not keep track of any instances of these.

- Actual use of the model by professional software developers: The ultimate evaluation of any model or tool for anti-regressive work guidance is when it is actually used by developers in order to guide their decisions. Moreover, we would like to measure actual productivity, maintainability and other similar attributes, so that it can be assessed that the used of the proposed anti-regressive model actually improves the evolution of the software. However, to achieve such level of validation is difficult. For example, developers may not be prepared to follow the recommendations of a model such MGM blindly. They may have other compelling “local” reasons to reduce the complexity of a different part of the code that the one recommended by the tool. One idea would be to run small classroom experiments, comparing the anti-regressive effectiveness of one group using the model and another group not using it. However, the results of these experiments (e.g. at a classroom level) may not scale up to real industrial development. Industrial scale experiments would be too expensive.

7. Conclusions

We have introduced here an approach for predicting which functions are best candidates for complexity control, that is, anti-regressive work. The focus of the study was specifically on OSS software, in order both to report on how OSS developers currently perform maintenance activities, and to provide the wider OSS communities with a means to better direct their anti-regressive work. The focus of the study were the functions (or methods) in the source code of a software system. As empirical metrics of the study, we selected the number of releases with changes (NRwC), and four measures of complexity (size, cyclomatic complexity, inbound and outbound couplings). We also argued that, for a source function, a decrease in the value of any of the complexity attributes was an instance of anti-regressive work on that particular function.

The developed approach included the proposal and testing of the Maintenance Guidance Model (MGM). The MGM ranks all the functions at release N and selects both the most complex and changed ones, and proposes them as the focus of the incoming anti-regressive work of release N+1.

To evaluate this model, data from eight OSS systems was used: the NRwC of functions and their four aspects of complexity were evaluated for all of the available releases of the selected projects. The candidates of MGM were tested against the actual observed anti-regressive work on functions performed between the Nth and the (N+1)th releases.

Two main observations can be drawn from the application of the MGM on these OSS projects:

1) the predictability of the MGM, tested against real data, had good performance in all of the systems. Using the median of the distribution of the predictions, at best a range from 60% to 100% of the MGM candidates coincide with those independently selected by OSS developers for anti-regressive work. On the contrary, and as visible in Table 2, we could not find a single complexity factor (among the 4 selected) which alone performed better than the others. Each system has one measurement which better captures the maintenance activities of developers.

2) The MGM predictability tends to increase as long as the systems evolve: dividing the life cycle of an application in two parts, we found that the model obtained a much better predictability in the second part, than in the first part of the life cycle. As mentioned above, an explanation could be an increase in the ability of developers to focus their anti-regressive efforts as the system evolves.

Further study of the results of the correlation analysis presented in section 2.2 should be done in order to find out whether these could be used to improve the MGM. In addition, we would like to investigate to what extent the inclusion of information about cumulative changes improves the MGM predictions. An interesting experiment would be to compare the MGM model, which is based on accumulated changes, with an alternative model that uses static characteristics of the code (the product) only. We also plan to replicate the approach for several
additional OSS systems to the ones presented here, and with other modeling approaches (e.g. machine learning).

Most of the selected projects were developed using C/C++, which do not have an IDE that supports well refactoring. The extension of this study to Java-based systems using, for example, the Eclipse IDE could lead to even more meaningful results, since refactoring is better supported in Eclipse. An interesting aspect to investigate empirically is the relationship between anti-regressive work and productivity, and in particular, whether those projects that apply anti-regressive work in a more systematic way (e.g. closer to the MGM predictions) display a higher productivity. A further aspect to investigate would be whether the distribution of cumulative change of software functions follows a Zipf-like distribution (equivalent to the Pareto distribution) [Feitelson et al 2006]. Such statistical regularities, if widely found and confirmed, may be used in future refinements of MGM.

One limitation to the replication of this study is the computational time needed to perform the coupling analysis. For larger systems (more than 1 millions LOCs), the evaluation of the couplings of functions takes several hours on a state-of-the-art PC when using an existing OSS package (Doxygen). We plan to introduce a better and faster coupling analysis algorithm in future research.

8. Acknowledgments

Many thanks are due to Dr Yijun Yu from The Open University for insightful suggestions on a previous version of this paper, both highlighting threats to validity that we were not aware of and for ideas of further work. We gratefully acknowledge the anonymous reviewers for their useful comments.

9. References


APPENDIX

<table>
<thead>
<tr>
<th>Domain</th>
<th>Total Number of Files</th>
<th>Total Number of Functions</th>
<th>Total Number of Lines of Code (LOC)</th>
<th>Skewness of distribution of accumulated change ('NRwC per function')</th>
<th>Total number of anti-regressive work instances since the second release available (observed)</th>
<th>Total anti-regressive work instances (correctly predicted)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arla</td>
<td>946</td>
<td>4,094</td>
<td>211,106</td>
<td>5.72</td>
<td>3,634</td>
<td>2,443</td>
</tr>
<tr>
<td>Gaim</td>
<td>819</td>
<td>5,411</td>
<td>235,710</td>
<td>5.23</td>
<td>7,875</td>
<td>5,370</td>
</tr>
<tr>
<td>Lcrzo</td>
<td>234</td>
<td>1,450</td>
<td>109,299</td>
<td>1.75</td>
<td>3,550</td>
<td>1,256</td>
</tr>
<tr>
<td>MPlayer</td>
<td>2,288</td>
<td>10,021</td>
<td>658,069</td>
<td>2.47</td>
<td>11,444</td>
<td>9,685</td>
</tr>
<tr>
<td>Netwib</td>
<td>1,057</td>
<td>4,296</td>
<td>282,916</td>
<td>1.43</td>
<td>1,536</td>
<td>483</td>
</tr>
<tr>
<td>rrds</td>
<td>59</td>
<td>350</td>
<td>27,000</td>
<td>2.27</td>
<td>187</td>
<td>129</td>
</tr>
<tr>
<td>xfce</td>
<td>802</td>
<td>4,719</td>
<td>540,043</td>
<td>2.99</td>
<td>4,648</td>
<td>2,940</td>
</tr>
<tr>
<td>xms</td>
<td>260</td>
<td>1,723</td>
<td>109,051</td>
<td>3.11</td>
<td>1,472</td>
<td>1,074</td>
</tr>
</tbody>
</table>

Table 4: Some characteristics of the selected projects (all measures are taken at the latest available release).

*This project experienced a complete restructuring in its 1.2x series of releases: from some 100KLOCs, and source 118 files, the project's size dropped to some 27 KLOCs and 59 source files.*