Sixth Workshop on Software Quality Analysis, Monitoring, Improvement, and Applications

SQAMIA 2017
Belgrade, Serbia, 11–13.09.2017

Proceedings

Department of mathematics and informatics
Faculty of Sciences
University of Novi Sad, Serbia
2017
Preface

This volume contains papers presented at the Sixth Workshop on Software Quality Analysis, Monitoring, Improvement, and Applications (SQAMIA 2017). SQAMIA 2017 was held during September 11 – 13, 2017., at the Hotel Prag, Belgrade, Serbia.

SQAMIA 2017 continued the tradition of successful SQAMIA workshops previously held in Novi Sad, Serbia (in 2012 and 2013), Lovran, Croatia (2014), Maribor, Slovenia (2015) and Budapest, Hungary (2016). The first SQAMIA workshop was organized within the 5th Balkan Conference in Informatics (BCI 2012). In 2013, SQAMIA became a standalone event intended to be an annual gathering of researchers and practitioners in the field of software quality.

The main objective of the SQAMIA series of workshops is to provide a forum for presentation, discussion and dissemination of the latest scientific achievements in the area of software quality, and to promote and improve interaction and collaboration among scientists and young researchers from the region and beyond. The workshop especially welcomes position papers, papers describing work in progress, tool demonstration papers, technical reports, and papers designed to provoke debate on present knowledge, open questions, and future research trends in software quality.

The SQAMIA 2017 workshop consisted of regular sessions with technical contributions reviewed and selected by an international program committee, as well as two invited talks by Nuno Antunes and Stéphane Ducasse. In total 19 papers were accepted and published in this proceedings volume. All published papers were at least double reviewed, and in some cases even quadruple reviewed. We are grateful to all PC members for submitting careful and timely opinions on the papers.

Our special thanks are also addressed to the members of the SQAMIA Initiative (sqamia.org) without who this and previous workshops could not be possible: Zoran Budimac (Novi Sad, Serbia), Tihana Galinac Grbac (Rijeka, Croatia), Marjan Heričko (Maribor, Slovenia), Zoltán Horváth (Budapest, Hungary) and Hannu Jaakkola (Pori, Finland).

We extend special thanks to the SQAMIA 2017 Organizing Committee from the Department of Mathematics and Informatics, Faculty of Sciences, University of Novi Sad, especially to the two co-chairs Nataša Sukur and Tijana Vslavski for their hard work and dedication to make this workshop the best it can be. Further we’d like to thank Doni Pracner for his patience and diligent work on making the proceedings.

This workshop is endorsed by COST action IC1402 Runtime verification beyond monitoring (ARVI).

And last, but not least, we thank all the participants of SQAMIA 2017 for their contributions that made all the work that went into SQAMIA 2017 worthwhile.

September 2017

Zoran Budimac
Workshop Organization

**General and Program Chair**

Zoran Budimac (University of Novi Sad, Serbia)

**Program Committee**

Nuno Antunes (University of Coimbra, Portugal)
Tihana Galinac Grbac (co-chair, University of Rijeka, Croatia)
Marjan Heričko (co-chair, University of Maribor, Slovenia)
Zoltán Horváth (co-chair, Eötvös Loránd University, Hungary)
Hannu Jaakkola (co-chair, Tampere University of Technology, Finland)
Harri Keto (Tampere University of Technology, Finland)
Vladimir Kurbalija (University of Novi Sad, Serbia)
Goran Mauša (University of Rijeka, Croatia)
Anastas Mishev (University of Sts. Cyril and Methodius, FYR Macedonia)
Zoltán Porkoláb (Eötvös Loránd University, Hungary)
Valentino Vranić (Slovak University of Technology in Bratislava, Slovakia)

**Additional Reviewers**

Michal Bystrický
Ivano Elia
Ján Lang
Rui Oliveira
Karol Rástočný

**Organizing Committee**

Nataša Sukur (co-chair, University of Novi Sad, Serbia)
Tijana Vislavski (co-chair, University of Novi Sad, Serbia)
Doni Pracner (University of Novi Sad, Serbia)
Ivan Pribela (University of Novi Sad, Serbia)

**Organizing Institution**

University of Novi Sad, Serbia
Faculty of Sciences
Department of Mathematics and Informatics

**Technical Editor**

Doni Pracner (University of Novi Sad, Serbia)
Table of Contents

- Approaches for Software Metrics Threshold Derivation: A Preliminary Review ............................ 1:1–1:08
  Tina Beranič, Marjan Heričko
- Two Dimensional Visualization of Software Metrics ....................................................... 2:1–2:06
  Tibor Brunner, Zoltán Porkoláb
- Tests of Graphics Rendering in Browsers .................................................................. 3:1–3:10
  Jaak Henno, Hannu Jaakkola, Jukka Mäkelä
- Improvement of Requirements Engineering Course – Medical Software Case Study ...................... 4:1–4:09
  Mirjana Ivanović, Amelia Badica, Maria Ganzha, Costin Badica, Marcin Paprzycki
- Technology and the Reincarnation Cycles of Software ..................................................... 5:1–5:10
  Hannu Jaakkola, Jaak Henno, Jukka Mäkelä
  David Kostanjevec, Maja Pušnik, Marjan Heričko, Boštjan Šumak, Gordana Rakić, Zoran Budimac
- Identifying Up-to-the-Minute Topics in Software Process Improvement Research ........................... 7:1–7:08
  Harri Keto, Hannu Jaakkola
- Blockchain Implementation Quality Challenges: A Literature Review ........................................... 8:1–8:08
  Bojana Koteska, Elena Karafiloski, Anustas Mishev
- Barriers and Facilitators of Digitalization in Organizations ................................................ 9:1–9:07
  Markku Kaüssisto
- Relationship Between Design and Defects for Software in Evolution ....................................... 10:1–10:10
  Matija Miletić, Monika Vukušić, Goran Mauša, Tihana Galinac Grbac
- A Practical Tutorial for FermaT and WSL Transformations ............................................. 11:1–11:08
  Doni Pracner, Zoran Budimac
- A Temporal File System for Student’s Assignments in The System Svetovid ............................. 12:1–12:08
  Ivan Pribela, Doni Pracner
- Case Study: Using Model Based Component Generator for Upgrade Projects ........................... 13:1–13:08
  Petar Rajkovic, Ivan Petkovic, Dragan Jankovic
- Containerized A/B Testing ............................................................................. 14:1–14:08
  Ádám Révész, Norbert Pataki
  Mário Santos, João Saraiva, Zoltán Porkoláb, Dániel Krupp
- Evaluating State Modeling Techniques in Alloy .................................................................... 16:1–16:09
  Allison Sullivan, Kaiyuan Wang, Sarfraz Kharshid, Darko Marinov
- Measuring Mangled Name Ambiguity in Large C / C++ Projects ......................................... 17:1–17:09
  Richard Salay, Zoltán Porkoláb, Dániel Krupp
- Comparison of Software Structures in Java and Erlang Programming Languages .................... 18:1–18:08
  Ana Vranković, Tihana Galinac Grbac, Melinda Tóth
- The Case for Non-Cohesive Packages ................................................................... 19:1–19:18
  Nicolas Anquetil, Muhammad Ussman Bhatti, Stéphane Ducasse, Andre Hora, Jannik Laval
Approaches for Software Metrics Threshold Derivation: A Preliminary Review

TINA BERANIČ and MARJAN HERIČKO, University of Maribor

Knowing the reliable software metrics threshold can contribute to product quality evaluation and, consequently, increase the usefulness of software metrics in practice. How to derive software metrics thresholds is a topic of many researchers, either proposing new approaches, or verifying existing methods on different practical projects. A literature review was conducted to summarise the available knowledge in this area. The aim was to find approaches for software metric threshold derivation and extract relevant information, such as programming language for which an approach was proposed and tested, what metrics were derived, and if approaches are automated with supporting tools. Data extracted from 14 primary studies reveal that the majority of studies present new approaches and approaches usually apply statistical analysis for threshold derivation. Primary programming language for which metrics are derived is Java, followed by C, C++ and C#. We also came across some tools used for threshold derivation, but in practise their use was not detected. Metrics derived, purpose of metrics derivation and other information are also summarised in the tables below.

Categories and Subject Descriptors: D.2.8 [Software Engineering] Metrics; D.2.9 [Software Engineering] Management-Software quality assurance (SQA)

General Terms: Software Metrics Threshold Derivation

Additional Key Words and Phrases: Software Metrics, Thresholds, Reference Values, Object-oriented, Literature Review

1. INTRODUCTION

To deliver software product in required quality, it is crucial to address and manage the quality assurance domain properly. Software metrics can be used for reflection of the qualitative characteristics of software modules in a quantitative way [Arar and Ayan 2016], presenting a control instrument in a software development and maintenance process [Alves et al. 2010]. Software metrics assess software from different views and, therefore, belong to different metrics groups, but overall, reflect the internal quality of software systems [Arar and Ayan 2016]. The usefulness of metrics without knowing their reference values is very limited, due mainly to interpretation difficulties [Ferreira et al. 2012]. To overcome the above-mentioned difficulties it is important that reliable reference values of software metrics are available.

Thresholds are heuristic values that are used to set ranges of desirable and undesirable metric values for measured software and, furthermore, used to identify anomalies which may be an actual problem [Lorenz and Kidd 1994]. Threshold values explain if a metric value is in the normal range [Ronchieri and Canaparo 2016] and, consequently, provide a scale for paying attention to threshold-exceeding components [Arar and Ayan 2016]. There are many approaches available to compute thresh-
old values. Anyway, as summarised by [Arar and Ayan 2016], threshold definition methodology should meet the following requirements: (1) It is built on representative benchmark data and not on opinions; (2) Data should be analysed statistically; (3) It is repeatable and clear [Alves et al. 2010; Sánchez-González et al. 2012].

The aim of the presented paper is to gather available knowledge regarding software metrics threshold derivation techniques. The first step in a literature review is to find already available related studies. [Ronchieri and Canaparo 2016] present a preliminary mapping study in the area of software metric thresholds in which they aim to present the identified state of current affairs. Two main research questions were formed, dealing with identification of the currently most important papers in software metrics threshold research community, and with the meaningful aggregation of those papers. During their review, they observed three main domain problems: (1) Unclear explanation of the method for selecting the technique that calculates threshold; (2) A direct application of the metric threshold values to different code context; (3) A lack of objective analysis of calculated thresholds [Ronchieri and Canaparo 2016]. Anyhow, the extracted domain specific data comprise the study topic and categorisation, number and type of analysed projects, types of derived metrics and programming language and computed threshold values. According to our knowledge, this is the only available review study dealing with the software metrics threshold values. This was also confirmed by [Ronchieri and Canaparo 2016].

The previously mentioned mapping study presented the starting point in our literature review. We addressed different research questions, so different extraction categories were defined. Since our search provided some additional articles that [Ronchieri and Canaparo 2016] did not examined, some extraction categories remain the same. Also, the search resources were expanded in our review.

The article is organised as follows. After the Introduction that presents an insight into the theoretical background and related work, Section 2 presents the performed literature review. Basic data extracted from primary studies are presented in this section. Section 3 explains and describes available approaches for software metrics threshold derivation, together with their connections and expansions. Extracted software metrics used in primary studies and details about derived threshold values are presented in Section 4. The last chapter concludes the article providing the discussion and future work opportunities.

2. A PRELIMINARY LITERATURE REVIEW OF THE RESEARCH AREA

Without knowing the threshold values of software metrics, their usefulness is limited. As stated by [Ferreira et al. 2012; Alves et al. 2010], threshold values for most metrics are still unknown. Nevertheless, numerous approaches for metrics derivation exist; there are many research opportunities that can be addressed aimed at improving domain knowledge. The performed literature review presents an insight into previous research in the area.

Based on our research purpose, four main research question were formed:

- **RQ1:** What approaches and methods are used for software metrics threshold derivation?
- **RQ2:** What is the purpose of threshold derivation?
- **RQ3:** For which software metrics thresholds are derived?
- **RQ4:** What tools are available for metrics' collection and threshold derivation?

Based on the research questions, the following search string was formed: ("software metrics" OR "source code metrics") AND ("threshold" OR "reference values"). With it, the search was conducted in selected digital libraries. Search results by data sources and final number of selected studies after applying inclusion and exclusion criteria are presented in Table I. Among the applied inclusion criteria
are that the study is available in selected digital libraries and is available in English, the study is from software engineering domain and that approaches for threshold derivation are presented or if they are just validated, the used metrics should be explained. Where available and appropriate, the search was limited to abstract, title and keywords. Since some search tools are looking across different digital libraries, some search results are listed in multiple categories.

In the end, 18 primary studies were selected for detailed analysis. Among them, four articles were excluded from the data extraction process. [Veado et al. 2016; Foucault et al. 2014; Sousa et al. 2017] are presenting developed tools for threshold derivation and [Vale and Figueiredo 2015] are presenting a tailored approach for threshold derivation for use in the context of software product lines. Finally, the data was extracted from 14 primary studies.

In Table II, basic information is presented about the selected primary studies. The first one is used programming language for which software metric thresholds were derived. As can be seen, the majority of studies derive threshold values for the Java programming language, while reference values for other programming languages, specifically C, C++ and C# are presented in 4 studies. Threshold values for other programming languages were not detected. The second information matter was the purpose of metric derivation. The majority of studies derive threshold to find or predict defects and errors in programmes, while some of them compute reference values for code smells detection purposes.

<table>
<thead>
<tr>
<th>Study</th>
<th>Programming Language</th>
<th>Purpose of metrics derivation</th>
</tr>
</thead>
<tbody>
<tr>
<td>[Alves et al. 2010]</td>
<td>Java, C#</td>
<td>Maintainability level</td>
</tr>
<tr>
<td>[Arar and Ayan 2016]</td>
<td>Java</td>
<td>Fault prediction</td>
</tr>
<tr>
<td>[Benlarbi et al. 2000]</td>
<td>C++</td>
<td>Fault prediction</td>
</tr>
<tr>
<td>[Boucher and Badri 2016]</td>
<td>Java</td>
<td>Fault prediction</td>
</tr>
<tr>
<td>[Ferreira et al. 2012]</td>
<td>Java</td>
<td>Detecting design flaws</td>
</tr>
<tr>
<td>[Fontana et al. 2015]</td>
<td>Java</td>
<td>Code smell detection</td>
</tr>
<tr>
<td>[Foucault et al. 2014]</td>
<td>Java</td>
<td>Risky classes</td>
</tr>
<tr>
<td>[Herbold et al. 2011]</td>
<td>Java, C#, C++, C</td>
<td>Evaluate quality attributes</td>
</tr>
<tr>
<td>[Hussain et al. 2016]</td>
<td>Java</td>
<td>Fault prediction</td>
</tr>
<tr>
<td>[Mihancea and Marinescu 2005]</td>
<td>Java, C++</td>
<td>Detecting design flaws</td>
</tr>
<tr>
<td>[Oliveira et al. 2014b]</td>
<td>Java</td>
<td>Internal quality</td>
</tr>
<tr>
<td>[Shatnawi 2010]</td>
<td>Java</td>
<td>Error risk level</td>
</tr>
<tr>
<td>[Shatnawi et al. 2010]</td>
<td>Java</td>
<td>Error risk level</td>
</tr>
</tbody>
</table>

The process of metric computation or threshold derivation is sometimes automated with tool support. As already mentioned, [Veado et al. 2016] present a TDTool for threshold derivation and [Oliveira et al. 2014a] present an RTTool for the same purpose. While TDTool supports any kind of metrics, RTTool compute threshold only for class level metrics that have heavy-tailed distribution. [Sousa et al. 2017] propose a FindSmells tool for smells detection in software systems based on software metrics and their thresholds. While linked to threshold computation, tools for metrics derivation was listed in some
articles. The tools that were used in the selected primary studies are Conecta [Ferreira et al. 2012], ckjm tool [Arar and Ayan 2016], Borland Together [Shatnawi 2010], Moose platform and VerveineJ [Oliveira et al. 2014b] and SIG Software Analysis Toolkit [Alves et al. 2010].

3. APPROACHES FOR THRESHOLD DERIVATION OF SOFTWARE METRICS

3.1 Novel Approaches

Different derivation approaches are proposed among the articles about metrics thresholds. According to [Fontana et al. 2015], approaches can be based on observation, statistical analysis, metrics analysis and machine learning. Detected approaches from primary studies are summarised and presented below in chronological order of publications.

[Erni and Lewerentz 1996] present a statistical method based on [DeMarco 1986] statement talking about judging metric values based on comparison to similar projects. They compute the average ($\mu$) and standard deviation ($\sigma$) of metric values of one software system, on the assumption that the metrics are distributed normally. The lower ($T_{min} = \mu - \sigma$) and the higher ($T_{max} = \mu + \sigma$) thresholds can be produced based on calculated values. Values that are outside the interval $\mu \pm \sigma$ are considered as outliers.

A method called Tuning Machine is presented by [Mihancea and Marinescu 2005]. The method is proposing a solution dealing with the problem of threshold values that should be used by a detection strategy. The new method for establishing proper threshold values is proposed with the help of a genetic algorithm.

Identifying the threshold values using Receiver Operating Characteristic (ROC) curves is presented by [Shatnawi et al. 2010]. They claim that meaningful and useful threshold values must be associated with design factors of interest. Therefore, their identification of metric thresholds is associated with class error probability, since their goal is to identify faults. ROC, a diagnostic accuracy test [Zweig and Campbell 1993], was used previously to make decisions in the diagnostic area in Radiology [Hanley and McNeil 1982] and clinical medicine [Zweig and Campbell 1993]. Since medicine represents a field in which metrics and their thresholds represent crucial information [Ferreira et al. 2012], many approaches now used in software engineering have roots in the medical domain. When calculating reference values of object-oriented metrics, several ways can be followed. However, [Shatnawi et al. 2010] are making the first attempt to use it for identifying object-oriented metric threshold values. They analyse three versions of Eclipse software, also using the error data from Bugzilla. The errors were then divided into three categories (Low, Medium, High) and statistical analyses were conducted to identify metric values that could classify Eclipse classes into different error categories (binary and ordinal).

Existing approaches that derive metrics threshold values systematically do not take into account the different statistical properties of software metrics [Alves et al. 2010]. For that reason, a method that derives metrics thresholds values based on empirical measurement is presented by [Alves et al. 2010]. It considers different data distribution and scales of source code metrics, and it is also resilient against outliers. For the presented method, which contains six steps, the benchmark of 100 object-oriented Java and C# systems were used for defining metrics reference values.

A statistical approach for threshold derivation with the help of a logistic regression model is presented by [Shatnawi 2010]. Logistic regression validates metrics and constructs a threshold model. The connection was checked between single metric and fault probability. For significant metrics thresholds, values were investigated with the use of the Bender method [Bender 1999], that has its source in the medical domain. Bender [Bender 1999] proposed two methods for finding possible thresholds: Value of an Acceptable Risk Level (VARL) and lower Value of an Acceptable Risk Gradient (VARG).
Approaches for Software Metrics Threshold Derivation: A Preliminary Review

Machine learning techniques were used in the approach by [Herbold et al. 2011]. They presented a programming language independent data driven method for threshold value calculation. Methodology can also be used for efficiency optimisation of existing metric sets and for the simplification of complex classification models.

[Ferreira et al. 2012] aimed at defining threshold values for object-oriented software metrics. Thresholds were derived by analysing the statistical properties of the data obtained by a large collection of different size and domain projects. After data were gathered, the values used most commonly in practice were identified. 40 projects were analysed and reference values for six metrics were calculated, taking into account the relevant data distribution.

Since it was shown that thresholds are connected to the context of a project, the approach by [Foucault et al. 2014] aims to overcome the mentioned drawback by presenting a method that can compute the threshold for any given context. They select software projects and entities randomly and follow the [Chidamber et al. 1998] principle that considers that a metric threshold can be derived from a quantile of the distribution of metric values. The input is quantile and estimation of corresponding thresholds is computed using a process that relies on the bootstrap statistical approach.

The Bender approach [Bender 1999] was used as a basis by [Hussain et al. 2016]. They proposed a model for metrics threshold derivation with use of non-linear functions, described with logistic regression coefficient.

3.2 Extensions and validations of Existing Approaches and Replicated Studies

An empirical method for extracting relative threshold for software metrics of real systems based on benchmark data is presented by [Oliveira et al. 2014b]. The absolute metrics threshold can be complemented and the definition of relative threshold introduced: \( p\% \) of the entities should have \( M \leq k \). \( M \) is the source code metric calculated and \( p \) is the percentage of entities that upper limit \( k \) should be applied to. Values of \( p \) and \( k \) are calculated.

[Fontana et al. 2015] also present a benchmark based data driven approach for deriving metrics reference values. Similar to [Oliveira et al. 2014b], the presented approach is inspired by [Alves et al. 2010]. The main goal is to design a method that can be applied on metrics that do not represent ratio values, typically ranging in the interval \([0,1]\). The method takes into account the statistical properties of obtained data of 74 open source projects.

Logistic regression was used by [Benlarbi et al. 2000], investigating the connection between faults and software metrics. Two models were constructed, one with threshold values and the other without. The models were then compared looking for connection.

[Arar and Ayan 2016] replicated the study by [Shatnawi 2010] and used a broader range of datasets and metrics, trying to resolve the problem regarding project dependent threshold values, so that other projects could benefit from the derived thresholds. Another replication study was published by [Yamashita et al. 2016] using the method by [Alves et al. 2010] is used.

The techniques based on the benchmark based method [Alves et al. 2010] and ROC curves [Shatnawi et al. 2010] were compared and investigated by [Boucher and Badri 2016]. Both methods were assessed as fault predictors and compared with the help of machine learning algorithms, which are often used as a fault-proneness prediction.

4. SOFTWARE METRIC THRESHOLDS

A number of different metrics were used in the selected primary studies and different threshold values were derived. Which software metrics in particular are presented Table III. Metrics are presented used either in the derivation or validation process. Most commonly, primary studies use the CK (Chidamber & Kemerer) object-oriented metrics [Chidamber and Kemerer 1994], consisting of six metrics: WMC
(Weighted Methods Per Class), DIT (Depth of Inheritance Tree), NOC (Number of Children), CBO (Coupling between Object Classes), RFC (Response For a Class), LCOM (Lack of Cohesion of Methods). Together with them, some other metrics are also used. Table III also gives information about the form in which the derived threshold values are presented. Usually, reference values are presented as a single numerical value, sometimes in the form of intervals. Rarely, primary studies present just approaches, and do not provide any concrete threshold value.

<table>
<thead>
<tr>
<th>Study</th>
<th>Metrics</th>
<th>Presented Thresholds</th>
</tr>
</thead>
<tbody>
<tr>
<td>[Arar and Ayan 2016]</td>
<td>WMC, CBO, RFC, LCOM, Ca, Ce, NPM, AMC, MAX, AVG, AVG, LOC, (e.g. LOC, MCCabe, fan-in, number of methods, number of parameters)</td>
<td>Yes (numeric) No (risk intervals)</td>
</tr>
<tr>
<td>[Alves et al. 2010]</td>
<td>SIG quality model metrics</td>
<td>Yes (numeric) No (risk intervals)</td>
</tr>
<tr>
<td>[Benlarbi et al. 2000]</td>
<td>CK metrics (without LCOM)</td>
<td>Yes (numeric) No (risk intervals)</td>
</tr>
<tr>
<td>[Boucher and Badri 2016]</td>
<td>CK metrics and SLOC</td>
<td>Yes (numeric) No (risk intervals)</td>
</tr>
<tr>
<td>[Ferreira et al. 2012]</td>
<td>LCOM, DIT, coupling factor, afferent couplings, number of public methods, number of public fields</td>
<td>Yes (interval good, regular, bad) Yes (interval low, medium, high)</td>
</tr>
<tr>
<td>[Fontana et al. 2015]</td>
<td>ATFD, WMC, NOAP + NOAM, WMC LOC, CYCLO, MAXNESTING, NOAV</td>
<td>Yes (interval low, medium, high) Yes (interval low, medium, high)</td>
</tr>
<tr>
<td>[Herbold et al. 2011]</td>
<td>VG, NBD, NFC, NST, WMC, CBO, RFC, NORM, NOM, LOC, NSM</td>
<td>Yes (numeric) No (risk intervals)</td>
</tr>
<tr>
<td>[Hussain et al. 2016]</td>
<td>DIT, CA, LCOM, NPM</td>
<td>Yes (numeric) No (risk intervals)</td>
</tr>
<tr>
<td>[Mihancea and Marinescu 2005]</td>
<td>ATFD, WMC, TCC, WOC, NOPA, NOAM</td>
<td>No (risk intervals) Yes (numeric)</td>
</tr>
<tr>
<td>[Oliveira et al. 2014b]</td>
<td>NOM, LOC, FAN-OUT, RFC, WMC, LCOM, PUBA/NOA</td>
<td>Yes (numeric)</td>
</tr>
<tr>
<td>[Shatnawi 2010]</td>
<td>CK metrics</td>
<td>Yes (numeric) No (risk intervals)</td>
</tr>
<tr>
<td>[Shatnawi et al. 2010]</td>
<td>CK metrics</td>
<td>Yes (numeric) No (risk intervals)</td>
</tr>
<tr>
<td>[Yamashita et al. 2016]</td>
<td>LOC, number of methods, fan-in, McCabe</td>
<td>Yes (interval) No (risk intervals)</td>
</tr>
</tbody>
</table>

Another point of interest was also the area of software metrics’ correlation and combination. The correlation of metrics, meaning combining semantically related metrics, were not detected. On the other hand, the metric combination was presented only for purpose of composing rules for code smell detection [Mihancea and Marinescu 2005; Fontana et al. 2015].

5. CONCLUSION

As presented in literature, thresholds are crucial for software metrics’ interpretation and, consequently, for increasing their use in practice [Ferreira et al. 2012]. Many derivation approaches can be found in the literature, each having its own characteristics. Deriving software metric thresholds is an important research topic, which was confirmed with the performed literature review and selected primary studies. Findings of the review are summarised in Table I, II and III, presenting answers to the proposed research questions.

In the context of RQ1, we searched for available software metrics derivation approaches, which are presented in chapter 3. First, approaches that were proposed for the first time are presented, like [Alves et al. 2010; Erni and Lewerentz 1996; Ferreira et al. 2012; Shatnawi 2010], after them, if available, the validation or extension of those approaches can be seen. The majority of primary studies derive thresholds for error finding purposes, some of them also for code smell detection, regarding
the answer on RQ2. With RQ3, we were looking for software metrics for which thresholds were derived. As can be seen in Table III, authors often use CK object-oriented metrics, adding other software metrics. The last research question, RQ4, was aimed at identifying tools, either for metric calculation or threshold derivation. Among primary studies, tools are proposed like TDTool [Veado et al. 2016], RTTool [Oliveira et al. 2014a] and FindSmells tool [Sousa et al. 2017] are proposed.

According to our findings, possible research directions can be highlighted. The presented approaches for software metric derivation that are, nowadays, usually used to find and predict errors, defects or deficiencies in software modules, can be used in the context of threshold derivation for code smell detection. Also, derived thresholds aimed at code smell detection can be compared to ones derived for fault prediction. Since different programme languages are not represented widely among primary studies, thresholds can be derived also for other languages and compared to already derived thresholds for Java. The use and comparisons of presented threshold derivation tools can be done, assessing and comparing their derived threshold values. Finally, tools can also be used in the context of code smell detection.

Acknowledgements

The authors acknowledge the financial support from the Slovenian Research Agency under The Young Researchers Programme (SICRIS/SRA code 35512, RO 0796, Programme P2-0057).

REFERENCES


Two Dimensional Visualization of Software Metrics

TIBOR BRUNNER and ZOLTÁN PORKOLÁB, Ericsson Ltd.

Successful software systems are under continuous change. Bug-fixing, adding new features modify or extend additional code to the existing code base. Under these activities we must be aware of software quality, avoid its degradation and alert when a major code refactoring seems to be inevitable. For this purpose we must continuously collect quality related data from the software: static analysis results, software metrics and other statistics. However, data have to be analyzed and presented in a way that the architects and designers could comprehend the information in various context: e.g. related to the number of changes on the code, with the relative distribution of the issues and in connection with the complexity of the given module. In this paper we show how to collect some of these key quality indicators and how to present them in a clear manner so architects and developers can overview the quality factors organized by nested components of the program in a multidimensional way. This method allows programmers to reason about correctness of the architecture, identify critical components and decide about necessary actions, like refactoring the architecture.

Categories and Subject Descriptors: D.2.8 [Metrics]: Complexity measures—Visualization; H.1.2 [Models and Principles]: User/Machine Systems—Human Information Processing

General Terms: Software metrics, Human Factors

Additional Key Words and Phrases: Software complexity, Two dimensional, Visualization

1. INTRODUCTION

Static program analysis is a methodology which means the observation of a software without its execution. Static analysis techniques target testing, correctness checking, comprehending and other purposes [Crawford et al. 1985; Gyimothy et al. 2005]. CodeCompass [Ericsson 2017] is a static analysis framework which gives opportunity to perform various kind of analysis on the source code and present the results in different visualization methods. The architecture of this framework consists of two orthogonal layers. On a vertical layer it has a classical server-client architecture, since the statically collected information is presented by a web server towards a GUI or other querying client scripts. And on the horizontal layer the server and the client are implemented as independent plug-ins.

Plug-ins provide the different functionalities of the framework. These can examine the code base from a wide range of aspects: some of them are language parsers which collect the named entities of a given language (variables, functions, classes, etc.) for further processing, others are inspecting the version control history of the project, yet others provide additional data from external sources, like third party databases.

In this paper first in Section 2 we overview the main features of CodeCompass, an open source code comprehension framework. We discuss the most important software metrics we measure in Section 3.
To collect measurement data we use plug-ins for CodeCompass. We introduce them in Section 4. Our paper concludes in Section 5.

2. CODECOMPASS

Bugfixing or new feature development requires a confident understanding of all details and consequences of the planned changes. For long existing stems, where the code base have been developed and maintained for decades by fluctuating teams, original intentions are lost, the documentation is untrustworthy or missing, the only reliable information is the code itself. Code comprehension of such large software systems is an essential, but usually very challenging task. As the method of comprehension is fundamentally different from writing new code, development tools are not performing well.

During the years, different programs have been developed with various complexity and feature set for code comprehension but none of them fulfilled all requirements. CodeCompass [Ericsson 2017] is an open source framework developed by Ericsson Ltd. and the Eötvös Loránd University, Budapest to help understanding large legacy software systems. Based on the LLVM/Clang compiler infrastructure, CodeCompass gives exact information on complex C/C++ language elements like overloading, inheritance, the (read or write) usage of variables, possible calls on function pointers and the virtual functions – features that various existing tools support only partially. The wide range of interactive visualizations extends further than the usual class and function call diagrams; architectural, component and interface diagrams are among the few of the implemented graphs.

To make comprehension more extensive, CodeCompass is not restricted to the source code. It also utilizes build information to explore the system architecture as well as version control information when available: git commit history and blame view are also visualized. Clang Static Analyzer results are also integrated to CodeCompass. Although the tool focuses mainly on C and C++, it also supports Java and Python languages.

A plug-in can introduce a model schema. CodeCompass establishes a connection to a relational database system and provides an Object Relational Mapping (ORM) tool to handle the persistence of ordinary C++ objects. Besides relational databases the given plug-in can also store its data in an arbitrary alternative database system or even in a single text file. The plug-in can provide a parser which fills this database. Currently CodeCompass contains many different type of parsers. Some of them parse the source code of different programming languages, others gather source control information from the Git database, or do text-search indexing. In this paper we will discuss the Metrics plug-in which provides metrics-related data in details in Section 3. The dataset collected by the parser is transferred to the client by a web server. This server listens on a port and routes the client queries based on the given URL to the single plug-ins. There are well defined interfaces between the client and the server, or the plug-in itself can also define its own API, since as an independent module of the framework only the plug-in has information about the structure of the stored dataset. The API can be accessed by a client program through a Remote Procedure Call (RPC) methodology. CodeCompass provides a web based GUI which enables different visualizations of the provided information. The GUI is also expandable by the plug-ins using JavaScript modules.

Having a web-based, plug-inable, extensible architecture, the CodeCompass framework can be an open platform to further code comprehension, static analysis and software metrics efforts. CodeCompass also a good starting point to develop a Language Server Protocol Clang Daemon [Microsoft 2017] prototype.

3. METRICS

Metrics are some quantitative measurements of a software. These are aiming to describe a project based on a given perspective [Fenton 1991].
3.1 Lines of Code

Maybe the most important metric is the Lines of Code. This is a simple way of describing the size of a project. There are subversions of this metric which give a more subtle picture: we can count the number lines which contain only comments, we can omit blank lines and we can compute pure source lines separately. This metric is independent from the programming language it is applied on.

3.2 Cyclomatic complexity

A more sophisticated way of measuring complexity of the software is Cyclomatic or McCabe metric [McCabe 1976]. This measures the linearly independent paths in a control flow. This metric can be easily applied on procedural languages, since it is determined by the number of decisions in the program. Note that this is also the number of parts the planar graph of the control-flow diagram divides the surface. Unfortunately this metric does not make distinction between the nested and sequential branches, though intuitively nested loops or conditionals are considered more complex than sequential ones.

3.3 Tight coupling

There are metrics which reflect on the architecture of the program on a higher level. Different types of Coupling can be defined which indicate how much the modules are independent from each other [Chidamber and Kemerer 1994; Henderson-Sellers 1996]. E.g. data coupling is tight when much information is shared between the modules via procedure parameters or global variables, or control coupling is tight when the module is controlling the flow of another by instructing it what to do, etc.

3.4 Runtime metrics

Not only static time metrics can be defined but also runtime ones which can be collected during program execution. These are like execution time and load time measurements or coverage metrics which show the ratio of the source code covered by a test suite and the amount of uncovered parts.

3.5 Number of bugs metrics

CodeCompass supports the compile time analysis of the project. This way the before-mentioned metrics can be collected. But it can also invoke external tools to gather quantitative descriptors of modules. LLVM/Clang is a compiler infrastructure which parses source codes in C++ language, builds its Abstract Syntax Tree (AST) which contains semantic information too. This enables programmers to run several checks on the program. Static Analyzer and Clang Tidy are two tools inside Clang which discover typical programming issues and misuses of the language. Some of these checks are simple enough so only consideration of the AST is sufficient: using namespace is strongly contraindicated in a header file. Its existence can be easily checked by inspecting the AST. Some others need path sensitive examination of the source: we can find division by zero or null pointer dereferences by following the control flow and note the possible values of the divisor or the pointer respectively. The technique used here is called symbolic execution which means the interpretation of the source code. Note that this is still a static analysis technique despite it has information about the symbolic values of variables and more complex expressions.

CodeChecker aims to collect the bug findings of these Clang tools and to store them in a database. The bugs can be queried through a public API. This way CodeCompass can access the number of errors committed in the source code. This gives a metric which describes the quality of a module.

4. METRICS PLUG-IN IN CODECOMPASS

In this section we describe the Metrics plug-in of CodeCompass from the database layout and the parser through the service to the GUI.
4.1 Database layout

Originally CodeCompass was developed as a code comprehension tool for large code bases. Scalability was always an important aspect since large-scale projects are not rare in industrial environment. This is the reason why not the whole AST is stored in the database. It turned out that for most tasks in code comprehension storing the named entities (classes, functions, variables, etc.) is sufficient. These are enough for navigating through the source code and inspect its regions.

The metrics differ from this in the sense that not only named language elements may possess metrics but files too. For example we can take the Lines of Code metrics for functions, classes and for files as well. In CodeCompass we chose a visualization method which works on the granularity of directories and source files. Thus in the database the following fields take place:

- **id.** Unique identifier of the metric.
- **file.** Identifier of the file to which the metric belongs.
- **metric.** Unsigned value of the metric.
- **type.** The type of the metric, like McCabe, Lines of Code, Number of Bugs, etc.

4.2 Parser

Usually it is the parser's job to collect all the information which is presented by the web service. However practically sometimes the service has enough information to answer some queries by computing the result on-the-fly. We decided that Lines of Code and McCabe metrics are computed while parsing the project. One reason is that the amount of information to store is proportional to the number of files which requires much less space compared to the other tables of the database (e.g. the AST node descriptor tables). The other reason is that some metrics are dependent on the programming language. Although it would be a good approximation of McCabe metrics to count the number of if, while, for, etc. keywords in a source file which introduce a control structure, but the answer would not be precise in case of comments or string literals containing these keywords. It is more convenient to use a language parser to build the AST and to count the control structures in it. However the disadvantage of this solution is that technically we loose the language independence, since building the AST requires a language parser added as a plug-in in CodeCompass.

4.3 Service

The main role of service layer is to serve the client. The question is that how much logic should the service perform; most of the times simply reading the database is sufficient but sometimes it is more convenient to accomplish a quick computation of which the result is not worth to store at parsing time. Another case when it is advantageous to entrust the service by doing the job is when the information is stored in a separate database. This is the case at CodeChecker where the checker results, i.e. the specific bugs of the source code are stored in an external database which can be accessed via another web interface. In CodeCompass we give the opportunity to visualize as many kind of metrics as many can be collected either in parsing time or in the service. Thus the service of Metrics plug-in provides one API function for gathering a specific metric for a given file: `int getMetrics(FileId, Metric type)`.

4.4 Graphical User Interface

CodeCompass provides a web-based user interface which can be opened in a browser. This enables us to create any spectacular visualization using the modern JavaScript frameworks and libraries. This also helps to achieve the largest user base, since no thick client application is required for browsing analysis results. In the literature we can find a big variety of visualization methods [Langelier et al. 2005]. Many times these methods are aiming to represent several dimensions in one picture which
Two Dimensional Visualization of Software Metrics

requires independent views. For this purpose the 3D space can be used [Wettel and Lanza 2008], or the texture of the diagram elements [Holten et al. 2005]. A common visualization for software metrics is the Treemap which means separate regions filling a surface and indicating the quantity by the area of a region [Balzer et al. 2005]. As for the metrics’ visualization we chose such a two-dimensional Treemap representation. In this view the source code hierarchy is visualized as rectangular boxes. Each box belongs to a directory or a file in the current folder. Those which belong to a directory can be clicked which event triggers a zooming animation which leads into the content of that directory. This view is two-dimensional, because the different metrics can be assigned either to the size of a box or to its color. For instance it can be set that the more lines of code is contained by a file, the bigger its size is. Or the more bugs are found in a specific directory, the bluer its color is.

In the picture below we can see an example of metrics’ usage. In this example the CodeCompass source code itself is parsed and analyzed. The used metrics are the lines of code in size dimension and the number of bugs in color dimension. In the first image we can see the content of the service directory. The largest subdirectory with LOC metrics is core-api since it has the biggest rectangle in the top left corner. The second largest is language-api, etc. In codechecker-api folder we can see a blue rectangle which indicates that it has a subdirectory with at least one found bug. By clicking on this rectangle we get the second picture which navigates towards include directory where the actual file (codeCheckerDBAccess.h) with a found bug can be seen.

5. CONCLUSION

In this paper we discussed a new approach to collect and visualize metrics data for large scale software systems. Our solution is implemented as a plug-in of the CodeCompass open source software comprehension platform. The parser plug-in collects the required data into a database, and the CodeCompass
server provides data for the clients as a service. The web-based graphical user interface is aimed to show various two-dimensional visualizations to reveal connections between the various components of the software system.

REFERENCES


Tests of Graphics Rendering in Browsers

JAAK HENNO  Tallinn University of Technology
HANNU JAAKKOLA, Tampere University of Technology, Pori Department
JUKKA MÄKELÄ, University of Lapland, Rovaniemi, Finland

Web browsers have become one of the most commonly used software and are important communication tool to access our data-driven, increasingly visual Internet. Browser graphics speed is essential for many commercial web applications – e-commerce sites, web portals, content management systems (CMS’s), therefore web developers should well understand their possibilities. Browsers can be seen as multi-input (HTML-text, images, CSS, Scripts) multi-output (code for processor, graphics card, sound system) translators, but little is known about their ‘internal life’, especially how they render graphics. Browsers interpreting HTML5 documents have two graphic modes: Retained Mode (images defined in HTML text and animated with CSS) and Immediate Mode (images created on canvas and animated with JavaScript). In order to understand differences of these modes in animation rendering speed were created nine different versions of an animation of Lunar Eclipse which were tested in six major PC and Android mobile phone browsers. Results indicate, that there are no significant differences in major browsers except that IE and Edge (still) lag behind in implementing novel graphics/video formats and that in all tested browsers Retained Mode is at least two times quicker than Immediate Mode.

Categories and Subject Descriptors: I.3.6 [Computer graphics]: Methodology and Techniques; I.3.4 [Computer graphics] Graphics Utilities
General Terms: Browser, Visualization, Layout Engine, Rendering Engine, Retained mode, Immediate Mode
Additional Key Words and Phrases: HTML5, CSS3, JavaScript, SVG, rendering

1 INTRODUCTION

The first program what (usually) everyone opens when starting a computer, tablet or internet-enabled phone is a browser. Browsers have become the most common communication/interaction tool for all Internet-connected world. Currently (spring 2017) there are available (for PC-s) more than 100 different browsers [Web Developers Notes 2017] with different features – built-in e-mail, add-blocking, torrent handling, streaming etc. About quarter of these are already discontinued (still available), but every year appear some new ones, e.g. only in 2016 appeared Brave (automatically blocks ads and trackers, handles torrents) [Brave 2016], Microsoft Edge (only for Windows 10), [Microsoft Edge 2016], Vivaldi [Vivaldi] and several others.

Although in general statistics usage of some of them may seem negligible, in some areas of the World and for some users they may be very important [Wikipedia 2016. Usage share of web browsers], therefore developers of commercial web applications should try to comply with most of them.

Web is first of all visual media and animation, movement is the major way to make WWW attractive. Modern browsers enable several technologies for creating animations, but little is known about their efficiency, especially speed and for a commercial website even one second delay in page opening could cost millions in lost sales over a year. In the following are analyzed browser’s graphic possibilities and created series of tests to compare the speed of different formats for creating browser animations.
2 BROWSER AS TRANSLATOR

Browser has become the major mechanism for presenting digital content on screen, i.e. a graphical User interface (UI). HTML/CSS is currently the most advanced and flexible UI framework. Browsers are often already considered as the basis of a whole Operating System (OS) [Prakash 2016].

Browser works like a programming language’s translator – it transforms information presented in format of input channels into format required by processor, graphics processor and other output devices, but its task is far more complex. Browsers accept input from several channels in several different formats – HTML/XML, CSS (Cascading Style Sheets), JavaScript, SVG (Scalable Vector Graphics), sound tags, keyboard and mouse or/and touch input; input may be provided from several files – the HTML-document itself and data from linked external sources, e.g. external CSS and JavaScript files, images, sound/video files. From these different types of inputs browser compiles complex output for computer/mobile screen containing text, images, video and sound.

![Browser as compiler with multiple input and output channels.](image)

A browser output is essentially a frame-based interactive video. Computer/tablet/mobile redraw the screen with constant frequency (currently usually 60 Hz), thus browser should constantly calculate from inputs (HTML, CSS, JavaScript, inputs received from user and network) next frame content - text/images on screen and accompanying audio. And browser should also manage device (PC, mobile) resources - developers want to use maximal color depth (bpp – Bits Per Pixel, determines color quality) and framerate (fps – Frames Per Second).

For classical (programming languages) translators we have a nice 'translator theory' and (more or less) established functional structure [Aho Ullman 1972]:

```
SCANNER → SYNTAX ANALYZER → ABSTRACT TREE GENERATION → CODE GENERATION
```

There does not yet exist comparable 'browser theory' and established functional structure for browsers. The complex functional scheme of browsers could be presented as in the scheme in Figure 2.

Browsers are developed by different organizations and use different subsystems and the 'inner workings' even of major browsers are mostly mystery. For open source browsers something is presented in [WebKit 2017], [Tali Garsiel, Paul Irish 2011], [Spyros Doulgeridis 2017] but in rather general terms. Even the term 'open source' has become nearly meaningless, knowledge that 16,231,043 lines of C++ code of the open source Chromium browser [Chromium (Google Chrome) 2017] are available does not help web developers; the Google's Chrome browser uses the Chromium codebase, but adds lot of its own, proprietary code (licensed codecs for proprietary media formats AAC, H.264, MP3, Google Update, Crash and Error Reporting etc.) [Chris Hoffman 2014], thus the codebase is essentially larger. The situation with other browsers is similar, e.g. codebase of the major open-source browser Firefox is freely downloadable [Firefox 54.0.1 source code], but what could a web developer do with > 1 GB, ca 85000000 lines of c/c* code? For a Web developer this is useless, web
developers are not c/c* experts, they are interested what this code can/will do and suggesting to search answers from code is similar to trying to guess human's behavior investigating his/her genome sequence.

Fig. 2. Possible functional scheme of a browser

All browsers have at least three major subsystems: XML parser (with refinements for HTML/XHTML, SVG, CSS), the JavaScript interpreter and the Layout/Rendering Engine [Grosskurth Godfrey 2005]. But all major browsers use their own, independently developed HTML/CSS/JavaScript parsers, layout and rendering engines.

Table I. Layout and JavaScript engines in some major browsers

<table>
<thead>
<tr>
<th>BROWSER</th>
<th>Layout Engine</th>
<th>JavaScript interpreter</th>
</tr>
</thead>
<tbody>
<tr>
<td>FIREFOX</td>
<td>Gecko</td>
<td>SpiderMonkey</td>
</tr>
<tr>
<td>CHROME</td>
<td>Blink (developed from WebKit)</td>
<td>V8</td>
</tr>
<tr>
<td>INTERNET EXPLORER</td>
<td>Trident</td>
<td>Chakra</td>
</tr>
<tr>
<td>MICROSOFT EDGE</td>
<td>EdgeHTML</td>
<td>Chakra</td>
</tr>
<tr>
<td>OPERA</td>
<td>WebKit</td>
<td>V8</td>
</tr>
<tr>
<td>SAFARI</td>
<td>WebKit</td>
<td>JavaScriptCore</td>
</tr>
</tbody>
</table>

This variety creates many questions. Performance of some components may be rather different [Jen Looper 2015]. How this influences browser's overall performance, in what order are applied CSS rules, weather determining attributes of elements of HTML document is better from HTML text or from JavaScript, levelling browsers build-in defaults (e.g. different built-in margins) – these and many other practical important problems for web developers are mostly unexplained in documentation.

The main factor affecting webpage opening speed is network speed. While this is the most important factor, page designer possibilities to change network configuration and/or servers are limited.

But there is also another important factor affecting page's opening speed – its graphics. HTTP Archive shows that in average images/animations/video make up 64% of a Web page's total size [Shull 2015]. Suitable organization of page graphical content, using best formats for animations, most effective technologies for images and video attributes (e.g. transparency) is of utmost importance for achieving fluid page presentation, essential for commercial web applications and games.
2.1 Graphic modes of browsers

With HTML5 browsers use two graphic modes for processing and rendering images: the Retained Mode and the Immediate Mode.

Retained mode was historically the first and is used e.g. with images presented by HTML `<img>` tag. This is for browser not a direct command 'start drawing', but a command 'show this image in page'. Where on page, over/under of which other page elements (sometimes also if/when) remain to for browser to decide.

Suppose browser finds in HTML text and CSS stylesheet lines:

```html
<img id=sun src=images/sun.gif>
#sun{
  position:fixed;
  left:0;
  top:0;
  z-index:1
}
```

The image is added to data structure (display list) for the all objects which should be displayed on page and layout engine decides, where and in which order (i.e. what appears on top of what) should be displayed on screen. Besides HTML `<img>` tag this is the display mode also for CSS background images, thus the most often used mode for displaying graphics in browser; for displaying are used rather complicated data structures (the display list) and layout algorithms.

The immediate mode was introduced in HTML5 with new element `canvas`. Canvas is an area on screen (html5 document window) where JavaScript commands draw directly; browser does not change anything, but sends the whole canvas as it is drawn by JavaScript to screen. For drawing on canvas can be used several interpreters of graphic commands, e.g. `CanvasRenderingContext2D` (2D graphics) allows to draw 2D graphical objects – lines, rectangles, circles, text, but there are also 3D contexts interpreting WebGL 3D commands. The major difference is that here programmer has himself to decide where to place the object and when to draw it. Since all digital screens are constantly redrawn, this enables animations.

To continue analogy with translators - the retained mode can be compared with compilers (everything is first computed and then used repeatedly, in every screen redraw), the immediate mode – with interpreters (what is computed is also used/executed at once, on next screen redraw).

The Cascaded Stylesheets language (CSS) appeared as a simple tool to format text in HTML documents. In next versions CSS2 and CSS3 its functionality has been significantly extended to cover also manipulating images (image placement, rotation, transparency, pixel noise, animation) and sound. Thus e.g. the latest version CSS3 allows to create rotating Earth and Moon rotating around the Earth using only CSS3, i.e. in the retained mode of browser. The main drawback (until now) is that CSS does not have 'real' variables – variables, which could get values from DOM (Domain Object Model – the data structure keeping all page elements) attributes. The preprocessor variables and CSS custom properties are a closed system accessible only from CSS and were in our tests not used.

2.2 Browser’s graphic possibilities

HTML5 allows to implement animations using several formats and technologies:

- showing animated GIF images either as a part of HTML-document (i.e. with HTML-code) or drawing with JavaScript; this old image format contains series of frames for storing short animations but is restricted to 8 bpp color resolution (i.e. image can have max 256 colors) and animation speed cannot be controlled by browser, it is pre-set when creating the GIF animation file; but animation (image) is easy to scale (make smaller, making it bigger destroys quality); this is similar to showing video in WebM format – browser does not have control, but shows sequence of already rendered frames;
- animation on HTML5 canvas with JavaScript (i.e. using immediate mode): showing/moving images, possibly clipping them;
- procedural texture – merging texture images changing their opacity [Cloud Procedural texture]; here this was used to produce Sun’s lava texture from only one image;
- CSS3 can animate object's position, scale, rotation and opacity [Paul Lewis, Paul Irish 2014]; it also allows to create frame-based sprite sheet animation without using JavaScript;
- SVG; SVG elements can be manipulated like HTML elements using transform functions, but many commands and attributes do not work the same way on SVG elements as they do on HTML elements, JavaScript feature detection fails, the local coordinate system of an element works differently for HTML elements and SVG elements, the CSS properties of SVG elements have different names, e.g. instead of background-color should be used fill etc.

For web developers and browser game authors is essential to know, how selection of some presentation format influences the overall performance. Thus on the game developing course presented by the first author in Tallinn University of Technology we decided to create series of ‘real-world’ animation tests to compare various technologies/formats both in Retained Mode and in Immediate Mode.

Graphics rendering in browsers has been tested also earlier ([Roast 2013], [Vladić et al 2012], [MotionMark 2016]), but these tests had different aims, they did not compare influence of different graphic and animation design formats to animation rendering speed.

2.3 Concerns of Web Developers

One of the most important issues for web page designers is page opening speed. Research shows, that users form an opinion about web page visual appeal already in 50 milliseconds [Lingaard et all 2006].

For commercial web applications - web store fronts, product advertisements etc. this may be crucial. Google research shows, that 53% of mobile users abandon sites that take over 3 seconds to load [Doubleclick 2016]. Another recent study [Kissmetrics 2011] found that:

- 47% of consumers expect a web page to load in 2 seconds or less;
- 40% of people abandon a website that takes more than 3 seconds to load;
- one second delay in page response may result in a 7% reduction in conversions;
- if an e-commerce site is making $100,000 per day, one second page delay could potentially cost you $2.5 million in lost sales every year.

Especially intense is this problem in mobile, web and cloud programming. Here a programmer is has minimal possibilities for understanding inner workings of code and debugging, but the code should not only work, but should be memory-efficient and work quickly.

It has been claimed that Internet user's attention span is all the time diminishing. Although recently this claim become dubious [Maybin 2017], understanding factors which influence page opening speed is of utmost importance for web developers.

2.4 Artificial aids: Shims, polyfills, jQuery

Development of web languages – HTML(5), CSS, JavaScript should be based on standards established by World Wide Web Consortium (W3C) [W3C 2017.Standards]. But browsers and their subsystems – HTML and CSS parsers, JavaScript engine etc. are developed by different organizations, thus features what and how they implement are often different, e.g. for quite a long time Microsoft browsers (IE6..IE9) did not follow W3C standards, but tried to introduce their own functionality and syntax.

In order to make browsers to behave (more or less) the same way so that content on users screen would appear similar in whatever browser is used were introduced many JavaScript libraries under different names – pre-processors, frameworks, shims, polyfills etc. They introduced for browsers which originally did not have some functionality or syntax the missing functionality with JavaScript. Best known is the jQuery, which was introduced to make Microsoft browsers IE6..IE9 similar to standards-obeying ones, but by now contains lot of additional features - animation, physics, fade-ins and fade-outs (the 'visual sugar') etc.
With rapid updates of browsers JavaScript frameworks intended to enrich their functionality are quickly becoming obsolete. Besides, these 'functionality enrichment' frameworks introduce a nontransparent layer of JavaScript code (often minified, i.e. made difficult for humans to read), which makes JavaScript debugging very difficult. The original reason for jQuery has vanished (Microsoft browsers IE10 and Edge are already more or less following standards), but jQuery is still sometimes considered almost as a standard, therefore in one test jQuery was also used. The jQuery library was introduced to make Microsoft browsers IE6..IE9 to understand standards, but currently Microsoft has also started to follow them, so jQuery is (mostly) not needed. jQuery introduces rather cryptic, difficult to understand syntax (what has to be learned) and is changing custom semantics. For instance, a cryptic jQuery command to get canvas object:

```javascript
var $canvas = $('#canvas');
```
returns array (#canvas suggests, that there are several canvas objects having the same id?!); equivalent to this, but more understandable plain JavaScript command

```javascript
var $canvas = document.getElementById('canvas');
```
returns 'flat' variable, thus uses of these variables also require different syntax.

Use of jQuery also increased memory requirements (as measured in IE 11). It was relatively easy to remove all dependencies of jQuery - only 8 lines had to be changed to convert the test script into 'clean' JavaScript without use of jQuery.

3 TESTS

All tests implemented the same animation - Moon Eclipse (actually happened during the course on 20.03.2015), using different formats and technologies. The scene contains two animated objects: Sun with animated lava texture and rotating Earth. Since handling transparency has been for browsers problematic for a long time (see and compare e.g. discussions [CSS Transparency Settings for All Browsers 2007], [Cross Browser Opacity 2014]), in tests were used also three objects with different transparency/opacity – Moon's shadow on Earth surface, Earth's air - the white half-transparent halo around Earth – and Earth night side – gradient overlay changing from transparent (opacity = 1) to grey (opacity = 0.2). All test animations eclipse1.htm..eclipse9.htm were HTML5-documents looking similar on screen, but implemented using different technologies/formats in order to test their performance, i.e. time required to show fixed-length animation.

![Fig. 3. The test application and its animated objects: Sun (in upper left corner with animated lava texture), Earth (in lower right corner, rotating), Moon (small grey circle close to earth), Moon shadow (half-transparent grey circle on Earth surface), Earth atmosphere (light half-transparent halo surrounding Earth), night (right) side of Earth.](image-url)
3.1 Test files

In the following table are described test files T1..T9; they all are available online with summary of results [Henno 2016].

<table>
<thead>
<tr>
<th>Test files</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1: eclipse1.htm</td>
<td>The whole screen is covered with canvas having cosmos as the CSS-defined background image; Earth is animated with JavaScript (texture is constantly moved behind a clipping circle, drawn on canvas by JavaScript); all other objects are images in HTML document placed using CSS attribute z-order over the canvas; Sun is animated GIF (32 frames) with transparent background, Moon, Moon shadow, Earth atmosphere are images, 50% transparency of Moon shadow, Earth atmosphere and day-side of Earth is 'built-in' to images in .png format with Photoshop (is not adjusted in the HTML-document); placement of images is defined with CSS.</td>
</tr>
<tr>
<td>T2: eclipse2.htm</td>
<td>Sun and Earth are as in previous, Earth Air and day-night mask are DIV-s, their transparency and clipping are defined with CSS rules</td>
</tr>
<tr>
<td>T3: eclipse3.htm</td>
<td>The whole screen is a DIV with cosmos as the CSS-defined background image; Sun is animated GIF, minimal canvas is used only behind Earth, which is animated with JavaScript (texture is constantly moved behind a clipping circle drawn by JavaScript using the 2D-graphics context of canvas); all other objects are images in HTML document placed using CSS attribute z-order over the canvas, 50% transparency of Moon shadow, Earth atmosphere and day-night mask is defined in Photoshop</td>
</tr>
<tr>
<td>T4: eclipse4.htm</td>
<td>As previous, but 50% transparency of Moon shadow and Earth atmosphere images is defined with CSS rules</td>
</tr>
<tr>
<td>T5: eclipse5.htm</td>
<td>The whole screen is a series of DIV-s (no canvas); the main DIV with cosmos as the CSS-defined background image covers the whole screen, smaller DIV-s for Earth, Moon, Moon shadow and Earth atmosphere images are placed over it using the CSS position, width/height and z-order attributes; Sun is animated with CSS3 rules (the 32 frames of the sprite sheet were obtained from the animated GIF); Earth is also animated with CSS (texture is constantly moved behind a CSS clipping circle); transparency of Moon shadow and Earth atmosphere images is defined in Photoshop</td>
</tr>
<tr>
<td>T6: eclipse6.htm</td>
<td>The whole screen is a series of DIV-s (no canvas); the main DIV with cosmos as the CSS-defined background image covers the whole screen, smaller DIV-s for Earth, Moon, Moon shadow and Earth atmosphere images are placed over it using the CSS position, width/height and z-order attributes; Sun is video in WebM format (defined by &lt;source src=&quot;images/sun.webm&quot; type='video/webm; codecs=&quot;vp8, vorbis&quot;/&gt;, the video is clipped by CSS clipping circle (works correctly only in Firefox; Chrome has its own proprietary format for alpha transparency in WebM-video); Earth is animated with CSS (texture is constantly moved behind a CSS clipping circle); transparency of Moon shadow and Earth atmosphere images is created in Photoshop</td>
</tr>
<tr>
<td>T7: eclipse7.htm</td>
<td>Sun texture is procedurally generated by JavaScript and jQuery on minimal canvas using two additional canvases (the idea from [Professor Cloud 2013]); all other elements are as in previous example</td>
</tr>
<tr>
<td>T8: eclipse8.htm</td>
<td>As in previous but jQuery library (253 kb) was removed</td>
</tr>
<tr>
<td>T9: eclipse9.htm</td>
<td>Texture of Sun is procedurally generated (without jQuery), Earth is JavaScript animation on a separate canvas, thus together there are 4 canvases</td>
</tr>
</tbody>
</table>

3.2 Animation technologies

All tests used two objects animated with different technologies: Sun (bubbling lava texture) and rotating Earth.

For Sun was used frame-based animation. In tests T1..T6 browser gets pre-rendered frames (from animated GIF, WebM-video or spreadsheet; in tests T7..T9 Sun's texture is procedurally generated (using additional canvases).

Earth rotation was created dragging Earth texture from left to right, but showing only a part of it in clipping circle - screen's 'hole' to see through. In tests T1..T4 the clipping circle was created on canvas with JavaScript, in tests T5..T9 was used similar feature available in CSS3.
In tests were also changed ways to create half-transparent images - Moon shadow, Earth air and Earth day-night half-transparent overlay. We wanted to compare transparency/opacity adjustment in browser using JavaScript and CSS (i.e. browser has to calculate alpha mask) versus pre-set transparency in .png image. The Earth Air (in immediate mode) was a div with radial gradient on background; the Earth day-night overly was created by linear clipped by a SVG circle.

4 RESULTS

All test files had a small JavaScript script, which measured the time what was used to run 5 rotations of the Earth; the results were shown on screen and stored using the HTML5 local storage feature. For measuring memory performance there are not yet general standards. In Chrome is available a proprietary method performance.memory [Colt McAnlis 2013 ], but in our tests Chrome reported always the same values. IE allows to see some memory statistics with the UI Responsiveness tool [Microsoft. Improving UI Responsiveness xxiv]:

<table>
<thead>
<tr>
<th>Result</th>
<th>Paint time (msec)</th>
<th>Frame rate (FPS)</th>
<th>Memory (MB)</th>
<th>CPU (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.6</td>
<td>60</td>
<td>58.7</td>
<td>2%</td>
<td></td>
</tr>
</tbody>
</table>

Fig. 5. The window of the Microsoft UI-responsiveness tool

The values produced by this tool varied 4-10% and therefore are not presented here; the only more or less constant change was 3-4% increase in used memory when the jQuery library was used (the test eclipse8.htm).

In order to eliminate computer speed and network latency, all tests were done locally under a local Wamp server.

These tests produced lot of numbers. In order to compare influence of different formats results were normalized, using the eclipse1.htm in Firefox as the control case, i.e. in the following table the first number is time(Ti) - total time for test Ti with this browser and the second – percentage of this time of the 'etalon' time, i.e. calculated with formula time(Ti)*100/timeFF(T1), where timeFF(T1) is the time reported for test T1 by Firefox.

<table>
<thead>
<tr>
<th>Test</th>
<th>Browser</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>FF 51</td>
</tr>
<tr>
<td>66773ms 100%</td>
<td>68281ms 102%</td>
</tr>
<tr>
<td>T2</td>
<td>66744ms 99%</td>
</tr>
<tr>
<td>T3</td>
<td>66735ms 99%</td>
</tr>
<tr>
<td>T4</td>
<td>66720ms 99%</td>
</tr>
<tr>
<td>T5</td>
<td>20011ms 29%</td>
</tr>
<tr>
<td>T6</td>
<td>20023ms 29%</td>
</tr>
<tr>
<td>T7</td>
<td>20010ms 29%</td>
</tr>
<tr>
<td>T8</td>
<td>20010ms 29%</td>
</tr>
<tr>
<td>T9</td>
<td>66721ms 99%</td>
</tr>
</tbody>
</table>

These tests were performed also with a mobile phone browsers in an android mobile (LG E975a, Android 4.4.2 'KitKat'). In mobile were used the phone's OS built-in browser, Chrome and UC cloud browser (made in China), which currently is a 'rising star' in the landscape of mobile browsers [StatCounter 2017]. In the following table are presented the raw results (test times in milliseconds) and results after normalizing using the Chrome browser as an etalon.
Table IV. Results of tests in mobile browsers

<table>
<thead>
<tr>
<th>Test</th>
<th>Chrome</th>
<th>LG OS browser</th>
<th>UC browser</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>66972ms 100%</td>
<td>80060ms 119%</td>
<td>78215ms 117%</td>
</tr>
<tr>
<td>T2</td>
<td>66882ms 100%</td>
<td>83453ms 125%</td>
<td>98413ms 145%</td>
</tr>
<tr>
<td>T3</td>
<td>66973ms 100%</td>
<td>85029ms 124%</td>
<td>69619ms 127%</td>
</tr>
<tr>
<td>T4</td>
<td>67308ms 100%</td>
<td>82938ms 124%</td>
<td>85169ms 127%</td>
</tr>
<tr>
<td>T5</td>
<td>19884ms 30%</td>
<td>20060ms 30%</td>
<td>19109ms 29%</td>
</tr>
<tr>
<td></td>
<td></td>
<td>– no clipping</td>
<td>no clipping</td>
</tr>
<tr>
<td>T6</td>
<td>19835ms 30%</td>
<td>No WebM</td>
<td>19930ms 30%</td>
</tr>
<tr>
<td></td>
<td></td>
<td>– no clipping</td>
<td>no clipping</td>
</tr>
<tr>
<td>T7</td>
<td>19938ms 30%</td>
<td>– no clipping</td>
<td>19129ms 29%</td>
</tr>
<tr>
<td></td>
<td></td>
<td>– no clipping</td>
<td>no clipping</td>
</tr>
<tr>
<td>T8</td>
<td>19938ms 30%</td>
<td>– no clipping</td>
<td>19835ms 30%</td>
</tr>
<tr>
<td></td>
<td></td>
<td>– no clipping</td>
<td>no clipping</td>
</tr>
<tr>
<td>T9</td>
<td>68086ms 100%</td>
<td>– no clipping</td>
<td>71027ms 106%</td>
</tr>
</tbody>
</table>

5 CONCLUSIONS

The performed tests allow to draw several conclusions:

- there are no essential differences in speed between major browsers, but Microsoft browsers do not (yet) implement CSS3 (only CSS2);
- HTML5 immediate mode (canvas+JavaScript) allows to create complicated animations, but reduces animation speed ca three times (tests T5, T6, T7 did not use canvas); this result was a bit surprising, since canvas is commonly considered the main element in all graphics-intense web applications (games, portals etc.) but becomes understandable if one thinks what actually is loaded as the canvas 2D context – this is an interpreter of 2D graphic commands, which has to build its own name table etc.; returning to the analogue with translators: compiled code (i.e. retained mode) is quicker than interpreted (immediate mode);
- using several canvases does not make application slower (test T9) – they all use the same 2D context, i.e. graphics interpreter;
- creating transparent masks with CSS and changing their opacity (also opacity bitmaps – Moon shadow) with JavaScript does not make application slower and allows better control of result (tests T2, T4);
- CSS3 animations and CSS3 clipping (with circle) are quick, but difficult to scale (changing size of frame-based CSS animation is very error prone) and did not work in Microsoft browsers IE10 and Edge;
- video in WebM format with transparent background (test T6) can be currently achieved (using CSS3 clipping) only in Firefox (Chrome has for this a proprietary extension [Sam Dutton 2016]);
- the speed of canvas animation depends essentially on size of animated objects – scaling page down decreased rendering time (we did some separate scaling tests);
- results of tests T7, T8, T9 indicate, that jQuery was officious – big, especially for mobile applications (current version 3.1.1 – 261 kB) and did not have any advantages.

REFERENCES

https://www.HTML5rocks.com/en/tutorials/internals/howbrowserswork/  
Improvement of Requirements Engineering Course –
Medical Software Case Study

MIRJANA IVANOVIĆ, University of Novi Sad, Faculty of Sciences, Department of Mathematics and Informatics
AMELIA BADICA, Faculty of Economics and Business Administration, University of Craiova
MARIA GANZHA, Systems Research Institute Polish Academy of Sciences and Warsaw University of Technology
COSTIN BADICA, Department of Computers and Information technology, University of Craiova
MARCIN PAPRZYCKI, Systems Research Institute Polish Academy of Sciences and Warsaw Management Academy

Requirements engineering is one of crucial phases in developing any kind of software. For complex and demanding software it is necessary, before starting actual development, to elicit and specify requirements in form of appropriate document that will facilitate communication between developers and users. Requirement Engineering (RE) is, usually, one of master courses during ICT studies. Nowadays, society is confronted with a global aging population, and significant efforts are dedicated to the development of complex healthcare and medical software. In this paper we will discuss some important elements and characteristics of requirement document template and also critical and important lessons for collecting requirements for healthcare and medical software. We also propose some steps for improving RE course that has been delivering for more than 10 years at the University of Novi Sad.

Categories and Subject Descriptors: D.2.1 [Requirements/Specifications] ; D.2.9 [Software quality assurance-SQA]: — Management; J.3 [Medical information systems]: —Life and medical sciences; K.3.2 [Curriculum]: —Computer and Information Science Education

1. INTRODUCTION
Remarkable gains in life expectancy have influence on society, as they result in global aging population. Wide range of stakeholders invests serious efforts to develop different kinds of healthcare and medical software, in order to support, on the one hand, independent living of old population and, on the other hand, unhealthy and disabled population. Majority of medical devices, available today, could not function without appropriate software. So-called medical software is a special kind of software, used within medical context, which possess the following characteristics:

1) standalone software used for diagnostic/therapeutic aim,
2) "medical device software" – embedded in specific medical (smart) devices,
(3) software that drives medical devices or acts as an accessory to medical devices,
(4) software that supports the design, realization, and testing of medical devices; or provides quality control management for such devices.

Healthcare and medicine domains, nowadays, are facing a number of exclusive challenges in technology adoption. Very important aspect is strict connection to the regulatory issues, dealing with privacy and protection of patient data, thus seriously influencing applications and services. Contrary to some other domains of software development, healthcare and medicine domains absorb and adapts to new technologies more slowly. One of the main reasons is that the regulatory and operational circumstances don’t adequately support it. It means that healthcare and medicine, contrary to information-communication technologies (ICT), move at a different tempo, dealing with complex integration of technology into rather inflexible regulatory environments.

In many health jurisdictions, unexpectedly, medical software is usually specified as medical device software (MDS) [Wang et al. 2014]. MDS is predominantly used to analyze patient data and support a diagnosis, or monitor the patient’s health. Any drawback in MDS can seriously harm patient’s health. So it is important that legislators, and regulatory agencies, obtain specific regulatory standards (usually in form of guidelines) to try to ensure the safety, security and reliability of MDS. These standards heavily support necessity and importance of complete and consistent requirement specifications for medical device software.

There are several worldwide established and recognized institutions (like, in Europe - European Medical Devices Directive or, in USA, FDA - Food and Drug Administration) that deal with important regulations, and related international standards, relevant to the development of healthcare, but also medical (device), software. The particular importance has to be devoted to documenting software and system requirements, for development of healthcare and medical (devices) software. In this context, it is important to introduce such topics and adequate case studie(s) within appropriate Software or Requirement Engineering courses within ICT study programs. This is particularly important since, likely, certain number of students, in their future professional life, will be part of teams that develop healthcare and medical software, or some specific components embedded into medical devices. For example, concerning medical device software it has been stated that: Even slightly erroneous behavior by such a device could lead to a grave incident. The FDA Manufacturer and User Facility Device Experience (MAUDE) database contains a large number of reports on such incidents [FDA 2017].

On the other hand, the European Medical Device Directive MDD 93/42/EEC [1993] looks at the software as a specific type and part of medical, usually smart, devices. It even specified rules under which software can be treated as a medical device. The software influences the functioning of a medical device; or it is intended for the analysis of patient data and use for/by patients to diagnose them. Rapid development of ICT, and other interconnected disciplines, influenced appearance of a new buzz word digital health incorporating: mHealth (mobile health), telehealth and telemedicine, wearable (smart) devices, and personalized medicine. Many stakeholders [FDA 2017] are involved in digital health activities: patients, healthcare practitioners, medical device industry, and recently unavoidable mobile application developers (that develop smart mobile health devices) and of course medical experts. Note that, the FDA has been working in the digital health field, to balance benefits and risks, in order to provide clarity using practical approaches in: wireless medical devices, mobile medical applications, medical device data systems and interoperability, software as a medical device (SaMD) and so on [HHS 2017]. On the other hand plain mobile applications are also promising to help people manage their own health and wellness, and gain access to useful information. The FDA encourages the development of mobile medical applications that:

(1) help patients self-manage their condition and track their health information with simple tools,
(2) provide easy access to information and help patients communicate specific medical conditions to healthcare givers,

(3) enable patients or care givers to interact with Electronic Health Record (EHR) or Personal Health Records (PHR) systems,

but also to monitor the safety and effectiveness of medical devices.

In this paper, we present some considerations and suggestions on specific requirements for healthcare and medical (device) software. Furthermore, we analyze some specifics of the medical domain and how to handle requirements to develop secure and reliable software in this area. Moreover, we will discuss main objectives for requirements template for developing specific type of medical software. All these considerations will be put in the context of a Requirement Engineering (RE) course. Here, possible advantages of introducing specific medical (device) software requirements as case study within such course will be discussed.

In this context, the paper is organized as follows. Section 2, discusses some elements about healthcare and medical software like: Standards, Guidelines and Requirement Templates. Section 3, features a brief overview concerning general objectives for MDS requirements template, as specific type of software in the medical domain. Section 4, as a central one, brings guidelines and propositions for improvement of practical part of the RE course. A Medical Case Study in RE course, as well as specific requirements and possible advantages, are discussed. We conclude the paper with some general comments and observations.

2. STANDARDS, GUIDELINES AND REQUIREMENT TEMPLATES ON MEDICAL (DEVICE) SOFTWARE

Healthcare and medical software development teams, have to use best practices for rapid product development, due to new or constantly changing regulatory requirements. Some best practices must not be neglected like: Patients are primary; Short software development cycles; Don’t try to avoid risks; Continuously validate developed parts/components.

Most important industry innovation, in software technologies and development, have led key government regulators to recognize the crucial importance of standalone medical (device) software products and applications, as well as their incorporation in bigger software architectures and workflows. This has been reflected in significant regulatory changes in multiple E.U. MDD and U.S. FDA guidance documents [Vogel 2011]. The hierarchy of international standards and regulations that are relevant for the development of, first of all, medical (device) software, but also, more generally, healthcare and medical software, is presented in Figure 1.

Slightly different regulations are proclaimed by different Institutions but, generally speaking, they are concentrated on safety, security and reliability issues. U.S. regulation requires that medical devices go through premarket approval [Vogel 2011]. In Canada, to market medical devices, manufacturers must be authorized and approved by the Canadian Medical Devices Bureau for their quality, safety, and effectiveness. For example the Medical Device Directive (MDD) 93/42/EEC (amendment MDD 2007/47/EC) regulates the implementation of medical device software. Also there are some ISO/IEC standards that cover the quality aspects and development process for medical (device) software [Jetley et al. 2013; Rust et al. 2015].

Medical (device) software is, definitely, a specific and unique kind of software, and RE tasks must be considered more seriously and comprehensively than for traditional software development. Considering the format and content of a requirements template (as important and necessary part of requirements specification), there exist multiple possibilities and different published requirements templates [Committee and Board 1998; Robertson and Robertson 2012; El Gamal and Kriedte 1996; Lempia and Miller 2009; Alspaugh et al. 1992; Ahmadi 2006; Lai 2004]. However, as commented in [G-
akoumakis and Xylomenos 1996], the needs of organizations working on different projects can, and do, vary, and unfortunately these templates are not fully appropriate for the specific needs for medical (device) software, and do not satisfy the requests of quality requirements documents in this area. Even above mentioned institutions that regulate healthcare and medical devices, have not proposed documentation templates for requirements for medical (device) software [Wang et al. 2014].

3. SOME GENERAL OBJECTIVES FOR MDS REQUIREMENTS TEMPLATE

In this context, let us now concentrate on particular type of requirements template devoted to the MDS. Definition, and separate parts of requirements template, are an important part of initial phases in RE. Designing a requirements template for the MDS, which is a highly specific case study, similarly to other application areas, one must start with a list of essential objectives. They have to consider, and specify, which features of the template have to be achieved. MDS regulatory systems consist of legal regulations that consider the public expectations and the dependability and safety of MDSs. There are several standards for setting the requirements of medical devices. However, they do not prescribe the explicit content of the requirements document, as they contain only recommendations concerning the general approach to produce MDSs. In [Wang et al. 2014], authors proposed specific template that has to satisfy a number of necessary objectives for development of MDSs. The objectives can be summarized as follows:

**Objective 1** - elicitation of requirements should be guided by template and governed by the relevant regulations and standards.

**Objective 2** – elicitation of requirements should be guided by template following several system perspectives; particular perspective should be presented by the viewpoint of one of the system’s environment actor, or partner applications or systems.

**Objective 3** – template decomposition should follow the essential principle of separation of concerns.

**Objective 4** – template must support documenting the device safety requirements and support their ranking; it also should support articulating the safe device/environment interactions.

**Objective 5** – template must support documenting the device’s threat targets; it also must allow specifying the reasonable time for accessing the shared environmental resources; it should provide the needed requirements for a thorough security assessment.
Objective 6 – template must support documenting the privacy requirements; such important requirements guarantee the protection of the user’s personal information.

Objective 7 – functional requirements should be formally documented but at the same time written in the form understandable for non-technical users, which is more or less standard way for requirements specification; the formalism has to support formal and automated verification of the functional requirements like: the space completeness, the dictionary, and behaviour consistency.

Objective 8 – template must be somehow general to document medical device software, i.e. to support a familiar approach.

Presented objectives have to be discussed into details with students of RE course, as a practical part of the course devoted to requirements specification and preparing requirements document. Here, it is important to actively discuss with them all necessary elements, uniqueness and particularities of healthcare and medical software especially directing their attention to privacy, security, and safety.

4. MEDICAL CASE STUDY IN RE COURSE – SPECIFIC REQUIREMENTS AND POSSIBLE ADVANTAGES

Creating successful healthcare and medical software, starting from electronic medical records and patient management systems, to medical devices, considerably differs from, so called, traditional software. Unique domain expertise is demanding, and combined with industry-specific requirements and best software practices, can significantly improve chances of success of the final software product.

Generally, efforts to bring new healthcare and medical (device) software to the market are continually underestimated. Such kind of projects have a host of industry-specific requirements. So, to increase success of such specific software, it is necessary to adopt well-planned and well-executed solutions that are highly secure, flexible, and tightly integrated with the workflow of numerous clinical users.

Therefore, we will now concentrate on critical lessons collected from years of healthcare software development presented in [Macadamian 2017], but we will also aim at changing, modifying and adjusting them, according to our experiences in requirement engineering processes regarding the realization of different scientific and professional projects, as well as in delivering the RE course for more than 10 years at the University of Novi Sad.

Presented lessons represent essential sources of useful guidelines for developing specific healthcare and medical software and understanding the unique requirements of the medical sector and by getting it right the first time. Understanding of uniqueness and proper preparation, can eventually reduce development costs and help end-users deliver better quality healthcare. Lessons can be used in introducing specific case study in the RE course.

Lesson 1: Requirements must be tailored to a practitioner’s specialty – Software developers usually do not understand that requirements highly depend on specialization and/or general knowledge, of clinical users, medical staff and caregivers.

IMPORTANT: So, it is important that during the requirements elicitation, requirement engineers and software developers have to be constantly in contact with healthcare and medical specialists in the specific area, in which the software is being developed.

Advantages of the Medical Case Study for the RE course (AMCSRE): It is important to present, to the students, specific characteristics of medical software and high importance of including medical experts in all phases of the requirement engineering processes. It also is important to outline specific problems and difficulties, i.e. different state of mind of software engineers and medical experts and suggest ways to make better and more understandable communication.

Lesson 2: Privacy restrictions must be balanced with usability requirements - Personal medical data, collected from different sources and using different medical devices, has made patients
privacy essential for end-users, healthcare organizations, and governments. Governments have created laws, essentially requiring the implementation of two security systems: access control (not overly restrictive - offer an override mechanism to prevent inappropriate access) and an audit trail (serves as a record of all events that can occur through the normal use of the application, and it alerts users to be careful when accessing data making them aware that their actions are being recorded).

**IMPORTANT:** It is important, when developing healthcare and medical software, to balance access control and audit trail mechanisms carefully. Privacy laws must be respected, but allow users to efficiently accomplishing their tasks.

**AMCSRE:** Special attention within the RE course must be paid to explaining to students the significance of privacy, within medical software. Also, they have to be aware of the need of inclusion of key stakeholders and legal regulations.

**Lesson 3: Model hospital processes before you write a line of code** – Healthcare and medical software has to map appropriately into the existing processes in medical institutions. For very complex and multifunctional healthcare and medical architectures, the separation of components of medical device software is of crucial importance. Such (new) components must not negatively affect, or significantly change, the existing processes and data processing within (existing) software.

**IMPORTANT:** It is critical that all processes must be understood before software is designed and well documented. Each hospital process must be fully understood before any software (or new component that will be integrated) can be designed to support it.

**AMCSRE:** This characteristic is important for almost all kinds of software, but it is especially important for software and particular components that are being developed for (existing) healthcare and medical software. Students have to know that all key and essential sources, for requirement elicitation, must be seriously considered and a comprehensive view of them is necessary. They must be aware of all aspects and complexities of the existing environment and architecture.

**Lesson 4: Doctors have no patience for software that doesn’t save them time** – Healthcare and medical workers, in hospitals all over the world, have to take care of constantly growing number of patients and, on the other hand, to do their jobs as efficiently as possible. Software developers usually do not consider seriously the time restrictions healthcare and medical staff face.

**IMPORTANT:** When designing and implementing a new healthcare and medical software, developers have to be aware of the doctor’s desire to finish clinical paperwork as quickly as possible.

**AMCSRE:** Healthcare and medical (device) software needs to be extremely responsive and informative. So, students must be aware of fact that, in this case, analysis and negotiation phases are essential. Healthcare practitioners and doctors must take essential part in these processes and precisely articulate their wishes and needs. Accordingly, requirement engineers and software developers must blindly follow their (doctor's) needs.

**Lesson 5: Validation is far more important than verification** – Generally, for health and medical software, the emphasis must be on application stability. Thus, software developers, with little healthcare experience, usually take care of software verification, but do not pay enough attention to validation. Software can always be made to do what you want it to do (verification), but it is noticeably harder to be sure you are building the right software for the job (validation) [Macadamian 2017]. In healthcare domains, software validation is critically important, requires strong domain expertise, and must be used to test all exceptional and rare medical cases.

**IMPORTANT:** A good software validation process will resolve multiple problems before implementation goes too far.

**AMCSRE:** It is necessary to emphasize to students that, in the implementation phase of healthcare and medical (device) software, validation process is extremely significant. Also, it is necessary to point
out to them that obtaining adequate expertise is crucial for developing healthcare and medical (device) software.

**Lesson 6: Input validation should not block users** – Healthcare applications, but especially medical device software, deal with rather complex data entries. Accordingly, important software development considerations must concentrate on accurate methods for data validation and treatment, and cannot be slowed by the requirements of software.

**IMPORTANT:** In healthcare and medical environments it is recommended to follow effective input validation best practices. It is necessary to ensure medical care is never delayed because of software requirements.

**AMCSRE:** It is important to emphasize to students that, in healthcare and medical software development, input validation is necessary to take special care of: to keep the number of required fields to a minimum; to not to block users from submitting a form; to provide defaults to handle missing (not provided) data; data quality checks should account for incomplete or a ‘fake’ placeholder data and advise users to complete the data entry once the values are known; to continue functioning in case when incomplete data appears. It is also desirable to invite doctors and actively include them in practical classes and provoke active discussion with students, in order to try to emphasize and illustrate different styles of medical staff thinking.

**Lesson 7: Database Designs must be flexible, as workplace requirements are likely to change** – In health and medical institutions (predominantly hospitals), work processes and requirements are continually and constantly changing, more than in majority of traditional software or environments. It is necessary to keep these constraints in mind, while developing and deploying software, and to be able to handle changing requirements.

**IMPORTANT:** With the appropriate data model, it is easy to handle these challenges and allow for continuous updates.

**AMCSRE:** This lesson is especially important to students, in order to illustrate and emphasize importance of requirements management, especially requirements changes and tractability.

**Lesson 8: Hospital processes are highly collaborative and asynchronous** – In hospitals, usually, majority of individuals work autonomously, in order to provide adequate healthcare. The usual scenario is that communication is carried out between caregivers and patients. When a difficult case appears, multi-staff collaboration is needed to gain experience, start discussion and reach solution.

**IMPORTANT:** Ideally, good healthcare and medical software in hospitals should support an autonomous workflow process, where each member is empowered to perform her/his tasks. But, the problem usually appears as each individual may have radically different working patterns. As a general rule, doctors’ schedules are more unpredictable as they are frequently getting interrupted. So, communication between healthcare and medical professionals is predominantly asynchronous.

**AMCSRE:** As healthcare and medical software has specific requirements, to follow radically different working patterns and specific communication needs, and thus developers must be fully aware of this. It is necessary to expose students to this uniqueness in developing healthcare and medical software, and insist to pay special attention in elicitation of requirements, as well as in trying to fully understand workflows of existing processes. Ideally, requirement engineers have access to multiple sources for requirements, especially including doctors’ expertise. Preferable approach of elicitation could be to use viewpoint analysis and interviews, scenarios and observations. Furthermore, students have to be introduced to the fact that doctors are always busy and have specific way of thinking, which is very different from that of ICT developers and experts.

**Lesson 9: Supporting interoperability standards is tougher than it looks**— Rather complex interfaces, with external systems, have to be implemented in majority of health and medical software.
To simplify and support the processes, it is obligatory to incorporate in software solution adequate interoperability standards.  

**IMPORTANT:** Interoperability is one of the biggest challenges in healthcare and medical ICT. So, the initial activity in realization of such kind of software is to ensure that the interoperability standards are properly planned and estimated.  

**AMCSRE:** During presentation of medical case study to students, it is necessary to explain them that once they have obtained a very thorough, granular task breakdown structure and estimates, they have solved half the problem [Macadamian 2017]. The other important part is avoiding the high risk of reworking the software as it evolves, so that they also must precisely and cautiously consider procedures and standards starting from very early phases of requirement engineering.  

**Lesson 10: Not all clinical users are doctors and nurses** – The most difficult groups, in developing healthcare and medical software, are doctors and nurses. So developers often neglect other groups of users who, directly or indirectly, use the software. Nevertheless, it is evident that doctors' and nurses' actions most directly affect the quality of patient care.  

**IMPORTANT:** Software developers have to take care to satisfy the requirements of healthcare providers, but also of users who may not use the application directly and can benefit downstream from the collected data. It is also of high importance, for modern and complex multisource data collection (as smart medical devices) software. This process is known by usability experts as stakeholder identification.  

**AMCSRE:** It is essential to explain to students that it is highly significant, in requirement engineering processes in healthcare and medical software, to include and frequently interact with a wider range of stakeholders and possible users of the software that is to be developed. Also, they have to be prepared for the fact that these initially identified stakeholders and experts can also identify other different groups of users who will also use the application. As a result, they become a valuable sources of requirement elicitation. This is especially important for medical device software, which is an essential part of modern, complex healthcare and medical software.  

5. **CONCLUDING REMARKS**

Developing healthcare and medical software presents a set of unique software development challenges and it is usually complex and fraught with risks. Such kind of software requires precisely elicited, analyzed and specified requirements using multiple data and requirement sources and high involvement of domain experts, foresight and specialized knowledge.  

This being the case, it is essential to include in a Requirement Engineering course specific hints, suggestions and issues to make students aware of the most commonly encountered issues in healthcare and medical software development, so that they can avoid typical mistakes, reduce schedule, and cost overruns, and help healthcare practitioners obtain and deliver better care for the patients, in future real-life situations. Special attention must be paid to the fact that software must be flexible and adaptable to clinical workflow processes and also that clinical end-users should be able to leverage well-designed systems that meet their needs as well.  

Documenting the requirements for healthcare and medical software (especially for MDSs) is a very challenging task, due to its uniqueness and specific nature, and the stringent regulatory rules. These systems ought to exhibit the highest dependability qualities such as safety, security, and availability. A systematic approach to tackle gathering and documentation of requirements of the system, in general, is essential and it is necessary to precisely explain such elements to the students. Objectives for designing a suitable requirements template document, for healthcare and medical software, is also important and specific elements of the template have to be presented to the students. Additionally, it could be de-
sirable to invite doctors to actively participate in classes and discuss healthcare and medical software issues with the students. It could be highly useful to students to have contact and communication with the very specific style of thinking of key medical stakeholders.

REFERENCES


Suzanne Robertson and James Robertson. 2012. Mastering the requirements process: Getting requirements right. Addison-wesley.


Technology and the Reincarnation Cycles of Software

HANNU JAAKKOLA, Tampere University of Technology
JAAK HENNO, Tallinn University of Technology, Estonia
JUKKA MAKELÄ, University of Lapland

The idea to write this paper was based on a discussion with colleagues about the progress of Artificial Intelligence (AI) over the last five decades. The authors have been active in the field since the 1970s. AI is just one example in our paper, but it is also a good example of a “wave driven technology”: it bounces out from time to time in the role of emerging technology (technology that has the capability to cause major changes) in ICT. Technology analyst companies now consider it to be at the frontier again, having an influence both itself and embedded in a wide variety of applications. It has a wide influence on the whole of society and business, e.g., in employment structure and business processes. In our discussion, we wish to find answers to the questions: What is the reason for the same technology reappearing time after time and being reported as one of the important sources of changes? What are the differences between the waves or generations, and the reasons behind them? We have built a hypothesis that the kernel of AI has remained more or less the same but that the changes are triggered by certain enabling technologies, used to implement the AI applications of each time. We term these enabling technologies “resources”. The same progress as that in AI fits many other areas of ICT. We transferred our discussion to the area of data management, which become the second goal of our analysis. The current data driven ICT manifests itself in big data analysis and new database technologies, but has roots in the early decades of computing. In general, at the beginning of the computer era (1940s), the aim was always to develop intelligent computer based software systems within the framework of technical restrictions. We also see the same limitations in the changes of software engineering tools and practices, as well as in the interpretations of ICT related concepts (semantics of concepts changes with time). In our paper, we will briefly introduce our ICT Change Analysis and Categorization framework and introduce the current state-of-the-art of emerging ICTs. The aim of our paper is to act as a discussion opener in the question “How was the progress of software and software engineering born – what are the factors controlling and enabling the changes?” Our aim is to point out that, in addition to excellence in the area of frontier technologies and tools, a lot of basic level understanding in the area of computing fundamentals is needed – just to understand the root of the current situation.

Categories and Subject Descriptors: I.2 [Artificial Intelligence]: Applications and Expert Systems; H.2 [Database Management]: Systems, Languages, Database Machines;


Additional Key Words and Phrases: Technology Analysis.

1. INTRODUCTION

Emerging or emergent technologies (also known as frontier technologies) are technologies that are perceived as capable of changing the status quo. Usually these technologies are new. Sometimes older technologies also provide the potential for remarkable changes. Emerging technologies have a radical novelty and potential for fast growth and impact. Uncertainty is also sometimes connected to emerging technologies – the progress is not as expected. The tools of technology analysis provide a means for detailed analysis of the recognition of the real life cycle of the technology – see e.g. [Jaakkola et al., 2017].

The starting point of this paper is the finding that many current frontier innovations have their roots somewhere in the past. The innovations of today may be evolutions or new generations of earlier innovations. The analysis of long-term progress also points out waves of the same innovations; some innovations appear in waves, some appear only once in such a role and remain a part of the mainstream. The idea to write this paper relates to the discussion of the role of Artificial Intelligence (AI) as a part of current ICT. We noticed that since the 1950s it has reappeared time after time and been reported as one of the important emerging technologies – it is at the top of the wave once again. We counted the current wave to be the fourth generation of AI ranked as emergent technologies; there

Authors contact information: hannu.jaakkola@tut.fi; jaak.henno@ttu.ee; jukka.makela@ulapland.fi

Copyright © by the paper’s authors. Copying permitted only for private and academic purposes.
could be more depending on the interpretation. Similar waves can be found in other technologies, such as data management, data analysis, autonomous devices, networking, certain fields of user interfaces, etc. This observation pushed us to think about the reasons for this wave-formed development.

Our hypothesis is that the essence of these wave-formed technologies has remained the same for decades, but the new resources available enable new modifications and capabilities in the applications. We define a wave-based technology as a technology that is reported several times in technology analysis to be one of the emergent technologies.

The aim of this paper is not to focus on technology studies in a wide scope. We will touch that topic only in the form of a short overview. Our recent paper [Jaakkola et al., 2017] from May 2017 contains a good analysis of the current situation of technological changes. We collected data from more than twenty reports of leading analysts with titles like “Technology outlook”, “Leading technologies 2017”, “Technology Forecast” etc. The perspective of the analysis was five years ahead from 2017; some of the reports had a longer time span, up to 20 years. The analyst companies cover Gartner, Cisco, Deloitte, Forrester, Fjord, Forbes, IDC, PWC, and the World Economic Forum among others. An overview of this study is in Section 2 of this paper; the aim of this overview is to provide a means to understand how the world appears now through the eyes of a technology analyst.

Above we pointed out “what the aim of this paper is not.” By this exclusion we wished to inform the reader that we want to focus on a different topic, even though at the beginning of the paper there is a lot of discussion of technological changes. The research problem of this paper is defined as follows: “What are the reasons for certain technologies appearing in waves in the role of emerging technologies in the ICT sector?” The problem is divided into three research questions:

1. What is the current state of emerging ICT related technologies?
2. What kind of evidence can be found to support the hypothesis of the wave-based appearance of certain technologies?
3. What kind of explaining factors can be found for the wave-based appearance?

The research questions also form the structure of this paper. In Section 2 we give a short overview of the current state in the area of emerging ICT technologies. In Section 3 we give two examples of wave-based technologies – Artificial Intelligence and Data Management (DM). Both of these have an equally long history as computers. Their role has changed during the decades. Both of them belong to the category of leading emerging technologies in the ICT sector today (and also in the past) – AI in the form of deep learning and neural network based (embedded) applications; DM in the form of intelligent data analytics and block chain technology. Section 4 handles our hypothesis related to the role of enabling technologies that have remained more or less the same during the decades. In Section 5 we conclude the paper.

2. A SHORT LESSON IN TECHNOLOGY ANALYSIS

For our conference paper [Jaakkola et al., 2017], we analyzed close to twenty technology analysis reports made by leading analysts. The aim of this analysis was to build a big picture of the role of ICT technologies in the current society. The focus of this paper was on the education sector, especially in changes in professions / employment and the needs to renew education. The results are organized in table format (p. 743) and illustrated as a classification structure (ICT Change Analysis and Categorization Framework – ICAC (p. 747). We will not return to this topic in detail in this paper, but quote the presented framework as an introduction to the topic (Figure 1).

In the figure, the current key technologies are classified according to Freeman and Perez [Freeman and Perez, 1988], which is a commonly used in the area of technology research. They classify the improvement capability of new technologies into four categories:

- Incremental changes appear continuously in existing products and services (continuing the existing trend);
- Radical changes appear when new research findings are applied in products to transfer their properties or performance to a new step or cycle (movement to a new trend);
• Changes in technological systems are caused by combinations of several incremental and radical innovations in societal and organizational systems;
• Changes in paradigms are revolutionary and lead to pervasive effects throughout the target system under discussion (in our case, the information society).

We have organized our findings in the form of a mind map. The arrows between the categories indicate the transfer between categories. Classification of technologies into categories is not completely unambiguous, but presents the subjective and best understanding of the authors.

![Mind Map Image]

**Fig. 1. ICT Change Analysis and Categorization framework [Jaakkola et al. 2017].**

In the “Incremental changes” category, there are technologies that are widely adopted in mainstream software products and development. Some of these have been revolutionary at the time of their first appearance, but in time have become a part of the mainstream. Technologies in the “Radical” category still have such a competitive edge that, applied in a novel way, some benefits in the form of faster growth or better quality can be reached. The current trend accelerates or continues on a higher level after a step upward. The technologies classified in the category “Changes in technological systems” enable new opportunities in society or business in the form of process changes. The effect of changes appears in behavioral models, which are enabled by certain technologies or a combination of them. The “Paradigm” category is the most interesting part of the analysis. These technologies are in some way revolutionary now. They change the world permanently, like cloud computing, which has provided access to scalable computing resources with minimal costs or open data access to enormous data sources to cultivate knowledge of it by using big data analytics. Our term “emerging” or “frontier technology” corresponds to the paradigm category.

This short overview, we hope, has provided access to our way of thinking of the role of technologies as a part of progress in ICT. For further analysis (Section 3), we have selected two technologies, which are handled from a historical perspective as evidence of wave-based changes:

• Artificial intelligence is reported as belonging to the category of paradigm changes. All current analysts report it as the most important source of changes today. It provides new products, replaces human work, and is embedded in a wide variety of intelligent systems. AI has, however, a long history, and it is easy to find it on the frontier already a couple of times earlier.
• Data management is found in our classification in three categories. Enriched data and context sensitivity have radical characteristics and the management of unstructured data, open data (management), and big data analytics still has the potential for paradigm level changes.
In Section 3 we will handle these two technologies from a historical point of view, as a part of progress during the decades of computing.

3. WAVE-BASED EMERGING TECHNOLOGIES – TWO EXAMPLES

In this section we will provide the reader with two examples of wave-based technologies – Artificial Intelligence and Data Management (DM). Both of these have as long a history as computers although their role has changed during the decades. Both of them belong to the category of leading emerging technologies in the ICT sector today – AI in the form of deep learning and neural network based (embedded) applications; DM in the form of intelligent data analytics and block chain technology. The discussion below points out their earlier waves (over a period of 60 years). In this section we concentrate on describing the changes they brought about; the reasons behind the phenomena are discussed in Section 5.

3.1 Artificial Intelligence

The term “Artificial Intelligence” was coined by John McCarthy in 1955. He is also the inventor of Lisp programming language. The first wave of AI in the role of emerging technology focused on programming languages like Lisp (in the 1950s) and later Prolog (early 1970s; Alain Colmerauer and Philippe Roussel). The main feature was modifiable code – the program (application) was able to modify itself, which was the source of learning capability and “intelligence”. The second wave relates to expert systems (1970s – 1980s), which were based on rule or frame based reasoning and hypertext. Ultimately, neither of these waves had a significant role in the progress of ICT, but attracted wide interest among the scientific community and industry – for a while.

Somewhere on the timeline parallel to the era of simple expert systems (1980s) there was an aim to build AI directly into the computer architecture; we call this the third wave of AI. One of these commercial architectures was Symbolics, which implemented Lisp in its processor. The origins of the company developing this computer, Symbolics Inc., were at MIT, in Cambridge, Massachusetts. Symbolics computers were produced in the 1980s; however, they never became a commercial success and finally disappeared from the market. Another architecture level innovation – or actually a set of innovations – was developed in the Japanese 5th Generation Computer System (FGCS Project). This project was an initiative of Japan’s Ministry of International Trade and Industry beginning in 1982. The aim was to create massively parallel computer architectures based on the concept of logic programming (Prolog). The Institute for New Generation Computer Technology (ICOT) had a revolutionary ten-year plan for the development of large computer systems, which were applicable to knowledge information processing systems. The idea never materialized on the commercial product level. It provided, however, an enormous step forward for industrial software development in Japan, as well as being a source of (parallel) computer architectures implemented by Japanese electronics companies in their products. To conclude, AI support on the architecture level remained a promise, in spite of enormous investments in Japan and marketing efforts in the case of Symbolics. Why? – would be a topic for another conference paper. In spite of a lack of commercial success, the FGCS project especially was followed with interest in the international scientific community and industry.

---

3 https://en.wikipedia.org/wiki/Symbolics
4 https://en.wikipedia.org/wiki/Fifth_generation_computer
5 The first author collaborated actively with ICOT; three scientific papers were published in 1984-1985.
Nowadays – in the fourth wave of AI - the key elements are machine learning technologies, neural networks, intelligent systems (as an application), and a variety of technologies related to data analytics. All technology analysts see it providing revolutionary opportunities in a wide variety of applications that replace human work, support humans in their work, in the form of robotics, as part of a variety of intelligent devices, etc. A good example, as an evidence of our hypothesis related to the role of enabling resources, is deep learning. Intelligence is based on learning. Deep learning theory has its roots in the 1980s. In practice it is question on the use of nonlinear statistics in data analysis and opportunity to learn from masses of data. More than thirty years was needed to make these theories work in practice – parallel computing and big data technologies have made it possible; earlier data was the bottleneck (quoted Professor Aapo Hyvärinen’s seminar presentation, Helsinki, August 31st, 2017).

3.2 Data Management

The second example we have selected for overall discussion in our paper is DM. In Section 4 below we will handle the progress of magnetic storing devices. The period before these in commercial use data was stored and handled in the form of punch cards, punch tapes etc., which have their roots in the pre-computer era. Magnetic devices, the first tapes and later disks, replaced these, but for a long time these primitive media remained a part of daily computing. The role of data in ICT sector has been discussed by scientists during the decades. The traditional term “computer science” refers to the importance of computing. The term “Data Science” as the replacement was presented by Peter Naur\(^6\) already as early as 1960. Nowadays the term is in common use and represents the interdisciplinary field about methods, used to extract knowledge from data, either structured or unstructured. It unifies theories drawn from mathematics, statistics, information science, and computer science, in particular machine learning, classification, cluster analysis, data mining, databases, and visualization.

Due to history and poor computing power, the first wave of data management was based on the sequential handling of data. When magnetic devices became available, the first wave of simple data management was still being used. The first database systems appeared in the late 1960s and early 1970s and then the second wave of DM began. The first databases implemented were implemented as navigational linked datasets that were close to the physical implementation of data. The progress enablers were the growth of computers' processing capacity, novel mass memories, and high-level programming languages (like COBOL in business applications).

The third wave of data management was based on the idea of Edgar Codd in the early 1970s. His invention of using the table format in data presentation and relational calculus to handle the data separated database architecture from the physical implementation of data. During the 1970s, several projects around the world were developing relational database ideas into commercial products. Computer users had to wait until the 1980s for the first relational database products. Eventually some mainstream products became worldwide standards. One of these was the SQL database language. As a result, the era is also known as the SQL Database era. In the 1990s, progress continued in the form of object databases and some other variants, which we consider more as an evolution step of relational databases than as a real reincarnation wave. The 1990s also gave birth to the open source relational database system MySQL that become one of the forerunners of the open source software movement.

The next step in DM was recognized with the appearance of a growing need to handle non-structured data. This was based on the need to handle live data streams, Internet based distributed data collections, as well as open data. This started the fourth wave of DM and on the timeline it is located in the early 2000s, and is continuing. It is interesting to note that the same actors are participating in the present wave as earlier. Where MySQL was one of the important products in the

\(^6\) See https://en.wikipedia.org/wiki/Peter_Naur.
SQL database era, there is now non-SQL product, MariaDB, in the same position. Both have the same innovator, Finnish Monty Widienius.

At the moment we can see the beginning of the fifth DM wave – i.e., the appearance of block chain technology. In analysts’ reports, it is seen to be in its breakthrough phase and has growing potential in a variety of applications that need trusted data (handling). Block chain is not a true replacement of other DM technologies, but extends the scope of DM into new areas (security, trust creation in data intensive operations, trusted contracts).

The analysis above is based on the “common sense” of the authors. Some Internet resources have been used to check time-related details. Consequently, there are no references in this part of our text.

4. THE ROLE OF ENABLING TECHNOLOGIES

Our hypothesis is that enabling technologies as key resources are the most important explanatory factor for the wave-based periodical re-appearance of certain emerging technologies. We list three such technologies in the ICT sector:

- computing power and memory capacity (= VLSI, circuit technologies),
- data storing capacity, and
- data transmission speed.

There are several others, which we do not handle in this paper.

The book by [Endres & Rombach, 2003] lists a variety of ICT-related trends that have affected the progress of systems and applications. Legislation indicates regularity in technological changes. These are based on theory and have evidence in practice, and also indicate a systematic continuum in time. The book covers more than two hundred laws relevant to ICT, classified in more than 20 categories. We have picked out the three most important laws that explain a lot of the progress in ICT:

- Moore’s law: the price/performance of processors is halved every 18 months;
- Hoagland’s law: the capacity of magnetic devices increases by a factor of ten every decade; and
- Cooper’s law: wireless bandwidth doubles every 2.5 years.

Additional laws that supplement the above-mentioned are:

- Butters’ Law: the amount of data one can transmit using optical fibers doubles every nine months;
- Rose’s Law: the number of qubits in a scalable quantum computing architecture should double every year; and
- Nielsen’s law: A high-end user's connection speed grows by 50% per year

The book provides references to the original sources or to the sources reporting the principles of the laws introduced. The latter three are from separate sources.

Moore’s law [Endres & Rombach 2003, pp. 244-247] is attributed to Gordon Moore, the co-founder and chair of Intel in the 1960s. The law was published first in an article [Moore, 1965]. Even though the law deals with the packing density of transistors in VLSI chips, the practical consequences are seen in the doubling of processor processing capacity (every 18 months) and the doubling of memory chip capacity (for the same price, every 15 months). Moore’s finding is based on his observations from the late 1950s and his “optimistic” prediction was that “There is no reason to believe it will not remain nearly constant for at least 10 years”. The evidence of today (Figure 2), however, shows that the law has remained valid until now (50 years) and the current understanding of the progress in material and VLSI technology do not yet foresee any physical limits for its continuation.

---

7http://www.dataversity.net/brief-history-database-management/
http://www.quickbase.com/articles/timeline-of-database-history
Fig 2. Exponential growth of computing power - Moore’s Law [Roser & Richie, 2017]

**Rose’s law** [Jurvetson, 2017] extends the progress defined by Moore to the era of quantum computers. In his article, Jurvetson describes that, as in Moore’s Law, the growth of the amount of qubits in a quantum computer is exponential - currently the doubling time of the number of qubits in a quantum computer is one year. In addition, the computational power of the quantum computer also grows exponentially with the number of entangled qubits (parallelism). This makes the growth compounded. We agree that the quantum computer era is not yet a reality outside laboratories. It is seen as a promising technology of the future and it fosters belief in the continuing of this progress even after Moore’s era.

**Hoagland’s law** [Endres & Rombach, 2003, pp. 247-249] predicts the capacity of magnetic devices to increase by a factor of ten every decade. The law is credited to Albert Hoagland. He was one of the developers of the IBM magnetic disk. IBM 305 RAMAC was the first commercial computer that used a moving-head hard disk drive, the RAMAC 350. At the time of its introduction, in the middle of the 1950s, the RAMAC 350 storage unit could hold the equivalent of 62500 punched card or 5M characters. Hoagland predicted a remarkable increase in the area density of magnetic storage devices, which he predicted to increase by 25-30% per year, i.e., doubling every three years. The growth has been exponential and provides the means for the large-scale data storages of today (Figure 3).

There are three laws above that refer to the growth of data transmission. **Cooper’s law** [Endres & Rombach 2003, pp. 249-250] reports that wireless bandwidth doubles every 2.5 years, which indicates growing opportunities for wireless data. Martin Cooper is an American pioneer and visionary in the area of wireless communication and mobile technology. His prediction (from 2000) is based on empirical analysis without theoretical evidence. Cisco has reported on current progress in two reports [Cisco 2017a; Cisco 2017b]. Global mobile data traffic in 2015 reached 3.7 exabytes (10**15) per month; in 2020 it is expected to be 30.6 exabytes. The traffic has grown 4000-fold over the past 10 years and almost 400-million-fold over the past 15 years. Even though these numbers represent the growth of the traffic itself, the numbers also indicate the improvement in transmission technologies. **Butters’ Law** [Roser & Richie, 2017] supplements the vision of progress in the data transmission area. It states that the amount of data that can be transmitted using optical fibers doubles every nine months, which indicates that the cost of transmission by optical fiber is halving every nine months. Nielsen’s law indicates the same progress [Nielsen, 2017] When combining the progress in the area of wired and wireless data transmission, we can simplify the progress in the form of Fred’s Law: the regular priced data transmission capacity doubles every year. The origin of this law is unknown (to the authors of this paper), but found some twenty years ago. However, the law simplifies real progress
in a simple and intelligible way, in spite of the fact that it is not exact – only indicating the expected exponential growth.

![Graph of increasing hard drive capacity, 1980-2011](image)

Fig 3. Increasing hard drive capacity, 1980-2011 [Roser and Richie, 2017]

5. CONCLUSION

The aim of this paper is to open up discussion on the factors allowing the periodical appearance and changes in important areas of ICT. We have given a general overview of the current situation based on selected technology analyst reports. We have selected two key technologies that have followed the pattern of periodical, wave-based changes.

In Section 4 we handled three enabling technologies that explain – at least partially - the changes in AI and DM. The answers to our research questions and solution to our research problem are illustrated in Figure 4. There are dependencies between the elements discussed:

- Progress in VLSI technology indicates fast growth in processing power and memory capacity provide a platform for more complex software and bigger amounts of data (in primary memory) for fast processing,
- Availability of big mass storage capacity to allow access to big data repositories to allow intelligent data handling, and
- Fast data transmission to allow distributed (and also parallel) processing.

All of these are key resources in new emergent applications and act as the evidence for our hypothesis.

No doubt the progress in VLSI technology is the most important of these. Both massive data storages and data transmission technology use improved and effective processing elements as their components. We claim that the *essence* of applications remains the same over the waves. AI still has the same principal task to solve as applications had in the 1950s during the first wave. Lack of resources allowed only simple applications; currently the resources allow more complex applications based on big data and scalable computing resources. The applications are constructed of the permanent *essence* and the *technology enabled variant*.

An additional aspect that is worth recognizing which explains the sequence of waves is the *evolution of concepts*. It is clear that the semantics of concepts is also changing over time. The AI of today does not have semantically the same meaning that it had in the 1950s, in the 1980s, etc. The same goes for the concept of DM. In Figure 4 we have included a box “software quality” with reference to “data quality” to indicate that even this concept has different semantics in different times.
In the 1950s, the period of a shortage of resources, quality software was based on minimal usage of the main memory (small size) and effective processing – the first wave. In the 1980s the focus was on the logical structure and maintainability of software – the second wave. The next wave (third) focused on quality (process) management. It was based on the idea that quality software is a product that is produced by a high capability and maturity processes.

Software product quality also has its own standard. The current version (2011), ISO 25000\textsuperscript{8} series, is the evolution of and replaces the series of ISO 9126\textsuperscript{9} (from 1991 and 2001) standards. These indicate the evolution of the concept of software product quality in ten years cycles. One of the remarkable changes is that ISO 25000 includes a new area, data quality. This indicates that focus must also be placed on the quality of the data that is handled by software products; is this the fourth wave?

At the beginning of the paper, we set our research problem and three research questions derived from it:

1. **What is the current state of emerging ICT related technologies**: Section 2 has given the answer to this in the form of a short technology review.

2. **What kind of evidence can be found to support the hypothesis of the wave-based appearance of certain technologies**: In Section 3, we have handled AI and DA as example technologies. Several others can be found, in addition to these. In Section 5, we have also handled the concept evolution, software quality as an example.

3. **What kind of explaining factors can be found for the wave-based appearance**: We have given an example of three enabling technologies that have an effect on the progress in the application fields discussed.

These answers act as a solution to our research problem “What are the reasons for certain technologies appearing in waves in the role of emerging technologies in the ICT sector?”.

Why must the waves be recognized? Being prepared to understand the changes, we have to be eager to recognize both the essence and the technological changes that provide new opportunities in applications. To understand the future, we have to know not only the history, but also the trends that lead us to the future.

---


Identifying Up-to-the-Minute Topics in Software Process Improvement Research

HARRI KETO AND HANNU JAAKKOLA, Tampere University of Technology

The development of standards and methods used in software process improvement has for many years been oriented to the identification of best practices, but recent trends show an increasing interest in new research topics, especially in the category of small-to-medium-sized enterprises (SME) and very small entities (VSE). What are these research areas, and what kind of relationship do they have with software process improvement? The goal of this study is to aggregate topics that are up-to-the-minute or very new openings in research from the viewpoint of SMEs and VSEs. Using a systematic mapping study and confining it to the last three years, 26 studies were selected for deeper analysis. Proposals for further studies are made by suggesting practically relevant research questions. Results indicate that software process improvement research remains a robust discipline with plenty of possibilities. The challenges can be met by co-operating with researchers from other disciplines.

Categories and Subject Descriptors: D.2.9 [Software Engineering]: Management—Software Process Models;

General Terms: Mapping study; Software process improvement; very small entities; small-to-medium-sized enterprises

Additional Key Words and Phrases: SPI, VSE, SME

1. INTRODUCTION

As part of the larger quality management approach called continuous process improvement, software process improvement (SPI) has become a serious engineering discipline for improving the quality of software engineering and its products. According to [Sommerville 2011, p.706], software process improvement means understanding existing processes and changing these processes to increase product quality and/or reduce costs and development time. Today, there are many international SPI standards and disciplined methods that are used in the field. For many years, the development of SPI standards and methods has been oriented to the identification of best practices and their mapping to form a coherent framework like CMMI, ISO/IEC 15504, or ISO/IEC 29110. However, a recent trend shows increasing interest in new areas like customized SPI models and the combination of agile methods and process improvement [Kuhrmann et al. 2016]. These do not contradict each other. On the contrary, recent literature studies also show an increasing interest in the merging of different approaches.

The goal of this study is to identify up-to-the-minute research topics and presumptions about the overriding knowledge bases of SPI in the context of very small entities (VSEs) and small-to-medium-sized enterprises (SMEs). The European Commission has defined the category of SME to consist of micro enterprises (staff < 10), small enterprises (staff < 50), and medium-sized enterprises (staff < 250) [European Commission 2005]. ISO/IEC JTC1/SC7 Working Group 24 has defined the acronym VSE to refer to an enterprise, organization, department, or project having up to 25 people [ISO 2016].

A systematic mapping study was chosen as the research method to be applied [Petersen et al. 2015] [Kitchenham et al. 2011]. We have limited the study to research papers that have been published since 2015 in journals or conference proceedings. The emphasis is on issues, thus the number of factors or studies is not our main interest. If the results are encouraging, it is possible to enlarge this study to cover a longer period and spread the search for a more complete set of papers.

This study is limited to small-to-medium-sized enterprises (SMEs) and very small entities (VSEs). There are two main reasons for this. Firstly, a considerable number of software organizations belong to this category [Richardson and Cresse von Wangenheim 2007][Larrucea et al. 2016]. Secondly, many
SMEs and VSEs have implemented agile practices in their development process to a greater or lesser extent. The influence of the agile paradigm poses challenges for SPI approaches. As some papers did not explicitly define the acronyms SME and VSE, it was unrealistic to make reliable deductions about the data between the SMEs and VSEs. The risk is that the categories do not separate the cases in a controlled way.

This paper is structured as follows. In section two, we present related studies in brief, which have been performed recently. In section three, we describe the research design and implementation. The results are presented in section 4. In the last chapter, we complete the paper with a discussion and proposals for further research.

2. RELATED STUDIES

In this section, we briefly present recent literature reviews closely related to our study.

In [Kuhrmann et al. 2016] a large systematic literature study is reported where 769 publications were analyzed. The study covers the field of SPI over a period of more than 25 years, presents a big picture of the state-of-the-art, and maps out directions for future research. There are no up-front constraints concerning the organizations, SPI methodology, or software engineering approach. The companies were scaled in three categories: VSE/SME, other company sizes, and global software engineering organizations (GSE). According to the study, the academic community shows a growing research interest in SPI in the context of very small entities and small-to-medium-sized enterprises. While standard assessment and maturity models like CMMI and ISO/IEC 15504 are considered to be well researched, other main research trends focus on new/custom models, success factors, and agile and lean development practices as part of SPI.

There are examples of literature studies that are confined to research on the combination of agile and maturity models. From the viewpoint of CMMI, a systematic literature review in [Henriques and Tanner 2017] brings out two major themes in this context: the coexistence of the agile with CMMI, and the development of agile principle based maturity models. After excluding and reviewing, the final working set was 39 articles. One of their suggestions on future research topics is how to attain higher levels of CMMI maturity using only agile methods. In [Selleri Silva et al. 2015], a similar research interest is reported.

In [da Silva and Carneiro 2016] the systematic literature study focuses on which difficulties are faced by these companies regarding SPI. The study is limited to SMEs. After filtering the relevant articles, 36 articles were left for further investigation. The study indicates that discussion and research on the problems and success factors of SPI implementation in SMEs are relevant and current.

3. RESEARCH DESIGN AND IMPLEMENTATION

In this section, we describe the research design and implementation. First, the research method is presented and justified. Then, after describing the research questions, we explain the research steps and make remarks about considerations and constraints concerning each step.

3.1 Research method

The research method is based on a systematic mapping study. According to [Petersen et al. 2015], a systematic mapping study is designed to give an overview of the research area by classifying and counting contributions in relation to the categories of the classification. In a nutshell, according to [Kitchenham et al. 2011], the goal is the classification and thematic analysis of literature, the research questions are generic, and only classification data is collected. Quality evaluation is not essential and it is complicated by the inclusive nature of the search. The result is a categorized set of papers related to a topic area. We make an exception in the method such that although we count the
papers per category, the number of papers is interpreted as being irrelevant. We justify this by the fact that even an individual contribution can be relevant from the point of view of further study. We restrict this study only to the past few years, so the identification of wide trends is not possible either. As for trends, we refer to related studies in section 2.

We have applied the following five research steps:

1) Defining research questions
2) Search for studies
3) Study selection
4) Classification of papers
5) Analysis

The process has been very iterative. Re-reading was necessary because two of the four classes were partly based on analysis. We used two general classification categories and two categories were deduced by analyzing the papers. Analysis was performed by making cross-reference tables between the categories.

3.2 Defining the research questions

The primary goal of this mapping study is to create a preliminary categorization of up-to-the-minute SPI research topics in the context of SMEs and VSEs. The second goal is to identify the main presumptions about the overriding knowledge bases of SPI. In other words, what are the trusted premises of SPI? The main research question is expressed as openly as possible. Instead of referring to predefined SPI themes, we try to identify relevant topics by recognizing the primary goals of the studies. Then, within the material found, assumptions about the source of SPI knowledge and proposals for further research can be identified. In Table I, we present three research questions and their respective goals.

<table>
<thead>
<tr>
<th>Research Question</th>
<th>Goal</th>
</tr>
</thead>
<tbody>
<tr>
<td>RQ1: What kinds of practical challenges of SPI can be identified from the viewpoint of a software organization?</td>
<td>To illustrate the primary challenges which a software organization has to overcome. These challenges reflect potential current research topics.</td>
</tr>
<tr>
<td>RQ2: What presumptions about the sources of SPI knowledge can be identified?</td>
<td>To illustrate assumptions about the characteristics and bases of knowledge behind the SPI approaches.</td>
</tr>
<tr>
<td>RQ3: What kinds of SPI-related research subjects are suggested or what subjects can be implicitly derived?</td>
<td>To illustrate potential and current SPI research subjects.</td>
</tr>
</tbody>
</table>

RQ1 reflects the topics that are relevant from the viewpoint of software organizations to overcome SPI challenges. By identifying the primary goals of SPI research, we can illustrate what the practical challenges behind the research are. RQ2 concentrates on the knowledge sources of SPI, i.e., where the overall understanding about the means, goals, and resources of SPI initiatives comes from. Practical research subjects can be derived from practical challenges and therefore RQ1 is a forerunner of RQ3.

3.3 Search for studies

To retrieve relevant research papers, a search string was defined. The common term “software process improvement” is a generally used expression. We expanded the possibilities by simplifying the string to “process improvement”. “Software” can be left out by restricting the query to the field of science. If there are extra papers on the result set, they can be excluded manually. The term “process assessment” is used in papers which concentrate only on the assessment step. We used this as an alternative to “improvement”. The other component of the query is the size of the organization. We first assumed that if the acronyms VSE and SME are used in the abstract, they should also be
explained. The test query showed that this is not true in some cases. As a result, we also used the acronyms in the query.

The search string was formed as follows: ("process improvement" OR "process assessment") AND ("small*" OR "VSE" OR "SME"). The searches performed on the databases of Scopus, IEEE Xplore, and ACM on June 10 2017. The search targets were abstracts. The subject area was limited to computer science and engineering. Only papers published in conference proceedings or journals were retrieved. The year of publication was restricted to 2015 or later. After removing the duplicates, a total number of 54 studies were found. Mendeley reference manager software was used to detect duplicates.

3.4 Study selection

The selection of relevant studies was made by filtering the papers found by using inclusion and exclusion criteria. We considered the following two inclusion criteria:

- **IC1**: The study presents or discusses methods, approaches, standards, scenarios, research challenges, or opportunities of software process improvement (SPI).
- **IC2**: The study presents research in the context of very small entities (VSEs) or small-to-medium-sized enterprises (SMEs).

For those publications that passed the inclusion criteria, five exclusion criteria were applied to reduce the publications to only those that were directly aligned with the focus of the study. These criteria were:

- **EC1**: The paper is not directly related to software engineering.
- **EC2**: The paper is not directly related to process improvement.
- **EC3**: The paper is not a primary study, but uses other studies as a main source.
- **EC4**: The paper is a previous version of a more complete study about the same research.
- **EC5**: The paper is not written in English.

After using the inclusion and exclusion criteria, 26 articles from the 54 were left for deeper analysis and classification (See the APPENDIX)

3.5 Classification of papers

We used three groups of qualities to classify the papers:

- **C1**: Category of SPI approach.
- **C2**: Primary goal of the study.
- **C3**: Presumptions about the primary source of SPI knowledge.

C1 is a general category which has been used by other researchers. The remaining categories C2 and C3 were drawn up by analyzing the content of the papers.

Pettersson et al. divide **SPI approaches** (C1) into two categories: **prescriptive top-down** and **inductive bottom-up** approaches [Pettersson et al. 2008] [Gorschek 2006]. We refer to these categories as the **P category** (prescriptive) and the **I category** (inductive). The P category frameworks are based on recognized best practices, which have been documented as a standard model. Frameworks like CMMI, ISO/IEC 15504, and ISO/IEC 29110 belong to this category. There is an assumption that the more an organization’s processes matches the requirements of the model, the higher its level of maturity or process capability. In the I category methods, the focus of the first step is to understand organizations’ current situation (processes) and to determine the causes of significant problems. The improvement efforts are focused on the most critical issues from the organization point of view. Examples of the inductive approach are the Experience Factory (EF), the Quality Improvement Paradigm (QIP) [Basili 1995], and the retrospective meeting of Scrum [Schwaber and Beedle 2001]. Using an inductive approach, the basis for improvement comes from organizational experience and knowledge rather than a normative reference model. Inductive methods are adapted to the current state of software organizations [Pettersson et al. 2008]. The **category of SPI approach** was concluded by identifying the primary referential background of the framework. There were 14 articles on the P category and 10
articles were classified into the I category. Furthermore, two articles were identified as having a mixed approach (P/I).

The second category C2, the goal of each study reflects the research problems RQ1 and RQ3. We interpreted the goal of each study from the viewpoint of the software organization. This way we emphasize the practical value of the papers. This was carried out by first writing down the goal of the research in one sentence. Then, by using an iterative reading of both the article and the goal, we picked out the keywords which mostly reflected the practical interest of the software organization. We used the following criterion for the keyword: it should be possible to form a question using the words: “How to ... SPI”. For example, with the keyword “enable” we can ask “How to enable SPI in an organization?” Table II shows how the keywords are divided into P and I category approaches.

Table II. Challenges of SPI in VSEs and SMEs

<table>
<thead>
<tr>
<th>SPI approach</th>
<th>Primary goal / Challenge (How to... SPI?)</th>
</tr>
</thead>
<tbody>
<tr>
<td>P category</td>
<td>adopt, adopt/integrate, enable, integrate, self-assess, simplify/tailor, simplify/integrate</td>
</tr>
<tr>
<td>I category</td>
<td>enable, identify, integrate, learn, perceive</td>
</tr>
<tr>
<td>P/I category</td>
<td>adopt, model/tailor</td>
</tr>
</tbody>
</table>

After a careful revision, 10 keywords were identified: adopt, enable, identify, integrate, learn, model, perceive, simplify, self-assess, and tailor. There were four articles where we had to use two keywords instead of one to capture the main message. Two articles were classified as “simplify/tailor” because their message was “How to simplify a standard model and tailor it for practical purposes”. The other two-part keywords were “adopt/integrate,” “integrate/simplify,” and “model/tailor.”

The third category C3 is presumptions about the primary source of SPI knowledge. In other words, the knowledge base that is trusted as an overriding source of SPI expertise. This was identified by keywords that answer research question RQ2: What presumptions about the sources of SPI knowledge can be identified? These “SPI drivers” were categorized into SPI standard, people, organization, method (other than SPI standard), and multi-factor.

The overriding sources of SPI knowledge that were found are presented in Table III. The category of SPI standard consists of industry standards, which are used together (multi-model) or are somehow tailored or simplified to adapt to the context. The organization category refers to properties that refer to organization-specific factors. The category of people refers to human properties. The category of method consists of approaches other than standards. The research category collects success factors that are identified and reported by researchers, mostly in case studies.

Table III. Primary source of SPI knowledge

<table>
<thead>
<tr>
<th>Category</th>
<th>Primary source of knowledge</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPI standard</td>
<td>lightweight model, multi-model, standard, tailored model</td>
</tr>
<tr>
<td>Organization</td>
<td>business objectives, information flows, organization environment, organizational characteristics, organizational elements, organizational knowledge, practical needs, process modeling</td>
</tr>
<tr>
<td>People</td>
<td>experience, human factors, interpersonal skills, beliefs, tacit knowledge, professional knowledge, culture</td>
</tr>
<tr>
<td>Method</td>
<td>agile approach, lean approach, total quality management (TQM)</td>
</tr>
<tr>
<td>Research</td>
<td>success factors</td>
</tr>
</tbody>
</table>

The categories of the primary sources of SPI knowledge overlap. For example, success factors in the research category include factors from the other groups. In that regard, the categories are not theoretically valid; they are merely the subjects of overlapping groups.
4. RESULTS

The categories and data presented in the previous chapter are analyzed by making cross-reference tables between the categories. The results are collected here under the research questions.

RQ1: What kinds of practical challenges of SPI can be identified from the viewpoint of a software organization?
In Table II, the challenges are presented by grouping them according to the SPI approach. In the P category, there are challenges about how to enable or adopt standard-based SPI in the organization, and how to integrate different approaches. There are also the challenges of simplifying and tailoring a standard-based approach. In addition, self-assessment is mentioned. Also, in the I category approaches, the integration of different SPI methods is involved. Another similarity is the enabling factors of an SPI initiative. In addition, there are challenges about the identification of the subject to improve and challenges about learning during the SPI initiative.

RQ2: What presumptions about the sources of SPI knowledge can be identified?
We defined the primary source of SPI knowledge as referring to the knowledge bases that are trusted as an overriding source of SPI expertise. In Table III, these trusted premises of SPI are categorized. In a standard-based approach, a lightweight or tailored model, in addition to the standard itself, is seen as the main driver of SPI. In a multi-model approach, two or more standards are used to form a comprehensive and coherent framework. Each standard has its own purpose in the framework. In the organization category, eight organizational properties or components of an operations model are identified. The people category consists of seven human-centric properties. These human issues together with the organization category refer to the fact that the methods of anthropocentric sciences and organization study could offer information and additional possibilities for SPI research. The method category refers to approaches other than standard SPI methodologies. The three identified approaches are well known and accepted quality initiatives. The agile and lean are modern approaches that concentrate on the customer-oriented production philosophy. TQM is a management approach that forms an overall framework for quality measurement and improvement.

RQ3: What kinds of SPI-related research subjects are suggested or what subjects can be implicitly derived?
Research question RQ3 is an offshoot of RQ1. The following research question proposals are derived from the practical challenges facing software organizations. The list is organized by the category of approach (C1).

- Proposed research questions of P category approaches:
  o How to adopt a prescriptive SPI method?
  o How to adopt a prescriptive SPI method and integrate it with other methods used?
  o How to enable a standard-based SPI initiative?
  o How to integrate a prescriptive SPI method with other methods used?
  o How to carry out self-assessment?
  o How to simplify a prescriptive method and tailor it for a project’s needs?
  o How to simplify a prescriptive method and integrate it with other methods used?

- Proposed research questions of I category approaches:
  o How to enable an inductive SPI initiative?
  o How to identify the targets of improvement in inductive SPI?
  o How to integrate an inductive method with other methods used?
  o How to enable learning in an inductive SPI initiative?
  o How to perceive progress in inductive SPI?

- Proposed research questions of P/I category approaches:
  o How to adopt an SPI approach in a multi-model SPI initiative?
o How to model a process and tailor methods in a multi-model SPI initiative?

Challenges such as self-assessment, simplification, integration, and tailoring are interesting research subjects from the viewpoint of VSEs and SMEs. Most prescriptive approaches presume an outside evaluator or at least special expertise, which means extra resource costs. Self-assessment with a simplified and tailored model is one way to challenge the thresholds of SPI. The tailoring of models for specific needs and the integration of approaches brings flexibility, but at the same time there is a risk that special expertise will be needed. Two challenging subjects stand out in the I category: the identification of improvement targets and perceiving the progress of SPI. This might indicate that inductive methods lack methodological support for these steps of the SPI process. However, since our sample consists of just a few papers, reliable deductions cannot be made. SPI is not the playground of a single method or approach. A lot of interest has also been focused on multi-model approaches and integration.

5. CONCLUSION

Using a systematic mapping study and confining it to the last three years, 26 studies were selected for deeper analysis. After classifying and analyzing the results set, proposals for further studies were made by suggesting practically relevant research questions. Results indicate that research of software process improvement is still a robust discipline with plenty of possibilities. By integrating the results of RQ3 with the knowledge sources listed in RQ2, even more variety in research possibilities can be obtained.

This study gives a good starting point for further study. Firstly, the timeline of the search should be extended. The chronological scale of searched papers should be at least five years. Secondly, the size of the target organizations should be narrowed down more. Our study covers data concerning both SME and VSE software organizations. Other studies state that there are significant differences between these categories [Richardson and Cresse von Wangenheim 2007] [Sánchez-Gordón and O’Connor 2016]. Finally, our results show clearly that software process improvement is not just a pocket of software engineering. The challenges can be met by co-operating with other disciplines. For example, a comparative study of prescriptive and inductive approaches with other fields of engineering could be fruitful.

REFERENCES


European Commission. 2005. The New SME Definition - User guide and model declaration,


**APPENDIX**

Access to the list of papers included in the analysis: https://tutfi-my.sharepoint.com/personal/harri_keto_tut_fi/_layouts/15/guestaccess.aspx?docid=08278b0dc0a394e109103b8388f0dfb&authkey=ASlZjCClJcDsFjoBJ1dKYU
A Preliminary Empirical Exploration of Quality Measurement for JavaScript Solutions

DAVID KOSTANJEVEC, MAJA PUŠNIK, MARJAN HERIČKO, BOŠTJAN ŠUMAK,
University of Maribor
GORDANA RAKIĆ and ZORAN BUDIMAC, University of Novi Sad

Contrary to the increasing popularity of JavaScript programming language in the field of web application development, the numerical expression of evidence about the quality of solutions developed in this language is still not reliable. Based on the preliminary literature review, which is the main subject of this paper, this area has not yet been fully explored. Measurement is done by application of general and object-oriented metrics, which can reflect only general characteristics of the solution, while the specifics related to the programming language are not expressible by existing metrics. Due to the popularity of the language and the increasing number of JavaScript projects, the idea is to determine appropriate metrics and approach to measurement for their application in practice. Finally, the measurement approach will be implemented in the SSQSA Framework to enable its application. The preliminary research presented in this paper was conducted during a student course of Empirical research methods at the University of Maribor and therefore is limited in number of included papers, depth of research, and analysis of its contents, which restricts us to preliminary conclusion only, but places the foundation and justifies the described forthcoming research.

Categories and Subject Descriptors: H.0. [Information Systems]: General; D.2.8 [Software Engineering]: Metrics — Complexity measures; Product metrics; D.2.9. [Software Engineering]: Management — Software quality assurance (SQA)

General Terms: Software quality assurance

Additional Key Words and Phrases: software metrics, quality metrics, JavaScript, software analysis, JavaScript analysis

1 INTRODUCTION

JavaScript is an increasingly popular programming language that is dynamically interpreted and has a simple syntax. The Angel List Job Posting (USA) Agency recorded 30.6% jobs as JavaScript oriented in 2016 (Chen 2017). Popularity however brings a lot of changes and innovations in the development of web applications and services. JavaScript was primarily used only for client-side functionalities and was running in a browser. However, it is now running on the servers as well (Capan 2013). Applications and services are written using the JavaScript language and executed on the Node.js platform, a relatively new technology which is continually evolving. JavaScript is also becoming the main programming language for developing hybrid mobile applications using frameworks such as Cordova, Phonegap, Titanium, Facebook's React Native, etc. (ValueCoders 2017).

Today Node.js is one of the most innovative solutions for building servers and web/mobile applications. The field is growing rapidly with the valuable contributions of other developers and technological giants (Abhishek 2015). Node.js also provides a platform for publishing third party packages that are available through the Node Packet Manager (NPM) public repository. At the time of writing, there are more than 540,000 packages, which can be simply installed, managed, and updated through package management (B. Pfretzschner in L. B. Othmane 2016) and (E. Wittern, P. Suter 2016). Today, there are many solutions based on NodeJS framework that constantly evolving. The most well-known Node.js frameworks are Express.js, Hapi.js, Socket.io, Mojito, Mean.js, Sails.js, Koa and others (Noeticsunil 2017). Following all frameworks as well as their success in terms of quality is
very difficult, as new Node.js frames are constantly appearing (Abhishek 2015), (Noeticsunil 2017). Because of the large number and how fast different frameworks are being developed, and updated, it would be necessary to have appropriate tools enabling for analysis and comparison of these frameworks in terms of different quality dimensions. The analysis should include benchmarking criteria for (1) the framework, (2) what the framework is intended for, and (3) quality of the software developed on the framework. The quality of the developed projects would be tested by using JavaScript software metrics that are still not precisely defined.

The overall goal of this research is to establish and to apply an appropriate measurement to monitor quality of applications written using the JavaScript programming language. Upon the establishment of an measurement approach, the SSQSA Framework (Gordana Rakić 2015) could be exploited to measure JavaScript solutions according to it. Existing frameworks so far extract different metrics for measuring quality of solutions written in different languages, including JavaScript, but with no processing of extracted numbers that would provide the user with useful information about the quality of the analyzed solution. In order to employ an analytic technique to generate a certain information about the quality of a JavaScript solution, we have to define which of extracted metrics we will take into account and how they measure the quality attributes in JavaScript solutions as well as provide quantitative data about the quality variables or factors.

Following described goals, we conducted a preliminary research to identify software metrics potentially appropriate for assessing the quality of JavaScript solutions. The goal of this paper is to review approaches for measuring complexity and other quality attributes applied to JavaScript solutions, in order to select applicable metrics. Based on the literature review, we tried to identify metrics comparable to classic OO metrics that are, according to the existing literature, used to measure quality attributes of JavaScript solutions. In the following sections, we first describe the review approach. Next, results of the review are presented and discussed, where we focus on the selection of the metrics and evaluate their suitability to the specified purpose. Finally, the last two sections provide limitations of this preliminary research and conclusions with the plans for future work.

2 REVIEW APPROACH

Within the overall examination, several research questions will stand in the foreground as guidelines for further research:

1. Which existing software metrics are appropriate for measuring the quality of JavaScript solutions?
2. What are the shortcomings of the existing JavaScript metrics?
3. What JavaScript metrics already exists and what are their reference values?
4. Is it possible to extend the existing set of JavaScript metrics for comprehensive software quality measurement of JavaScript based solutions?
5. What tools for static analysis of JavaScript code already exists and which metrics do the support?

Guided by these research questions, preliminary research was focused on determination of the current situation in measuring the quality of JavaScript solutions. Specific research questions that should be answered by this preliminary literature review are:

1. Which software metrics have been used in existing literature for measuring the quality of JavaScript solutions?
2. Are these metrics appropriate to measure the quality of JavaScript solutions?

Following databases of scientific papers were used for searching of relevant literature: (1) Web of Science, (2) Science Direct and (3) IEEE Xplore Digital Library. The search was focused on research published in the past 5 years (from 2012 to 2017) in the field of computer science and informatics. The
used keywords were “JavaScript Metric”, “static analysis”, “JavaScript analyse” and “software analyse”. Table I presents combination of keyword sequence for each database and number of results (conducted at the end of May 2017).

Table I. Set of keyword combinations

<table>
<thead>
<tr>
<th>Database</th>
<th>A combination of a keyword sequence</th>
<th># of results</th>
</tr>
</thead>
<tbody>
<tr>
<td>IEEE Xplorer</td>
<td>(javascript) OR Javascript AND ((metric)OR (static analysis*) OR (software analy*)OR (“Abstract”:staticanalys*))</td>
<td>28</td>
</tr>
<tr>
<td>ScienceDirect</td>
<td>(TITLE-ABSTR-KEY(javascript)) AND (TITLE-ABSTR-KEY(metric) OR TITLE-ABSTR-KEY(static analy*) OR TITLE-ABSTR-KEY/software analy*)</td>
<td>27</td>
</tr>
<tr>
<td>Web of Science</td>
<td>TS=(javascript AND metric*) OR SU=(static analy* OR software analy*)</td>
<td>38</td>
</tr>
</tbody>
</table>

13 papers were found and included in this preliminary research, addressing listed general questions, without a specific selection criterion. However, the preliminary research in this paper is not a systematic literature review and focuses on discovery of software metrics applied on JavaScript solutions and exploration of their usability. Other research questions will be addressed in more depth in future work.

3 REVIEW RESULTS

Authors in (Y. Ko, H. Lee, J. Dolby 2015) present a new approach for analyzing large JavaScript applications with static analysis. The research describes their tool and focuses on the analysis of products written in the JavaScript programming language.

The authors in (S. Mirshokraie, A. Mesbah 2013) focused on the JavaScript programming language and proposed a set of mutations that are specific to web applications. They suggest a technique that complements the static and dynamic analysis of the program to guide the process of generating mutations, on sections of the program code, where there is a greater likelihood of errors or could affect the output of the program. The paper presents the MUTANDIS tool and gives an assessment of effectiveness (S. Mirshokraie, A. Mesbah 2013).

Similarly, the paper (S. Rostami, L. Eshkevari, D. Mazinanian 2016) presents the JSDeodorant tool, a plug-in for Eclipse. The tool allows us to observe classes in JavaScript, it can identify the divergence between modules and utilities when examining objects in a program code. The main purpose of the paper is the presentation of the techniques, provided by the JSDeodorant tool, the comparison of the tool with the more familiar tool JClassFinder, and the quantitative and qualitative evaluation of the results to recognize their limitations and possibilities for future improvements (S. Rostami, L. Eshkevari, D. Mazinanian 2016). The authors in (Mesbah 2013) present the JSNOSE tool that uses the technique of detecting bad programming patterns. The tool compares the program code with a set of 13 samples of JavaScript to find "smelly" parts of the code. Parts of bad code can have a poor effect on the whole project, maintenance or understanding. The JSNOSE tool in the paper is also tested on eleven web applications (Mesbah 2013). By increasing the use of JavaScript frameworks for web applications, there is an increasing demand for the quality of the written code including fast maintenance, reliability and speed.

Based on the preliminary literature review, we concluded that there are already several metrics defined, mostly connected to object-oriented paradigm and all of them are not completely suitable for JavaScript program code. However, JavaScript metrics were included in a research conducted by (Alberto S. Nuñez-Varela, Héctor G. Pérez-Gonzalez, Francisco E. Martínez-Perez 2017), where a total of 190 different metrics was identified for the object-oriented paradigm. The most common metrics and total occurrences of specific metric, are presented in the Table II for programming languages Java, AspectJ, C++, C, C#, Jak, Ada, COBOL, Pharo, PHP, Python, Ruby, as well as for JavaScript.
Table II. Commonly applied metrics

<table>
<thead>
<tr>
<th>Metric</th>
<th>Total occurrences</th>
</tr>
</thead>
<tbody>
<tr>
<td>Weighted Methods per Class (WMC)</td>
<td>89</td>
</tr>
<tr>
<td>Coupling Between Objects (CBO)</td>
<td>89</td>
</tr>
<tr>
<td>Lack of Cohesion in Methods (LCOM)</td>
<td>86</td>
</tr>
<tr>
<td>Depth of Inheritance Tree (DIT)</td>
<td>81</td>
</tr>
<tr>
<td>Lines of Code (LOC)</td>
<td>79</td>
</tr>
<tr>
<td>Number of Children (NOC)</td>
<td>77</td>
</tr>
<tr>
<td>Response for a Class (RFC)</td>
<td>72</td>
</tr>
<tr>
<td>Number of Methods (NOM)</td>
<td>57</td>
</tr>
<tr>
<td>Cyclomatic Complexity (V(G))</td>
<td>55</td>
</tr>
<tr>
<td>Number of Attributes (NOA)</td>
<td>43</td>
</tr>
<tr>
<td>Fan-out (FANOUT)</td>
<td>27</td>
</tr>
<tr>
<td>Fan-in (FANIN), Number of Public Methods (NOMP)</td>
<td>22</td>
</tr>
<tr>
<td>Lines of Comments (LCOMM)</td>
<td>21</td>
</tr>
<tr>
<td>Afferent Couplings (Ca), Efferent Couplings (Co)</td>
<td>20</td>
</tr>
<tr>
<td>Lack of Cohesion in Methods 2 (LCOM2)</td>
<td>18</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Metric</th>
<th>Total occurrences</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lines of Source Code (SLOC)</td>
<td>17</td>
</tr>
<tr>
<td>Lack of Cohesion in Methods 3 (LCOM3)</td>
<td>16</td>
</tr>
<tr>
<td>Cohesion Among Methods (CAM), Number of Classes (NCLASS), Number of Parameters (NPAR)</td>
<td>15</td>
</tr>
<tr>
<td>Nesting level (NEST), Message Passage Coupling (MPC)</td>
<td>14</td>
</tr>
<tr>
<td>Number of Overridden Methods (NMO), Number of Public Attributes (NOPA)</td>
<td>13</td>
</tr>
<tr>
<td>Lack of Cohesion in Methods 4 (LCOM4)</td>
<td>12</td>
</tr>
<tr>
<td>Effort (E)</td>
<td>11</td>
</tr>
<tr>
<td>Instability (I)</td>
<td>11</td>
</tr>
<tr>
<td>Lack of Cohesion in Methods 5 (LCOM5)</td>
<td>11</td>
</tr>
<tr>
<td>Tight Class Cohesion (TCC)</td>
<td>11</td>
</tr>
<tr>
<td>Abstractness (A)</td>
<td>10</td>
</tr>
<tr>
<td>Loose Class Cohesion (LCC), Number of Statements (STAT)</td>
<td>11</td>
</tr>
<tr>
<td>Number of Methods Inherited (NMI)</td>
<td>11</td>
</tr>
<tr>
<td>Attribute Hiding Factor (AHF), Class Cohesion (CC), Data Abstraction Coupling (DAC), Data Access Metric (DAM), Method of Aggregation (MOA), Method Hiding Factor (MHF), Normalized Distance from Main Sequence (Dn)</td>
<td>Less than 10</td>
</tr>
</tbody>
</table>

All listed metrics (Table II) however cannot be applied to JavaScript program code. According to the research (L. H. Silva, D. Hovadick, M. T. Valente, A. Bergel, N. Anquetil 2016), which presents the JSClassFinder tool, only few metrics are suitable. The mentioned tool creates a structure model from the program code and is object-oriented, enabling visualization of the code (UML class diagram) and presenting information about classes, methods, attributes, deductions, and relationships. Metrics obtained with the tool and suitable for JavaScript measurement according to (L. H. Silva, D. Hovadick, M. T. Valente, A. Bergel, N. Anquetil 2016) are the following:
1. Number of classes,
2. Number of method,
3. Number of attributes,
4. Number of subclasses,
5. Depth of inheritance tree.

The selected five metrics are not enough for comprehensive measuring of JavaScript solution quality or complexity; therefore, extended research will be conducted, focusing on additional research questions.

4 LIMITATIONS AND THREATS TO VALIDITY

This research has limitations that have to be identified and discussed. Since the research was conducted by students within the course Empirical research methods it is not complete and other papers of existing research must be taken into consideration. Furthermore, this is only a preliminary
research from the empirical perspective. After comprehensive empirical research, confirmed conclusions have to be challenged in practical application to examine real usability of selected metrics and discover their weaknesses. Final conclusions should lead to answers to the research questions.

5 CONCLUSION AND FUTURE WORK

Several metrics have been identified and are already commonly used for software quality measurement; however most of them are reportedly not suitable for JavaScript source code measurements. A set of metrics dedicated to measuring the quality of JavaScript solutions must be defined and evaluated. The preliminary research in this paper is an initial effort to examine the field of JavaScript metrics, providing basic insight into the research field.

The future work will include a systematic literature review of the field, practical examination of selected metrics and tools and definition of metrics suitable for JavaScript program code measurement in terms of quality and complexity according to defined research questions. Extended formation of appropriate measurements for JavaScript source code will be implemented by integrating them with the SSQSA Framework.

6 ACKNOWLEDGMENTS

This joint work is enabled by bilateral project “Multidimensional quality control for e-business applications” between Serbia and Slovenia (2016-2017). Furthermore, the two authors from University of Novi Sad were partially supported by the Ministry of Education, Science, and Technological development, Republic of Serbia, through project no. OI 174023. The authors acknowledge the financial support from the Slovenian Research Agency (research core funding No. J5-8230).

7 REFERENCES


Blockchain Implementation Quality Challenges: A Literature Review

BOJANA KOTESKA, ELENA KARAFILOSKI and ANASTAS MISHEV, University SS. Cyril and Methodius, Faculty of Computer Science and Engineering, Skopje, Macedonia

Blockchain is a public digital and distributed database solution providing decentralized management of transaction data. Since the introduction of Bitcoin cryptocurrency, which was the first implementation of the Blockchain technology in 2008, the interest in Blockchain technology has been constantly increasing. Blockchain is not applicable only in financial transaction systems but it is transforming our society from the way we use our cars, smartphones, healthcare, vote, and even personal identification. As such when we discuss the Blockchain implementation we are really discussing three different things: Blockchain ledger, Blockchain network and clients. Modern Blockchain implementations have to adapt to some technical challenges and limitations required for Blockchain technology. For example, security, privacy, throughput, size and bandwidth, performance, usability, data integrity and scalability are just some of the attributes required for high quality Blockchain implementation. In this paper, we aim to analyze the current quality issues in the Blockchain implementation and to identify the Blockchain quality attributes. A literature review is conducted to investigate the current quality requirements for Blockchain implementation. Findings show that the research on quality requirements for Blockchain implementation is still in its early stage. The conclusions of this study could be used for further analysis of the quality attributes required for the Blockchain implementations and improvement of the quality of the Blockchain systems.

Categories and Subject Descriptors: D.2.4 [Software Engineering]: Software/Program Verification — Validation; K.6.4 [System Management] Quality Assurance

General Terms: Reliability, Security

Additional Key Words and Phrases: Blockchain, quality requirements, software quality.

1. INTRODUCTION

Blockchain is a technology that redefines trust in the new generation systems. It spreads the idea of processing any type of transaction without a mediator. Mediators, like corporations and governments, almost always come as central entities that receive, process and store the transactions. All the trust we put as users in any system is trust in the mediators who are obliged to process the transactions using correct business logic. Mediators are fully in control of data security and data privacy too. In a case of Blockchain systems, the trust is decentralized. Users just need to trust the system and the smart code that is shared between all the participants. From technical point of view, Blockchain is a distributed database that exists on a P2P network (Fig. 1). This P2P network is a backbone of the system because every node in the network is on the same level as all the other nodes. Although nodes can come in many forms, there is no central node that is an authority. Every node stores a local copy of the Blockchain. If consensus of nodes agrees upon transaction’s validity, then the transaction is considered valid [Pilkington 2015].

The Blockchain database can be a simple file that just stores minimum required data about the encrypted transactions. All the transactions are grouped in timestamped blocks (Fig.2). Once the transaction is in the block, it is

Author’s address: Bojana Koteska, FCSE, Rugjer Boshkovikj 16, P.O. Box 393 1000, Skopje; email: bojana.koteska@finki.ukim.mk; Elena Karafiloski, FCSE, Rugjer Boshkovikj 16, P.O. Box 393 1000, Skopje; email: elena.karafiloski@gmail.com; Anastas Mishev, FCSE, Rugjer Boshkovikj 16, P.O. Box 393 1000, Skopje; email: anastas.mishev@finki.ukim.mk.

Copyright © by the paper’s authors. Copying permitted only for private and academic purposes.
irreversible. All the nodes can access the transactions, but cannot change or delete a transaction from the Blockchain. The business logic by which the nodes operate is often defined in so called "smart contracts". Smart contracts specify all the conditions that must be met before a transaction is executed. They come together as programmed rules for writing and reading from the Blockchain database. The smart contracts are installed on every node too.

The Blockchain is constantly growing. In a case of Bitcoin, there is new block every ten minutes. The last block in the Blockchain contains the latest executed transactions. Every block has a header and a body. The header contains metadata like time of block’s creation and a link to the previous block. That is how the blocks are linked. All the transactions with the addresses of the parties included in the transactions are listed in the block’s body.

There are public and private Blockchains. Bitcoin is a public Blockchain because it was designed to be completely open, decentralized, and permissionless. This means that anyone can participate without establishing an identity and there is no central authority that controls admission. New Blockchain platforms like Hyperledger [News 2017] take a novel approach to the model, in part by managing the admission of participants. In other words, Hyperledger is a permissioned, shared ledger modified to respond to the multitude of industrial use case requirements by providing a secure model for identity. Here, new techniques are replacing the need of block’s mining. Because the environment is more controlled and contains smaller number of nodes which are authorized, mining can be avoided and the consensus can be simpler by rule.

All nodes in the network can have a permission to create new transactions. Depending on the application, one end user may be one node, or may connect to one server node using web application. In both cases, the end user creates
new transactions. When a transaction is created, it has to go through validation and confirmation stages before it enters
the Blockchain and it is broadcasted to the network. P2P nodes share the transaction between themselves almost in
a real time. Every node receives the transaction and validates its structure and actions. The validation checklists are
defined in the Blockchain system. They may check the transaction’s parameters, size, values etc. If valid, the node
saves the transaction into its transaction pool. If not, the transaction is immediately removed from the pool.

Some of the nodes are so called “miners”. They use special hardware to work on creating the next block. Miners
take all the available transactions from the transaction pool and include them in a new candidate block. Mining a block
in Bitcoin is done with a proof of work concept [Vukolic 2015]. That is calculating a random hash value using data of
the candidate block. To guess a correct hash value there is a need of great processing power that calculates millions of
values in a second. The correct hash value must satisfy a defined difficulty target. This number is calculated using all
block’s metadata including the hash of the previous block. This is the key to Blockchain security. If someone tries to
change a transaction from the past, the hash value of the block that contains the transaction must be calculated again.
All hash values for the blocks that came afterwards must be calculated again too. This is not feasible, unless more
than half of the nodes in a network are malicious. When a new block is created, it is broadcasted to the network. All
nodes receive the block, validate it and all the transactions in it. If valid, all nodes put it as the next block in their local
Blockchain. Transactions that are included in the created block, are then removed from the pool.

The research has shown that most of the studies have focused on one or several particular aspects connected to
Blockchain implementation. To the best of our knowledge, no studies have been investigated in details the quality
requirements and solutions for Blockchain implementations. This type of study can help to identify quality requirements
that can be taken into account in the new Blockchain implementations. Furthermore, these study could be helpful to
understand how different Blockchain characteristics and current issues could affect the quality of Blockchain systems.

In this research, we have conducted a literature review to investigate the level of evidence available in published
literature on the quality requirements for Blockchain implementations.

The paper is organized as follows. In Section 2, we research the current quality issues in the Blockchains imple-
mentations. Section 3 provides the overview of the quality requirements and solutions. Conclusive remarks are given
in the last section, Section 4.

2. CURRENT QUALITY ISSUES IN BLOCKCHAIN IMPLEMENTATIONS

Blockchain technology is still in testing phase and its implementation has some issues that have to be addressed when
trying to establish Blockchain free cryptographically secured system. In this section, we summarize the most common
Blockchain quality issues found in this research.

Throughput issues. In his book, Swan [Swan 2015] has discussed seven technical future challenges and limitations
of the Blockchain technology. One of them is throughput. The problem with current Bitcoin network is that it processes
3 to 20 transaction per second (tps) [Xu et al. 2016], with a maximum possible theoretical throughput seven tps.
In comparison, VISA transaction network can process 2000 tps (has been stress-tested in 2013 to handle 47,000
transactions per second [Trillo 2013]) and Twitter network 5000 tps.

Latency issues. Time factor is one of the most critical issues in Blockchain implementations. Having the requests
processed on Internet almost immediately, it is an obstacle in regards to the universal technology acceptance, [Beck
et al. 2016]. In order to provide security the Bitcoin transaction block, the time needed to complete one transaction
is about 10 minutes. For a larger transfer amounts, the cost of a double spend attack can last about an hour. VISA
transaction completion process takes seconds at most [Swan 2015] [Yli-Huumo et al. 2016].

Size and bandwidth issues. In February 2016, the size of the Blockchain in the Bitcoin network was over 50,000
MB and it could grow 214 PB in each year. Current Bitcoin size is 1MB and a new block is created every 10 minutes.
It means that an average 500 transaction can be handled in one block. This is a serious issue if Blockchain needs to
handle more transactions [Swan 2015] [Yli-Huumo et al. 2016].

Scalability issues. One of the main challenging problems in Blockchain implementations is scalability. In order
to provide the theoretically proved security, the Blockchain implementation must have a large number of full nodes.
Otherwise, the implementation might result in a less decentralized system such as in a case with Bitcoin [Beck et al. 2016]. The scalability limits of the Blockchain are connected to the size of the data on Blockchain, the transaction processing rate, and the latency of data transmission. On the other hand, the latency between the transaction submission and confirmation is affected by the consensus protocol. For example, the time between the transaction submission and confirmation on Bitcoin is around 1 hour (10-minute block interval per block and 6-block confirmations), and around 3 minutes on Ethereum (14-second block interval per block and 12-block confirmation) [Xu et al. 2017].

**Cost issues.** Blockchain use is not free of cost which is a drawback of decentralization and the blockchain technology. The users have to pay for the transactions and computational power. One fact that users will chose centralized solutions is because they will not be constantly reminded that an action has a fee, but the prices will be more hidden [Beck et al. 2016].

**Data malleability issues.** Data malleability is a potential issue in the Blockchain implementation. The signatures do not provide guarantee the ownership of the Bitcoin transferred in a transaction. An attacker can modify and rebroadcasts a transaction which can cause problems with approval of transaction confirmation. [Decker and Wattenhofer 2014] [Yli-Huumo et al. 2016].

**Authentication issues.** Another problem connected to the Blockchain transactions is the authentication. An example of incident with the authentication is the well-known case in Mt.Gox [Bos et al. 2014] when the storage of customer private keys was attacked and stolen.

**Privacy issues.** One issue connected to Blockchain privacy is the problem with multiple addresses. For example, users of the Bitcoin system can create any number of addresses and researchers try to cluster all addresses that belong to the same user [Herrera-Joancomartì and Pérez-Solà 2016]. Address clustering is performed to trace the economic movements of the same users. The goal is to find all addresses included in the transaction that belong to the same user [Reid and Harrigan 2013] [Androulaki et al. 2013]. In [Koshy et al. 2014], the authors found that some of the Bitcoin addresses can be mapped to IP addresses by analyzing the transaction traffic.

**Double-spending attacks.** Blockchain implementations are susceptible to double-spending attacks. In a case of Bitcoin, a double-spending attack can occur when the attacker keeps his/her bitcoin while receiving services that can be spent again. This happens when the attacker credits an account, receives the service or goods by the account holder and then reorganize the ledger by reversing the transaction that credits the account. Nakamoto [Nakamoto 2008] claims that Bitcoin system is preventing double-spending attacks by modeling the attacker and the set of honest players as to competing actors performing a random walk moving toward a single direction with probabilistic steps. However, it is not claimed that in Bitcoin’s decentralized environment the attacker may attempt to introduce disagreement between the honest miners [Garay et al. 2015].

**Security issues.** A problem with the public distributed ledgers is the highly speculative nature with a trade-off between the dimension of the network and decentralization [Atzori 2015]. The Bitcoin Blockchain has a possibility of 51% attack. In such a case, one miner can have full control of the majority of the network which is a serious problem. In [Lim et al. 2014], the authors have identified different security breaches that occurred in Bitcoin including DDoS attacks, account hacking using Trojan horses and viruses from ads. According to [Vasek and Moore 2015], a total of $11 million had been contributed to scams by 13000 Bitcoin victims from September 2013 to September 2014.

**Wasted Resources.** The energy spend of mining in the Bitcoin network is approximately $15 million per day [Swan 2015]. The waste in the Bitcoin is a result on the Proof-of-Work effort. In such a case, the probability of mining a block depends on the work done by the miner [Yli-Huumo et al. 2016].

**Usability issues.** The Bitcoin API is less user-friendly that the other modern API-s [Swan 2015] [Yli-Huumo et al. 2016].

**Versioning, hard forks, multiple chains.** Smaller chains with smaller number of nodes are more susceptible to 51% attack. Another issue is the hard merging of split chains for administrative or versioning purposes [Swan 2015] [Yli-Huumo et al. 2016].
3. QUALITY REQUIREMENTS AND SOLUTIONS FOR BLOCKCHAIN IMPLEMENTATIONS

To ensure the trustworthiness of a Blockchain system, the Blockchain implementation must provide high data integrity, security, reliability and node privacy [Porru et al. 2017] [Swan 2015].

A possible solution to the throughput problem is to increase the size of each block, but it can lead to other size problems [Yli-Huumo et al. 2016]. If the same Blockchain is used on a wide variety of applications, more performance is needed [Beck et al. 2016].

An alternative scenario to solve the problem with size and bandwidth is to reduce the number of submitted transactions and to stop recording any voting activity, but to record only the different value of negotiable variables and the final voting result [Xu et al. 2016].

A solution to the scalability problem is to have many blockchains for different purposes. For example, some of them can be used for specific purposes while other for generalized tasks. The benefit of this approach is that the Blockchains can use each other to provide security for one another, no matter what their purpose is. In a such case, a miner can mine Blockchains with a suitable size and also security will be on a satisfactory level [Beck et al. 2016].

One option to solve the authentication issue is the BlueWallet device which is a Bitcoin hardware token which secure and sign Bitcoin transactions and communicates by using Bluetooth Low Energy [Bamert et al. 2014]. Another solution to this problem is the proposed certification system for Bitcoin which offers a guarantee to send and receive Bitcoins only to/from certified users, and control of the creation of Bitcoins addresses by trusted authorities [Ateniese et al. 2014]. In [Mann and Loebenberger 2017], the authors proposed a two-factor authentication for a Bitcoin wallet by using a smart phone as the second authentication factor.

The privacy of the Blockchain transaction is an essential quality requirement in the Blockchain implementations. Transactions are public and all the data on the Blockchain are visible to all participants, but they must not be linked to identities. A more appropriate solution is a permissioned Blockchain which will allow developers to grant permissions to the participants [Xu et al. 2016]. If Blockchain systems are to operate at a global scale, a new model for privacy-preserving identities is needed. The model must allow entities to verify the security of an identity, to access the independence of an identity from any given authority and to access the source of trust for a digital identity. MIT researches proposed the ChainAnchor system which is designed for permissioned Blockchains and it adds an identity and privacy-preserving layer above the Blockchain [Shrier et al. 2016]. A human resource information management model based on Blockchain that reduces the risk of authenticity of human resource information is presented in [Wang et al. 2017]. This model provides authentic and decision support information to the human resource management in an organization. Consensus mechanism, smart contract, accounting, and payment functions of the Blockchain provides the basic support for human resource information management.

A new scheme for increasing the energy efficiency is presented in [Paul et al. 2014]. Each block of the Blockchain is modified by adding some extra bytes to utilize the timestamp more effectively. An alternative to Proof-of-Work effort is Proof-of-Stake in which the resource that is compared is the amount of Bitcoin a miner holds. If someone is holding 1% of the Bitcoin, he can mine 1% of the "Proof-of-Stake blocks" [Yli-Huumo et al. 2016].

One of the biggest challenges is the data storage capacity limitation in the current Blockchain implementations. Taking into advance the cost efficiency, performance, and flexibility, the real design challenge is to decide what data and computation should be placed on-chain and what data should be stored off-chain [Xu et al. 2016]. A common practice for storing data in the Blockchain ledger is to store raw data off-chain, and to store meta-data, small critical data, and hashes of the raw data only on-chain [Xu et al. 2017].

Security and reliability of the software that implements Blockchain technology can be ensured by continuous testing techniques. Such testing techniques that could improve the quality of Blockchain systems were recently presented by IBM [Ojha 2017].

The testing of the Blockchain implementations takes into consideration several implementation aspects. First, the level of validation depends on the platform implementation (public or private). If the platform is private, then it requires much more effort in testing. Next, if the test platform does not provide replica of the Blockchain implementation, then
additional time will be required for setting up the environment. If the Blockchain implementation has connection with other applications, then the consistency must be checked by making integration testing. One of the key issues is performance of the Blockchain implementation. A strategy that will handle a large number of transactions must be applied, so the results from the performance testing are satisfactory. In a case of a real scenario testing, the early involvement of the testing team is very important since the other system components or applications can be affected by the Blockchain implementation. An example of the Blockchain testing process is shown in Fig. 3. The system appreciation phase includes the creation of testing plan and strategy that provides detailed view of the impacted components in the system. In the test design assurance phase, the testing team should create a detailed level test strategy and to assign traceability to the requirements. The key components that need to be done in this phase are: building a model of the blocks structure, transactions and contracts, defining use cases for each section and validation end points, specifying non-functional requirements and security testing needs. The result from the test planning phase should be a full test strategy with a specified testing methodology. In order to verify and validate the test plan, the use cases should be mapped to detailed level test cases. In the last phase, the test cases execution and result verification are performed. The execution of test cases can be automated by using frameworks for unit testing. Testing should be done according the test plan and the list of defects needs to be reported along with the test execution status [Sundarraman 2017].

Software architecture that includes specific design notations, patterns and meta models and also modeling languages that provide designing graph models (UML diagrams) can enhance the quality of software that implements the Blockchain technology. Additionally, specific metrics required to measure complexity, communication capability, resource consumption and performance could be defined and used for evaluating and improvement of the software process [Porru et al. 2017].

If the Blockchain implementations are not tested properly they will likely fail at some point. There are different ways to verify a distributed system such as Blockchain. In [McCaffrey 2015], the authors propose formal verification methods and lineage driven fault injection (random, heuristic...).

The evaluation of the Blockchain performance can be done using a few simple qualitative and quantitative metrics [Kakavand and Kost De Sevres 2016]:

—Submission Throughput: maximum number of transactions submitted per second permitted by each node and by the network.
—Maximum/Average Validation Throughput (processing speed) of the network: parameter that determines the maximum/average number of transactions validated per second permitted by the network.
—Average Transaction Validation Latency: the average time taken to validate the transaction from the time of its submission. This metric measures the average time of waiting of the users for their transaction to be validated.
—Latency Volatility: measure of the variety of the transaction processing time.
—Security: The evaluation of system security requires a threat model which is able to define the types and scope of adversaries and attacks on the system. The following analysis are required to perform the security evaluation of the Blockchain implementation: Transaction and block immutability, transaction censorship resistance, denial of Service (DoS) resilience, trust requirements of users and oracles, protocol governance and node membership services and transaction confidentiality and user anonymity.

—Confidentiality: ability of Blockchain nodes to hide the contents of the transaction and or the identity as having participated in that transaction.

—Transaction fees: Users pay small transaction fee to the network in order to process transactions or execute smart contracts. These fees are used to cover the maintenance costs or protection of malicious computational tasks.

—Hardware requirements: Memory/storage per node, processor resources needed to validate transactions and blocks, network usage.

—Scalability: Number of nodes, transactions, users, geographic dispersion.

—Complexity: measures the development, maintenance, and operation complexity of Blockchain infrastructure.

—Smart-contract limitations: limitations of the code deployed to the Blockchain caused by smart contract scripting language and the underlying consensus protocols.

The Blockchain implementation into an established supply chain system is not an easy process because the existing supply chains are hard to be changed and adapted. Some companies spend years putting supply chains into the existing systems [Mougayar 2016].

4. CONCLUSION

Blockchain is a distributed database solution providing decentralized transaction and data management. Modern Blockchain implementations have to ensure security, privacy, throughput, size and bandwidth, performance, usability, data integrity and scalability. However, these quality attributes set up a lot of challenges that need to be addressed.

To understand the current research in the field of Blockchain implementation quality, we have analyzed the current quality issues in the Blockchain implementation and identified the Blockchain quality attributes. The research has shown that this topic is still immature. The results indicate that the Blockchain implementations need to be improved in terms of scalability, latency, throughput, cost-effectiveness, authentication, privacy, security, etc. The goal of this study was to investigate the quality requirements and solutions for Blockchain implementation. We provided an overview of the quality recommendations and solutions for Blockchain that could improve the quality of the new Blockchain implementations. This study contributes to the theory by analyzing the Blockchain quality from literature and by providing an integrated view of quality requirements for Blockchain implementations.

REFERENCES


8:8 • Bojana Koteska et al


Barriers and Facilitators of Digitalization in Organizations

MARKKU KUUSISTO, Tampere University of Technology

Digitalization has transformed industries and societies in profound manner. Some changes have been documented as increased productivity and reduced costs of communications and information processing, to name a few. However, new technologies are not always fully embraced by organizations. This paper delves into barriers and facilitators surrounding adoption of digital technologies. The study was done as a systematic literature review. The paper introduces the most common technology acceptance models. Even while not being directly facilitators or inhibitors of use of new digital assets in organization, these models help to explain how the technology is accepted in organizations. The paper also introduces main inhibitors of digitalization and use of novel digital solutions in organizations. Moreover, the most significant facilitating factors are presented in the paper.

General Terms: Digitalization, Organizations
Additional Key Words and Phrases: Technology Acceptance

1 INTRODUCTION

Digitalization is a hype word of the decade. Not by accident, when thinking about how much digital technologies have altered the way we live and function. The profits and alterations caused by digitalization have been studied and shown convincingly. Even if the effects are clear, the ways to adapt to digitalization and the organizational prerequisites for adaptation are not thoroughly studied. It is thus an interesting topic. In this article, a literature review of the barriers and facilitators of digitalization in organizations is presented. In this article digitalization is defined as the automation of tasks accomplished by digital technologies like business information systems and as the change in the way of working.

There are a few studies regarding the inhibiting effects of organizational inertia and incumbent systems [Haag 2004, Polites and Karahanna 2012], and large body of research on how innovations diffuse and are adopted in companies [Jeyarajaj et al. 2006, Jones et al. 2010, Yao et al. 2009, Scupola 2012]. Main theoretical models featured in these studies are technological acceptance model (TAM), the unified theory of acceptance and usage of technology (UTAUT) and technology-organization-environment framework (TOE).

Each one is further explained in their own sections. In addition, there is small stream of extant literature on drivers and barriers of IT adoption in organizations. Beyond these points, however, no literature was found with the applied methodology. Managerial perspective on how to facilitate digitalization seems to be almost white area in the map – even though many studies suggest top management support is a key issue in IT adoption and diffusion.

This literature review was inductively created beginning with keywords “digitalization” “barrier” and “facilitator” mixed in different combinations in scopus, andor and google scholar. After selecting articles back- and forward reference searches were conducted on each article to attain fuller picture of the phenomenon. Articles were selected on same basis as the ones in first round from the search engines. This reference search was iteratively repeated for every new article selected this way until no new articles surfaced. Only articles from peer-reviewed journals or conferences were selected to provide rigor for the research.

Author's address: M. Kuusisto, Pohjoisranta 11, P.O. Box 311, 28100 Pori Finland; email: markku.kuusisto@tut.fi

Copyright © by the paper’s authors. Copying permitted only for private and academic purposes.
The article proceeds as follows: Organizational Inertia is discussed in section two. Section three is about TAM. UTAUT is presented in section four. Fifth section brings forth TOE-model. Facilitators of Digitalization are discussed in section six and finally Information Systems strategic alignment is addressed in section seven. Section eight concludes the article.

2 ORGANIZATIONAL INERTIA

The Organizational inertia is seen as a barrier for adopting digitalization in organizations. Polites and Karahanna [2012] define inertia as: “inertia in an IS context as user attachment to, and persistence in, using an incumbent system (i.e., the status quo), even if there are better alternatives or incentives to change.” Haag [2004] further conceptualizes organizational inertia to have five sub-dimensions. These are cognitive, behavioral, socio-cognitive, economic and political aspects. Cognitive dimension refers to managerial tendency of using incumbent systems even while knowing there are better alternatives available. Key manager having much resistance to new systems can easily hold back the whole organization.

Behavioral inertia is the tendency to keep doing things in certain way, just because they have always been done that way. Socio-cognitive dimension consists of change-inhibiting culture in company making changes hard to implement. Economic inertia entails both sunk costs in legacy systems as well as costs of adopting the new system. Political inertia refers to environmental reasons – partners and customers holding back the adoption of new innovation as it would affect them as well [Haag 2004]. Polites and Karahanna [2012] find support for their claim that individual working habits lead to organizational inertia. Habits can be considered to be a good thing since carrying out habitual tasks requires less concentration and leaves the employee’s mind available to think other tasks while shortening decision times. However, in the context of adopting new systems or advancing digitalization, habitual working methods need to be broken in order to advance with the new way of working.

3 TECHNOLOGY ACCEPTANCE MODEL

The Technology acceptance model (TAM) has been widely used in organizational studies. Gangwar et al. [2013] consider it being the dominant model for explaining technology adoption at all organization levels and at individual level. It was adapted from the theory of reasoned action. Since it has been used extensively, it has developed some advantages such as well-researched and validated inventory of psychometric measurements [Gangwar et al 2013].

TAM assumes that the more accepting users are to use a new system, the more likely they are to use time and effort on learning and adopting the new system over the old one [Jones et al. 2010]. TAM conceptualizes two key antecedents for adoption of new system. First one is perceived ease of usage. Perceived ease of usage is defined as “the degree to which the prospective user expects the target system to be free of effort”. This is rather intuitional – the easier a new system is to use, the happier persons are to adopt it. Another antecedent is perceived usefulness. Its definition is as follows: “the prospective user’s subjective probability that using a specific application system will increase his or her job performance within an organizational context” [Gangwar et al 2013]. Perceived ease of use affects the perceived usefulness as well as the attitude of user.

These perceived notions of the technology to be adapted form individuals attitude toward using the new technology. This attitude then motivates a behaviour intention which in turn initiates the actual behavior [Williams et al 2015]. Conceptual model of TAM is shown in Figure 1.

Some forms of TAM take out attitude, arguing that the antecedents affect the behavior intention directly. These are called parsimonious models of TAM. Key thing in TAM is that it does not make any assumptions about the actual quality of the new technology or innovation but focuses on what the user perceives of it. In their study of forced technology situations Jones et al. [2010] found that managerial support has major influence over perceived ease of use. In all cases it should be possible to influence the perceived ease of use with proper education during implementation of the technology.
4 THE UNIFIED THEORY OF ACCEPTANCE AND USE OF TECHNOLOGY

Past research on user acceptance of technology has been rich in volume and also in theories generated [Williams et al 2015]. The unified theory of acceptance and use of technology (UTAUT) was combined from several theories in 2003 by Venkatesh et al [2003]. They reviewed and integrated eight dominant models of the time to create one with more explanation power. Theories included in forming UTAUT are: Theory of Reasoned Action, the Technology Acceptance Model, the Motivational Model, the Theory of Planned Behavior, a combined TBP/TAM, the Model of PC Utilization, Innovation Diffusion Theory, and Social Cognitive Theory [Tornatzky and Fleischer 1990]. In their study, Venkatesh et al. [2003] show that UTAUT outperforms the theories it has been based on. Since its creation, UTAUT has been widely used in variety of fields [Williams et al 2015].

As can be seen from Figure 2, UTAUT has some degree of similarity with TAM. This is not surprising as TAM is one of the theories UTAUT has been based on. UTAUT adds six new constructs in addition to those found from TAM – and discards attitude. The new construct in direct determinants of behavioral intention added is social influence. Another new construct is facilitating conditions, which is seen as direct determinant of use behavior [Williams et al 2015].

The other four constructs that the model adds are conceptualized as moderators for the direct determinants. These are user's gender, age, experience and voluntariness of use. These moderating
constructs are not applicable for organizational research as such. However, it can be argued that these constructs can be applied to organization as well by calculating mean values of all the employees of the organization. Indeed, few studies have been made on organizational context with UTAUT [Gangwar et al 2013]

5 TECHNOLOGY-ORGANIZATION ENVIRONMENT FRAMEWORK

In their meta-analysis of research conducted between 2010 and 2012 Gangwar et al. [2013] identify Technology-Organization-Environment (TOE) framework as one of the more widespread frameworks when researching IT adoption. TOE framework was originally developed by Tornatzky and Fleischer [1990]. Main benefit of TOE is that it is free from industry and company size restrictions. Critics of TOE state that the framework is just taxonomy and does not really offer any conceptual depth. It contains three contexts, which are explained in the next paragraphs and elaborated in Figure 3.

Technological context holds all the variables influencing adoption of innovation. Gangwar et al. [2013] found that: “The studies found that system assimilation, trailability, complexity, perceived direct benefits, perceived indirect benefits and standardization are significant variables while observability is found insignificant” Organizational context is the most interesting one considering the scope of this thesis. It refers to organizational characteristics and resources of company. The studies identify several significant aspects of organization: degree of formalization, managerial structure, trust, human resources, organizational slack, innovation capacity, knowledge capability, linkages among employees, financial resources, firm structure, operational capability, strategic use of technology, technological resources, top management support, quality of human capital, organizational knowledge accumulation, expertise and infra-structure and organizational readiness [Gangwar et al 2013, Bradford et al. 2014].

Many of these organizational topics identified in TOE are tied to findings on the effects of the digitalization in this research. Environmental context focuses on the environment in which the company operates. In this case it means mostly factors influencing whole industry, such as government regulations or incentives. “Significant variables in environmental context include customer mandate, competitive pressure, external pressure, internal pressure, trading partner pressure, vendor support, commercial dependence, environmental uncertainty, information intensity and network intensity while government regulation is not identified as significant variable” [Gangwar et al 2013].

Fig. 3. The context of technological innovation [Tornatzky and Fleischer 1990]
6 FACILITATORS OF DIGITALIZATION

Some studies have set to find out what drives digitalization. Some of the answers are intuitive and others maybe not so. Yao et al. [2009] find support for the very intuitive assumption that bigger IT spending helps in adopting new technologies. Human resources management practices have also been linked as factors facilitating digitalization [Carroll and Wagar 2010]. Jeyaraj et al. [2006] published a meta-analysis of the research made in the subject of diffusion of IT-based innovations between 1992 and 2003. In their study of 99 research articles, they find four best predictors for IT application, here presented in Figure 4. The scores in the figure are calculated as percentage of the times the factor was found significant from all the studies it was used. External pressure was found being significant facilitator of IT adoption in all six of the studies it was tested on. External pressure stems from suppliers, customers or industry standards. Professionalism of IS unit was found significant in 7 studies of the total 8 times it was studied.

This finding is seconded by Scupola [2012], who identifies the lack of knowledge to specify system requirements and the lack of IT competence as organizational operative barriers. External information sources was also found to be significant in seven of the eight studies it was studied. Top management support was studied the most of the best predictors. It had been in 12 studies, of which 10 found it to be significant [Jeyaraj et al 2006].

![Figure 4: Facilitators of IT adoption (Jeyaraj et al 2006)](image)

Scupola [2012] studied ITC adoption in facilities management supply chains of Denmark. She extracted both organizational and technology driven facilitators for the adoption process. These findings present support to the work of Jeyaraj et al. [2006], offering organizational drivers closely related to top management support. These drivers include company policy and better strategic and tactic facilities management decisions. She also identifies seven external drivers and barriers such as industry characteristics, supplier interdependence, lack of collaboration among software providers and government regulation.

7 INFORMATION SYSTEMS STRATEGIC ALIGNMENT

A topic that borders the effects of digitalization is information systems (IS) strategic alignment. There is a large body of research done in this topic ([Preston and Karahanna 2009, Reich and Benbasat 2000, Johnson and Lederer 2010, Alaceva and Rusu 2015]), including the barriers and inhibitors of IS strategic alignment on companies. The results of these studies are included in this thesis, as they offer reason why information systems are not perceived as working well in companies - something that should act as a barrier for further digitalization as well.
There is no clear, agreed on definition or model for strategic alignment of IS. Preston and Karahanna [2009] find two views for the term in their literature review of the subject. First one, the intellectual dimension of strategic alignment, defines it as alignment between business and IS on various dimensions such as strategy, plans or infrastructure of processes. The second one, the social dimension of strategic alignment is defined as the mutual understanding and commitment to business, objectives and plans between business and IT departments.

Alaceva and Rusu [2015] argue that companies cannot reach intellectual dimension if the social dimension is not achieved before. They study the social dimension in their case study of a large Swedish company. They divide this dimension in four subgroups: Shared domain knowledge between business and IT executives, Successful IT history, Communication between business and IT executives and connections between business and IT planning. It seems that communication, connection and shared domain knowledge should be interlinked as concepts, as they are mainly asserting that the main barrier of IS alignment is lack of communication and understanding between business and IS departments.

A study by Johnson and Lederer [2010] support the finding of Alaceva and Rusu [2015], with the result that the prerequisite for IT alignment is mutual understanding of CEO (Chief Executive Officer) and CIO (Chief Information Officer) of the company. Conceptually the results from these two studies are very close even though the terms used are a bit different.

8 DISCUSSION

Even while digitalization as a word and as a phenomenon has been hyped for the past decade, many aspects are still unclear. It has no standard definition – in many discussions it is used to convey different meanings. The main objective of this article is to summarize the state of research made on understanding how digitalization is advanced or how its progression is halted in organizations. It seems that the most important factors in software are the ones affecting how user perceives it. This might be changing a with the introduction of artificial intelligence models which would render most of the users obsolete and provide a substantial leap forward in productivity of an organization. This productivity leap should be more imminent in large private organizations where a lot automatizable office work is being done. Small and governmental organizations might be slower in adapting the new artificial intelligence based technologies. Management support was unsurprisingly found to have major effect in advancement of new technologies in organizations. However, majority of the current studies and of the current explanations are not very fine grained. A fruitful research topic would be “what are the organizational prequisites for IS adoption” – with the aim of more fine grained information about the phenomenon than simply “top management support”.

9 REFERENCES


Relationship Between Design and Defects for Software in Evolution

MATIJA MILETIĆ, MONIKA VUKUŠIĆ, GORAN MAUŠA and TIHANA GALINAC GRBAC,
University of Rijeka, Faculty of Engineering

Successful prediction of defects at an early stage is one of the main goals of software quality assurance. Having an indicator of the severity of defect occurrence may bring further benefit to allocation of testing resources. Code churn in terms of modified lines of code was found to be a good indicators of bugs. However, that metric does not reveal the structural changes of the code design. The idea of this project is to analyze the relationship between the evolution of object oriented software metrics and the appearance of defects. To achieve this, the absolute and relative differences between the initial version of a class and its current version were calculated for each metrics as indicators of design change. Then, the correlation between these differences and the number of defects were analyzed. Our case study showed that certain metrics have no influence on defect occurrence, while several of them exhibit moderate level of correlation. In addition, we concluded that the relative differences were inappropriate indicator for determining the relationship.

Categories and Subject Descriptors: D.2.5 [SOFTWARE ENGINEERING]: Testing and Debugging—Tracing; D.2.9 [SOFTWARE ENGINEERING]: Management—Software quality assurance (SQA); H.3.3 [INFORMATION STORAGE AND RETRIEVAL] Information Search and Retrieval

Additional Key Words and Phrases: Design chage, defect occurrence, evolving software, correlation

1. INTRODUCTION

In software development, one of the main problems is predicting defects during the software product life cycle. In our analysis, we focused on that problem and investigated whether it can be tackled as early as in the design stage of evolving software. In the design stage of software life cycle, most of the effort is invested in emergence of a module. Then, with each upgrade of this module, it is possible to break concepts of good design. Numerous changes are introduced, and these changes can also cause defects. From this standpoint, we would like to find the design metrics that are critical to the appearance of defects. Product metrics have been successfully used to build prediction models [Basili et al. 1996], so a great number of such metrics is investigated in this study.

When developing a software product, it is essential to plan the necessary actions needed to upgrade a certain functionality. Often, through this process, developers need to modify, delete or add new lines of program code. The amount of program code that goes through these actions is called code churn. Developers should strive to minimize the quantity of program lines that will be modified, thus achieving a smaller code churn ratio [Munson and Elbaum 1998]. After the development phase, the software...

This work has been supported in part by Croatian Science Foundation’s funding of the project UIP-2014-09-7945 and by the University of Rijeka Research Grant 13.09.2.2.16.

Author's address: M. Miletić; email: mmiletic@riteh.hr; M. Vukušić; email: movukusic@riteh.hr; G. Mauša, Faculty of engineering, Vukovarska 58, 51000 Rijeka, Croatia; email: gmausa@riteh.hr; T. Galinac Grbac, Faculty of engineering, Vukovarska 58, 51000 Rijeka, Croatia; email: tgalinac@riteh.hr.

Copyright © by the paper's authors. Copying permitted only for private and academic purposes.

product also requires maintenance. Many changes are often introduced during maintenance, which are caused by various factors. Some of them include changes in the environment, unused code, rarely updated code, etc. With the aging of the software product and the consequent links of the above-mentioned factors, the software becomes increasingly difficult to maintain, leading to software design deterioration [Land 2002]. Code churn was used to predict defects in many studies but a finer investigation of source code changes yields stronger correlation [Giger et al. 2011].

Projects in evolution are consisting of a number of versions of program code. In the design stage of each version, new requirements are introduced into the existing system architecture. The effort put into this phase affects the effort required to fix defects [Nadir et al. 2016]. Different design metrics of the object-oriented program code can be measured then. It would be highly beneficial to consider the effect of the program design change on the severity of defect occurrence at this early stage. Their relationship will be analyzed as a first step to achieve this goal. The main contribution of this paper is the methodology for investigating the effect of the program design change on the severity of defect occurrence. The design change is represented by the relative and absolute differences of software metrics between consecutive releases of an evolving project. The number of defects is regarded as the severity of occurrence of defects, with higher values corresponding to greater severity.

The study analyzed the above mentioned relationship between calculated differences and defect occurrence according to the methodology presented in section 2. This project performed a case study based on a sample of data from two Eclipse open source projects according to the description in section 3. The results of this analysis are represented in form of scatter plot diagrams and 3D histograms in section 4. Furthermore, the results were grouped for easier comprehension and interpretation discussed in section 5. Based on calculated data, it is possible to determine if there are some metrics that developers should pay special attention to in the design phase. In addition to these pieces of information, it should be possible to reduce the defect appearance. A future step would be to build a software defect prediction model based on the output data of our program. For example, Genetic Programming algorithms have recently been used to generate high-quality predictors that perform this task successfully [Mauša and Grbac 2017].

2. METHODOLOGY

Figure 1 displays data flow from our project and encapsulates all the steps in data processing. Input data includes several consecutive releases of open-source projects and contains the values of software metrics and the number of defect occurrences. The output data is consisted of csv files that contain program design change and graphical representations of correlation analysis.

To measure the program design change, static code attributes software metrics are used. The amount of change is expressed by the relative and absolute difference between the current and a previous release of a class. Relative code churn was found to have stronger correlation with defect density that absolute churn for lines of code (LOC) [Nagappan and Ball 2005] so this study will analyze both. The values of metrics and calculated differences for each metric are written in output file so that we can use them in the later analysis. The program also adds a binary indicator if the class is found in a previous version of program code, named "Class found". It is worth noting that a great number of software modules in a complex system is expected not to contain defects or code change [Fenton and Ohlsson 2000] [Runeson et al. 2013]. Since this project is interested in the severity of defect occurrence, the non-defective software modules are not considered. Furthermore, because of interest in the quantity of design change, the software modules without a previous version are also not considered.

Within this case study, two algorithms are prepared: "compareFiles.py" and "compareOldestFiles.py". These algorithms calculate the amount of change for all the metrics of software modules in a selected (current) release. The "compareFiles.py" script computes the differences between the current release
and all the previous releases selected by the user, while the "compareOldestFiles.py" script computes the differences between the current release of a software module and its first (oldest) appearance in the evolution of the project. The computed amount of change is added to input data and algorithms generate the "writeResults.csv" or "writeOldestResults.csv" output files.

These output files are afterwards used as input files for "correlationCount.py" and "generateHistogram.py" algorithms. The "correlationCount.py" algorithm generates a scatter plot diagram between the amount of change of each metric and the number of defects, with corresponding correlation coefficients. The correlation analysis is a standard procedure in analyzing the relationship between a metric and defect count and it may even reveal a good candidate predictors for defect prediction models [Zhang et al. 2017]. The "generateHistogram.py" algorithm generates a histogram with a 3D projection. To analyze the results, they are grouped according to Spearman’s correlation coefficient. The aim of this analysis is to detect which design metrics are critical to the severity of defect occurrence.

3. CASE STUDY

The idea of this project was to analyze data from different datasets and try to find relationship between input data and defect occurrence in program code. To accomplish this, the following functionalities were implemented iteratively:

(1) Algorithm that computes absolute and relative difference between metrics for every class inside current project version and one or more selected files for comparing.

(2) Algorithm that computes relative and absolute differences for every class from current file with the oldest version of project that contains certain class.

(3) Graphical representation of data collected from previous algorithms:
   (a) Histogram within 3D projection showing relationship between relative or absolute differences, defect count and frequency of defect occurrences inside a certain range.
   (b) Scatter diagrams showing relationship between relative or absolute differences and defect count accompanied with calculated Pearson’s and Spearman’s coefficient.

3.1 Datasets

Input datasets that are used in our research included 5 consecutive versions of Eclipse JDT (Java development tools) and PDE (Plug-in Development Environment) open source projects. The data were
collected using a systematically defined data collection procedure [Mauša et al. 2016] implemented in Bug-Code Analyzer tool. The JDT project provides the tool plug-ins that implement a Java IDE supporting the development of any Java application, including Eclipse plug-ins\textsuperscript{1}. The Plug-in Development Environment (PDE) provides tools to create, develop, test, debug, build and deploy Eclipse plug-ins, fragments, features, update sites, and RCP products\textsuperscript{2}. The included releases are named: 2.0, 2.1, 3.0, 3.1 and 3.2.

The datasets are given in csv format with comma as the delimiter and point as decimal mark. First row contains description of metrics: column 1 is the file path, columns 2-49 are independent variables (software metrics) described with their abbreviations, column 50 is the dependent variable (number of defects). The datasets were cleared of files that contained .example or .tests in their path. Full list of metrics and the explanation of their abbreviations can be found in [Mauša and Grbac 2017] and on-line\textsuperscript{3}.

3.2 Evaluation

After the D’Agostino and Pearson’s normality test was performed, it was found that input data is not normally distributed. This analysis is necessary to select an appropriate method for the calculation of correlation coefficients. Unlike Pearson’s correlation, the Spearman’s correlation analysis does not require the data to be normally distributed. Hence, this non-parametric statistical method had to be used for the purpose of correlation expression [D’Agostino and Pearson 1973].

Spearman’s correlation coefficient is a statistical measure of the power of a monotonic relationship between paired data. It is calculated according to the formula presented in equation 1 [Myers and Well 1991].

$$r_s = 1 - \frac{6 \sum_{i=1}^{n} d_i^2}{n^3 - n}$$

(1)

where $r_s$ denotes its value in a sample $s$. The range of possible values is as follows [Lehman 2005]:

$$-1 \leq r_s \leq 1$$

(2)

The interpretation is similar to Pearson’s correlation, but closer to a stronger monotonic relationship. The relationships between variables may be determined by their correlative dependence as follows:

(1) Positive correlation - the small value of one variable corresponds to the small value of the second variable, also the high value of one variable corresponds to the high value of the other variable

(2) Negative correlation - the small value of one variable corresponds to the high value of the second variable and vice versa.

(3) Correlation does not exist when a value of a variable can not be inferred from the value of another variable based on the value of a variable. The points in such a graph are scattered [Evans 1996].

Correlation is the size of the effect and so we can verbally describe the correlation force using the Table I [Evans 1996].

\textsuperscript{1}http://www.eclipse.org/jdt/
\textsuperscript{2}http://www.eclipse.org/pde/
\textsuperscript{3}http://www.seiplab.riteh.uniri.hr/wp-content/uploads/2016/12/Table-of-Metrics.pdf
Table I. Correlation ranges

<table>
<thead>
<tr>
<th>Coefficient</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00-0.19</td>
<td>Very poor correlation</td>
</tr>
<tr>
<td>0.20-0.39</td>
<td>Low correlation</td>
</tr>
<tr>
<td>0.40-0.59</td>
<td>Moderate correlation</td>
</tr>
<tr>
<td>0.60-0.79</td>
<td>Strong correlation</td>
</tr>
<tr>
<td>0.80-1.00</td>
<td>Very strong correlation</td>
</tr>
</tbody>
</table>

4. RESULTS

The first analysis counted the number of files that had its earliest version in each of the previous releases. Table II shows this number as follows: first column represents the number of files that had no previous release, the second column represent the number of files that had its earliest version in the previous release, and so on until the last column shows the number of files that had its earliest version in the oldest release.

Table II. Number of files within each release of open source project

<table>
<thead>
<tr>
<th>Project</th>
<th>Release 3.2</th>
<th>Release 3.1</th>
<th>Release 3.0</th>
<th>Release 2.1</th>
<th>Release 2.0</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>JDT</td>
<td>333</td>
<td>231</td>
<td>339</td>
<td>110</td>
<td>1220</td>
<td>2233</td>
</tr>
<tr>
<td>PDE</td>
<td>417</td>
<td>271</td>
<td>255</td>
<td>79</td>
<td>329</td>
<td>1351</td>
</tr>
</tbody>
</table>

In Table II we can see that in "PDE" project the number of newly released files is approximately equally distributed in every release. For the "JDT" project we can see that most of the files were created in the first release (2.0). Every following release is mainly built upon the previous release and the number of files depends on the newly added features. With the exception of Release 2.1, most of the other versions have similarly distributed newly released files.

Tables III and IV show the obtained Spearman correlation coefficients for the following pairs:

—Correlation between the metric's real value and number of defects (column "value")
—Correlation between the absolute difference of the metric and number of defects (column "abs_diff")
—Correlation between the relative difference of the metric and number of defects (column "rel_diff")

Under the column "single_previous_rls" the correlation values are shown for the comparison between the fifth version of the open source project release and its previous version. Under the column "oldest_rls" the same values are shown, but for the comparison between the fifth version and the oldest appearance of every class in it.

Using the previously mention tables we can see which of the metrics achieve the greatest correlation coefficient. These metric have the most significant impact on the occurrence of defects. For that reason, in Tables III and IV we showed only the metrics with the correlation coefficient value greater then "0.20". Metrics that fall into the category of "Very poor correlation" in both datasets include the following: HCLOC, MI, DIT, MINC, S_R, FOUT, NSUP, NSUB, INTR, MOD. Metrics that achieve the "Very poor correlation" only in "JDT" project include HCWORD, LCOM, COH, EXT, MPC. Similarly, metrics that fall into that category only in "PDE" project include CLOC, CWORD, AVCC, CBO, CCML, F_IN, HIER, SIX, NCO.

The degree of correlation between the absolute and relative differences and the number of defects are shown in a scatter plot diagram representing a two-dimensional graph. The y-axis represents the number of defects, while the x-axis represents calculated absolute or relative difference of the certain metric. Additionally, Spearman's correlation coefficient is calculated and displayed above diagram. In order to make results more intuitive and better visualized, results are shown graphically on histogram in 3D projection. From them it is easier to bring conclusions about how the data is divided between...
given ranges. Every histogram has x, y, and z axis. X-axis represents computed relative or absolute difference between metrics. Y-axis represents data about defect count from the given input file. Z-axis represents frequency of data appearance from y-axis in a range on x-axis. Due to limited space, these diagrams are shown only for SLOC_L metric which had the greatest value of correlation coefficient in both projects in figures 2 and 3. After absolute difference value higher than 500 the number of defect occurrences is negligible. The biggest concentration of defect occurrence with values between 0 and 5 are in range of absolute difference from 0 to 500, with frequency value from z axis of around 500. Similarly, on the right side of the figure the second histogram represents number of defect occurrences inside a certain range of relative difference on x axis. From the Figure 3 we can see that the biggest concentration of the defect occurrences are in a range of 0 to 1 values on x axis. After the relative difference reaches the value of 1 there is no significant defect occurrence.

![Graph showing scatter diagrams of the correlation between differences and number of defects for the SLOC_L metric, with the corresponding Spearman's correlation coefficient.](image)

**Fig. 2.** Scatter diagrams of the correlation between differences and number of defects for the SLOC_L metric, with the corresponding Spearman’s correlation coefficient.

5. DISCUSSION

In this chapter we will give a more detailed overview of the obtained results and show them on the example of specific metrics. Further more, we will discuss results obtained from two different datasets, that refer to "JDT" and "PDE" open source project releases.

Analysis of the obtained results have shown that certain metrics from different datasets fall into the same correlation range. Analyzing both datasets, we noticed that the greatest correlation coefficient are achieved for the difference between the value of the metric itself and the number of defects. This confirms the general appropriateness of metrics for software defect prediction and indicates their capability to distinguish the severity of defect occurrence. However, this study was motivated to investigate whether similar effect can be discovered for the quantity of change. When sorting the data by the "value" column we can observe that most of the metrics fall into the category of moderate correlation. The lowest value of correlation coefficients are achieved between the relative difference and the number of defects. Hence, we used the metrics from the "abs_diff" column for this discussion.

When analyzing the data from the Tables III and IV we can see that the highest correlation coefficient are most often achieved by the same metrics. For example, we can see from the Table IV that
metrics SLOC\_L and SLOC\_P achieve similar correlation coefficients in single and oldest releases. In contrast, metric R\_R falls into "Moderate correlation" in the single version, but in the oldest version it falls into negative "Very poor correlation". Hence, we can conclude that both of the proposed approaches may contribute to the research performed in this project.

Tables III and IV do not contain the same set of metrics because the ones with low correlation coefficient were omitted. However, there are some metrics with higher correlation coefficient. Metrics LOC (Total Lines of code), SLOC\_P (Physical executable source lines of code) and SLOC\_L (Logical source lines of code) achieve the greatest positive correlation coefficients in both datasets. The correlation of these metrics is intuitive because the greater amount of code is introduced, the greater is the possibility for a defect to occur. It is also important to mention the cyclomatic complexity metric which is used to indicate the complexity of a program [McCabe 1976]. The correlation coefficients of metrics MVG (McCabe VG Complexity), AVCC (Average Cyclomatic Complexity of all the methods in the class) and MAXCC (Maximum Cyclomatic Complexity of any method in the class) fall into the category of positive "moderate correlation". This shows that more independent paths through an algorithm, i.e. higher cyclomatic complexity, increase the possibility of defect occurrence. Furthermore, MVG exhibits the highest correlation coefficient in JDT project when a file is compared to its oldest version.

For better representation of data Figures 2 and 3 are displaying computed results for SLOC\_L metric. The figures clearly show higher Spearman's correlation coefficient for relationship between the absolute difference value and the number of defect occurrence, with coefficient value around 0.48, than for the relative difference value. Spearman's correlation coefficient between the relative difference and defect occurrence is around 0.20, thus is very low. This is a strong indicator that absolute differences are more relevant for measuring the aforementioned relationship of defect occurrence.

From the "JDT" dataset the following metrics achieved the greatest correlation coefficients in the single previous version: HCLOC, MVC, LOC, SLOC\_P, SLOC\_L. Similarly in the oldest version, metrics MVG, LOC, SLOC\_P, SLOC\_L and C\_SLOC achieved the greatest correlation coefficients. Metrics with the greatest correlation coefficient in the "PDE" dataset from the single previous version include UWCS, LMC, BLOC, MVG and No\_Methods metrics. From the oldest version they include BLOC, RFC, LMC, No\_Methods and UWCS metrics.

5.1 Threat to Validity

The validity of this small scale case study is affected by the choice of data. This is a first preliminary study, so it is based only on a sample of data from five consecutive releases of two open source projects from Eclipse community, namely, JDT and PDE. Thus, external validity is the main threat to this study.
Table III. Spearman’s correlation coefficient for JDT input data

<table>
<thead>
<tr>
<th>Metrics</th>
<th>single_previous_rls</th>
<th>oldest_rls</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOC</td>
<td>0.58</td>
<td>0.53</td>
</tr>
<tr>
<td>SLOC_P</td>
<td>0.58</td>
<td>0.52</td>
</tr>
<tr>
<td>SLOC_L</td>
<td>0.58</td>
<td>0.52</td>
</tr>
<tr>
<td>MVG</td>
<td>0.62</td>
<td>0.56</td>
</tr>
<tr>
<td>BLOC</td>
<td>0.49</td>
<td>0.44</td>
</tr>
<tr>
<td>C_SLOC</td>
<td>0.58</td>
<td>0.49</td>
</tr>
<tr>
<td>CLOC</td>
<td>0.40</td>
<td>0.35</td>
</tr>
<tr>
<td>CWORD</td>
<td>0.51</td>
<td>0.45</td>
</tr>
<tr>
<td>No_Methods</td>
<td>0.50</td>
<td>0.41</td>
</tr>
<tr>
<td>AVCC</td>
<td>0.47</td>
<td>0.41</td>
</tr>
<tr>
<td>NOS</td>
<td>0.53</td>
<td>0.47</td>
</tr>
<tr>
<td>HBUG</td>
<td>0.53</td>
<td>0.48</td>
</tr>
<tr>
<td>HEFF</td>
<td>0.52</td>
<td>0.48</td>
</tr>
<tr>
<td>UWCS</td>
<td>0.51</td>
<td>0.41</td>
</tr>
<tr>
<td>INST</td>
<td>0.28</td>
<td>0.28</td>
</tr>
<tr>
<td>PACK</td>
<td>0.13</td>
<td>0.11</td>
</tr>
<tr>
<td>RFC</td>
<td>0.49</td>
<td>0.40</td>
</tr>
<tr>
<td>CBO</td>
<td>0.10</td>
<td>0.39</td>
</tr>
<tr>
<td>CCML</td>
<td>0.45</td>
<td>0.36</td>
</tr>
<tr>
<td>NLOC</td>
<td>0.52</td>
<td>0.45</td>
</tr>
<tr>
<td>PLIN</td>
<td>0.10</td>
<td>0.36</td>
</tr>
<tr>
<td>R_VAR</td>
<td>0.24</td>
<td>0.47</td>
</tr>
<tr>
<td>LMC</td>
<td>0.54</td>
<td>0.42</td>
</tr>
<tr>
<td>LCOM2</td>
<td>0.42</td>
<td>0.33</td>
</tr>
<tr>
<td>MAXCC</td>
<td>0.54</td>
<td>0.47</td>
</tr>
<tr>
<td>HVOL</td>
<td>0.52</td>
<td>0.48</td>
</tr>
<tr>
<td>HIER</td>
<td>0.38</td>
<td>0.21</td>
</tr>
<tr>
<td>NQU</td>
<td>0.21</td>
<td>0.18</td>
</tr>
<tr>
<td>SIX</td>
<td>-0.30</td>
<td>-0.42</td>
</tr>
<tr>
<td>TCC</td>
<td>0.57</td>
<td>0.49</td>
</tr>
<tr>
<td>NCO</td>
<td>0.51</td>
<td>0.42</td>
</tr>
<tr>
<td>CCOM</td>
<td>0.53</td>
<td>0.44</td>
</tr>
<tr>
<td>HLTH</td>
<td>0.53</td>
<td>0.48</td>
</tr>
</tbody>
</table>

because it cannot discover general conclusions. Nevertheless, the obtained results are a motivation to put more effort in this research direction. Projects from different background, like different communities, development methodologies or written in different programming language need to be included to obtain more general conclusions. As industrial data are difficult to obtain, the construct validity is also threatened. That is why the chosen projects are complex and long lasting ones that may approximate industrial projects. The statistical analyses are threats to internal validity, but they are all based on well known and widely used tests. The conclusions about the level of correlation and the importance of software metrics for defect prediction lacks a precise explanation, hence threatening the conclusion validity. The causality of the conclusions is unknown and it remains open to speculations.

6. CONCLUSION

When developing a project it is very important to pay attention to the appearance of defects in program code. Since the design is the phase in which the requirements are introduced into the existing system architecture, it is particularly important to pay attention to the occurrence of malfunctions that
may arise due to necessary changes. The goal of this project was to establish the correlation between changes in design metrics and the appearance of defects.

This paper provides a methodology for estimating the relationship between the quantity of change and the severity of defect occurrence. The methodology was used in a small scale case study which used 5 subsequent releases of "JDT" and "PDE" Eclipse open source community projects as datasets. The algorithms that implement the proposed methodology have been applied to all software metrics in the datasets. Generated output data is used as the input file to algorithms that computes metric correlation and graphically display obtained relationships. The results were afterwards grouped according to Spearman’s correlation coefficient, as described in section 3.1. The analysis showed a very week or week correlation between changes in metric value and defect occurrence in design stage for most of the metrics. Furthermore, there was no strong correlation found between any of the design metrics and the defect occurrence. However, these results were somewhat expected. The aim of this project was to find design metrics that could bring additional information in the prediction of defects. For example, SLOC_P, SLOC_L and HEFF metrics exhibit a moderate correlation between the changes in the value of a particular metric and defect occurrence. The moderate level of correlation is an indication enough that the proposed metric calculation may improve the categorization of severity of defect occurrence and possibly the software defect prediction. The fact that the values of metrics themselves, which are traditionally already used for software defect prediction, have similar values of correlation,
encourages us to continue this research. When these metrics are known it is possible to pay more at-
tention to them when introducing new requirements, or re-designing the existing software modules, and thus reduce the possibility of defect occurrence.

To summarize, this research came to following conclusions:

(1) Metrics like HCLOC, MI, DIT, MINC, S,R, FOUT, NSUP, NSUB, INTR and MOD exhibit very poor correlation in both projects and may have lower impact on the level of defect occurrence.

(2) Metrics SLOC_L and SLOC_P achieve the highest correlation coefficients in both input datasets. Hence, we can conclude that they have the greatest impact on defect occurrence.

(3) Spearman’s correlation coefficients between the relative difference and defect occurrence are very low. Thus, we can conclude that they are not good indicators of design change quality.

REFERENCES


A Practical Tutorial For FermaT and WSL Transformations

DONI PRACNER and ZORAN BUDIMAC, University of Novi Sad

The transformation system FermaT and the language WSL can be very powerful tools and are successfully used even in some industrial applications. In this paper we try to illustrate a process of making a new transformation and some intermediate steps and ways to test the new functionalities in a simple way.

Categories and Subject Descriptors: D.2.7 [Software Engineering] Distribution, Maintenance, and Enhancement
General Terms: Theory, Experimentation
Additional Key Words and Phrases: software evolution, FermaT, WSL, transformations

1. INTRODUCTION

Software evolution is mainly about how program code changes in its environment. A very important part of these changes is the re-engineering of software that can be done very efficiently and more importantly reliably by using tools that offer formally provable transformations.

FermaT is the current implementation of the language WSL (short for Wide Spectrum Language) and the surrounding code transformation libraries. It is available under the GPL v3 software licence and works on most computer platforms, including Linux, Windows and Mac OS. Early versions of this tool were developed as “The Maintainer’s assistant” [Ward 1989] and it has been developed and reimplemented since then. It has been used in several industrial projects of converting legacy assembly code to human understandable and maintainable C and COBOL [Ward 1999] [Ward 2004] [Ward et al. 2004] [Ward 2013]. It also has support for program slicing [Ward and Zedan 2017] and can be used to derive program code from abstract specifications [Ward and Zedan 2014]. A companion graphical application FermaT Maintenance Environment (FME) [Ladkau 2007] is also available and can be very useful especially for initial experiment with the transformation system. This tutorial will give just a brief overview of WSL and some of the needed ideas and will not go into depths with the syntax of the language that is available in the official manual [Ward et al. 2008].

This paper is organised as follows: Section 2 shows some aspects of FermaT and how to use it and expand it, mainly through the expression and condition simplifiers (Section 2.1) A working example is introduced in Section 2.2 which is then developed into a full transformation that can be added to the system in Section 2.3. Finally Section 3 gives a brief conclusion to the paper. Along the way Some
ways of checking the results of transformations will also be presented, as well as ways to find more information on the structure of the programs, both of which are very important for beginners.

2. WORKING WITH META-WSL

WSL has the ability to work with programs written in WSL. This part of the language is called MetaWSL. A piece of code under inspection is represented by an abstract syntax tree (AST), or more precisely in the current implementation by a list that can have lists as its elements. Meta-WSL procedures know how to handle these lists and what to expect in them when they represent a valid program. Items in the lists have associated general and specific types. For instance a while loop will be of type T_While and will contain in its list a T.Condition and T.Statements. An assignment is of type T_Assign and holds one or more items of type T_Assign (it is separated because it is used in other places as well, and there can be multiple simultaneous assignments) which in turn holds an T_Lvalue and a T_Expression. Tables with these relations can be found in the reference manual for WSL [Ward et al. 2008], or can be found by directly extracting the information from the source code with a command like shown in Figure 1.

```
> grep "\[T_Assign\]" $Fermat/src/adt/WSL-init.wsl
Syntax_Name[T_Assign] := "Assign";
Syntax_Comps[T_Assign] := <T_Lvalue, T_Expression>;
```

Fig. 1. Grepping component types in WSL

To assist many operations, especially transformations, there is also a concept of the current program that is being worked on (accessible via the @Program procedure), the current position in it (represented by a list of indexes in the lists starting from the top one, obtained by @Posn) and the current item (@I) in it. There are many built in procedures to move around in the program and to change it. The current program can be specified with @New_Program and to understand better what are the components of particular statements, the @Print_WSL procedure can be used, as shown in Figure 2. To print the code of an item there is the pretty print @PP_Item procedure, that accepts three parameters: the item, the width of the maximum line, and the file to print out the code to. If the filename is an empty string it will be printed out to the standard output. When working with the whole program, a convenient shorthand is the @Checkpoint procedure which accepts only one parameter, the file name, and always prints the whole program in 80 characters width.

2.1 Item simplification

One of common things done in transforming programs is to detect specific patterns and simplify them. WSL has matching constructs for this type of work. To demonstrate a simple example we’ll try to make a matcher for calling absolute values on negated values and replace them with just the value (of course this is already present in FermaT). To quickly test our matcher, we can create a small program that will define an entry to test on, use the checkpoint command to display the code before and after the changes, and use the FOREACH construct to apply our matcher to all expressions. This is demonstrated in Figure 3. Alternatively we could use commands to move the current position (such as @Down, @Right, etc) in the program to an appropriate expression and apply it just there.

Displaying the changed program, or saving it in a file gives a good indication whether the changes were successful, but additional checks can and should be applied especially when prototyping. This can be done with @Syntax_OK?, as shown in the end of Figure 3. This can catch some subtle errors.
Program:
@New_Program(FILL Statements
  a := 5;
  b := 10;
  PRINT(a + b)
  ENDFILL);

@Print_WSL(@Program, "")

Output:
Statements
Assignment
  : Assign
    : Var_Lvalue a
    : Number 5
Assignment
  : Assign
    : Var_Lvalue b
    : Number 10
Print
  : Expressions
    : Plus
      : Variable a
      : Variable b

Fig. 2. Creating a new program and viewing its structure

@New_Program(FILL Statements
  a:=5; b:=-5;
  PRINT(ABS(-a)); PRINT(ABS(-b))
  ENDFILL);

@Checkpoint("");

PRINT("transforming");

FOREACH Expression DO
  IFMATCH Expression ABS(- ~?x)
    THEN @Paste_Over(FILL Expression ABS(~?x) ENDFILL) ENDMATCH;
  OD;

@Checkpoint("");
IF @Syntax_OK?(@Program) THEN
  PRINT("Syntax OK")
ELSE
  ERROR("Bad syntax") FI

Fig. 3. Absolute value expression matching

For instance if in our paste over command we used “Expressions” instead of “Expression”, the output would look appropriate and the saved file would actually be valid, but the syntax tree would have a problem, and further transformations would likely fail. Calling @Syntax_OK on this version of the code would result in a report shown in Figure 4.

FermaT has a built in maths simplifier and several procedures that rely on it. Procedures such as Simplify, Simplify_Cond and Simplify_Expn return a new simplified item, while others like @Or, @And will first combine conditions and try to simplify them together. More information about the simplifier and adding patterns to it can be found in the manual and the documentation that comes with FermaT, more precisely the document doc/adding-patterns.txt.
2.2 Example: Converting Numeric Codes to Strings

As a working example we will work on a relatively simple and understandable problem. One of the tools that generates WSL code works on bytecode that has numeric codes for characters and handles that by creating @List_To_String calls with the appropriate numbers [Pracner and Budimac 2017]. To make the code more understandable to humans and more compact in general, we will try to transform those into strings.

The first version (Figure 5) assumes that everything in a @List_To_String is a number literal and will not do anything if this is not the case. If all the items in the list are numbers, we can apply the MAP function to the list, and convert the items to their values (@V), convert those to a string with @List_To_String and finally make a valid string item with @Make. This is then pasted over the initial expression. As can be seen from the output shown in Figure 5 this means that if a numeric variable is in the list, the code will not transform it at all. In this example a constant propagation transformation executed before this code would actually solve the problem, but in a general case, the variable “a” could be a user input, or a procedure parameter, and might not be replaceable.

Output:
```plaintext
....
result ------------
a := 92;
PRINT("A");
PRINT("AB");
PRINT(-a);
PRINT(@List_To_String(<a>));
PRINT(@List_To_String(<a, 67>))
```

Fig. 5. Program that converts lists with only numbers to strings
To have a more fine grained handling of individual items, we need to go through them one by one and store the changed versions in a list (called “res” in the code given in Figure 6). If a numeric code is found it should be converted to a character and added to the list. If the previous item in the list was a string, it should be added to it, otherwise it is added as a new item in the list. In this version we will also include anything that is not a number as a separate entry in the resulting list. The final step, when the list is completed is to paste it over the original expression, but it is important to consider that multiple entries should be concatenated, while a single entry is directly pasted over.

```
FOREACH Expression DO
VAR < x := < >, res := < > >:
  IFMATCH Expression @List_To_String(< ~x >)
  THEN
    FOR elt IN x DO
      IF @ST(elt) = T_Number
      THEN
        IF NOT EMPTY?(res) AND @ST(HEAD(res)) = T_String THEN
          t := HEAD(res);
          t := @Make(T_String, @V(t) ++ @List_To_String(< @V(elt) >), < >);
          res := TAIL(res);
          res := < t > ++ res
        ELSE
          s := @Make(T_String, @List_To_String(< @V(elt) >), < >);
          res := < s > ++ res
        FI
      ELSE
        res := < FILL Expression @List_To_String(< ~?elt >) ENDFILL > ++ res
      FI
    OD;
  IF LENGTH(res)>1 THEN
    @Paste_Over(@Make(T_Concat, < >, REVERSE(res)))
  ELSE
    @Paste_Over(HEAD(res))
  FI;
  SKIP
ENDMATCH ENDVAR OD;
```

Fig. 6. Second version of the string converter

It is important to note that while this insistence on converting numbers to characters increases the readability of the code, it can also result in a more complex program with more expressions, which is often not desirable, and is definitely not something to be included in the general simplifier. The actual code that is included in FermaT’s simplifier is shown in Figure 7 and is much more similar to the first version of the code, just much more compact. It also takes care to not convert number 34, which is the code for double quotes and can not be included in a regular string as such, since it is the string delimiter. Further effort could be made handle this case as well, but it is not very common and it would significantly increase the code complexity.

There are also a few other improvements that will be discussed in the next section when the code from Figure 6 will be converted into a full transformation.
Transformations in WSL can be anything that changes the current program while keeping the semantics of the original, with a few potential exceptional cases. For instance there are transformations that will reverse an IF/ELSE statement (for instance to make the condition evaluation simpler), or unroll the first loop of a FOR statement. Additionally there are applicability tests that will check if the semantics would be affected by the specific transformation. For instance there is a transformation that deleted the current item, but only if it is redundant. Therefore the test for this transformation is checking if the current item is redundant.

The main exception to the preservation of full semantics are the slicing transformations, as these by definition preserve only a part of the original behaviour of the program that is relevant to the slicing criterion. These will not be covered in this paper.

The transformations that are built into FermaT are kept in the src/trans folder and new ones can be added there and the whole system recompiled. Alternatively it can be added “on the fly” in the working directory using a patch.tr file. More details about this can be found in the manual and the documentation that comes with FermaT.

Transformations themselves are represented by two WSL files. The first one holds the code of the transformation. There are two main entry points that need to be defined in this file. The first one is a test procedure that has no parameters and checks whether the transformation can be applied to the current item in the program. It should raise errors with @Fail if the transformation is inapplicable, or call @Pass otherwise. The other procedure is the actual transformation that receives a single parameter with any potential additional data, for example a rename transformation will receive the old and the new names. Other than these there can be any number of helper procedures defined, and to comply with the definition of a WSL program the file also needs to contain a body for the main program which can be a single @SKIP instruction.

The second file should be named the same as the first one with a “.d” suffix and it holds the description of the transformation and some meta information as well as the names of the actual procedures to test and to apply the transformation from the first file. Figure 8 shows how this file should look for a new transformation that is based on the code shown in the previous section.

Figure 9 shows how the main file with the new transformation should look like. This transformation has no additional procedures. The test procedure just calls @Pass always, since there are no specific pre conditions needed for the transformation to be applied and the semantics are not changed anyway — at worst it will not find anything to change and leave the original program as it is. A more zealous version of the test procedure could check if there are any @List_To_String items, and even analyse their content to see if the transformations will change anything. On the other hand a general approach to programs that call transformations is to first call the test and then to call the main transformation, which would result in duplicate checks and loss of efficiency.
IF EMPTY?(TR_SimplifyChar) THEN TR_SimplifyChar := @New_TR_Number() FI;

TRs_Proc_Name[TR_SimplifyChar] := "SimplifyChar";
TRs_Test[TR_SimplifyChar] := !XF funct(@SimplifyChar_Test);
TRs_Code[TR_SimplifyChar] := !XF funct(@SimplifyChar_Code);
TRs_Name[TR_SimplifyChar] := "Simplify Char";
TRs_Keywords[TR_SimplifyChar] := < "Simplify" >;
TRs_Help[TR_SimplifyChar] := "Simplify Char will find expressions like ' @List_To_String(<97>)' and replace them with chars."
TRs_Prompt[TR_SimplifyChar] := ""
TRs_Data_Gen_Type[TR_SimplifyChar] := ""

Fig. 8. Transformation description file

MW_PROC @SimplifyChar_Test() ==
  @Pass END;

MW_PROC @SimplifyChar_Code(Data) ==
  FOREACH Expression DO
    IFMATCH Expression @List_To_String(<~*>)
      THEN VAR < res := < >, IS_OK := 1 >:
        FOR elt IN x DO
          IF @ST(elt) = T_Number
            THEN IF NOT (EMPTY?(res)) AND @ST(HEAD(res)) = T_String
              THEN res := <@Make(T_String, @V(HEAD(res))
                  ++ @List_To_String(<@V(elt)>),
                  < >) > ++ res
            ELSE res := <@Make(T_String,
                @List_To_String(<@V(elt)>), < >) ++ res FI
          ELSIF @ST(elt) = T_Variable
            THEN res := <FILL Expression @List_To_String(<~?elt>) ENDFILL>
                ++ res
          ELSE IS_OK := 0 FI OD;
      IF IS_OK = 1
        THEN IF EMPTY?(res)
          THEN @Paste_Over(@Make(T_String, "", < >))
        ELSIF LENGTH(res) > 1
          THEN @Paste_Over(@Make(T_Concat, < >, REVERSE(res)))
        ELSE @Paste_Over(HEAD(res)) FI FI ENDMATCH OD END;
  SKIP

Fig. 9. Transformation main file
The main procedure is very similar to the earlier developed version shown in Figure [6] with some commands being less verbose and less temporary variables used, which in turn can make it harder to read. There are several functional improvements beside that. In a proper program @List_To_String should receive a list of numbers and numeric variables, anything other in the list results in undefined behaviour, therefore it is probably best if our transformation leaves any problematic call as it was. This means that the new version only handles numbers and variables, and if the current item is anything else it sets the error flag which will result in no changes being applied. The other thing that is changed is that the new version correctly handles an empty list being passed to the procedure and replaces it with an empty string (the previous version would crash if this was the case).

3. CONCLUSIONS

FermaT and the WSL language offer a powerful transformation library that has been developed for many years, and has been used in industrial applications of re-engineering legacy software [Ward 2004] [Ward 2013].

This paper shows some practical steps in working with FermaT and WSL, especially in early prototyping of new ideas. Some aspects of the maths simplifier and full transformations are presented with a concrete understandable example being developed in the process. Attention was also given to ways to check if the written programs output valid WSL code and how to find more information on the structure of programs, which tends to be a big problem for beginners.

REFERENCES


Martin Ward. 2013. Assembler restructuring in FermaT. In SCAM. IEEE, 147–156. DOI: http://dx.doi.org/10.1109/SCAM.2013.6648196


A Temporal File System for Student’s Assignments in The System Svetovid

IVAN PRIBELA and DONI PRACNER, University of Novi Sad

Storing students’ assignments in practical computer science exams has several specific problems. In this paper we present some of the advantages of using a file system with a temporal dimension and present two working solutions, one of which was used in several courses with hundreds of students.

Categories and Subject Descriptors: K.3.2 [Computers And Education]: Computer and Information Science Education—Computer science education; D.2.11 [Software Engineering]: Software Architectures—Data abstraction

General Terms: Design, Experimentation

Additional Key Words and Phrases: file system, temporal, git, submission system, Svetovid

1. INTRODUCTION

Many computer science courses are focused on practical exercises as a form of continual assessment and examination. This approach undoubtedly offers friendly conditions to the students. As they use computers throughout the process of program development, this conditions are very close to the environment they will meet at work. However, in such circumstances, there are a few major problems that come from intricacies of electronic submission of the final solutions.

The first problem stems from the tendency of students to cheat and collude during solution development. This is only amplified by poor academic discipline among the students [Pribela et al. 2009].

Another problem is posed by bad or incomplete submissions on the part of the students, whether intentional or unintentional [Reek 1989]. If a student forgets to submit some of the required files the program might fail during assessment. Contacting the student to get the missing files delays the assessment, and there is no guarantee that the student has not been working on the missing part after the due date had passed.

The last problem is that in most cases only the final solution is submitted. There is no record of the effort or the path that the student took to come to the submitted state [Joy et al. 2005]. After the submission it is too late for any corrections, unless the teacher is willing to accept re-submissions and has spare time to re-evaluate them.

The described problems come from our personal experience, but are also recognised by many other institutions [Reek 1989, Hawkes 1998, Dempster 1998, Joy et al. 2005, Vujošević-Janičić et al. 2013, Christian and Trivedi 2016].
To solve them, a better method of submission is needed. This method should allow a student to give a complete and usable copy of the solution to the teacher without making it accessible to other students. Also the teacher should be able to run and check the solution without effort and focus on giving prompt feedback to the student.

As most of submission systems are focused only on submission, we have developed an interactive development environment integrated with a secure submission system in one package [Pribela et al. 2009].

This environment, however, focuses mainly on the prevention of plagiarism and collusion among the students. It provides to the students a standard set of tools expected in development environments like access to a console, compilation and execution of the code, etc. It also incorporates an editor specially designed to enable copying, pasting, dragging and dropping within the environment but disallowing it from and to the outside. This environment represents the first layer of defense against cheating: the prevention.

Described features are not the only vectors that students use to cheat, with prevailing other channel being the retyping from other sources. With this in mind and in the light of ever evolving and ubiquitous social media platforms, a second layer of defense is recommended. This second layer represent detection and investigation of any wrongdoing and is enabled by tracking of history and evolution of the student's solution through time. Although detection can never be completely certain the added information can only help in the investigation.

To address this and the other problems, we implemented a special file system that is described in this paper.

Usually, different students use different tools and programming environments while working on the solutions to the programming assignments. This diversity enables every student to use their preferred tools and environments. When the student is done working on the solution, the submission procedure is started. The usual submission process then copies the necessary files and records the submission timestamp.

To address this, we envisioned a process of continual submission. More precisely, a snapshot of the code is taken and submitted at key points in time during the work on the assignment. All snapshots are stored in a single file system structure and the project state can be recalled as it was at previous moments in time.

The requirements for this special file system are outlined in the following paragraphs.

The Svetovid system should manage multiple hierarchical file systems. These systems will be used to store and retrieve data like texts of student assignments, students' solutions of these assignments, test suites for the submitted solutions, or any other kind of files.

The requirements placed for the individual file systems are very similar to requirements for many other modern file systems. Each file system has a root folder which contains other folders and files. These file systems work just like any other hierarchical file system, files have content, while folders hold files and other subfolders.

However, unlike most other modern file systems, a detailed history of meta data and file content is kept. This means that any file system can be viewed as it was at any moment in the past. This enables the teacher to examine how the student's solution evolved, or to check if there were any inconsistencies and the student might be cheating. The student can also view all previous versions of their own files giving them the possibility to revert to an older version if that is more satisfactory.

The rest of this paper is organised as follows: Section 2 present existing solutions for storing data in file systems, mainly virtual file systems and version control systems. Section 3 describes the two solutions given, one with a general temporal implementation (Section 3.1) and the other that relies
on the Git version control system (Section 3.2. Finally a general overview, conclusions and options for future work are given in Section 4.

2. EXISTING SOLUTIONS

2.1 Virtual File Systems

In implementing the proposed solution, we took the approach of a Virtual File System. A virtual file system provides an abstraction layer on top of a more concrete file system. The main goal of this approach is to provide access to different types of concrete file systems in a uniform way. As we also wanted to support multiple platforms, using Java was a natural choice as it is platform independent.

When it comes to Java programming language and its libraries, there are a few good virtual file system interfaces that are available.

*Apache Commons Virtual File System* (Commons VFS) is one of them. It is a single consistent API for accessing numerous file system types and has a lot of mature features like caching, event delivery, integration support, etc [The Apache Software Foundation 2016a].

*Eclipse File System* (EFS) is another solution that is used mainly to store data in the Eclipse workspace [Eclipse Foundation 2017]. It is an API in the Eclipse platform to abstract details of the concrete file system. Although open and freely extensible it is not widely used outside the Eclipse environment, as it forces file system implementations to fit into narrow concepts for the Eclipse Workspace [Blewitt 2006].

Besides these two libraries, Java NIO.2 File System was released with Java 1.7 as a standard interface with attempt to modernise and improve native Java support for filesystem access. It is comparable in features with the previous libraries, but its late arrival is the main cause for its slow adoption [Leonard 2012].

2.2 Version Control Systems

All previously mentioned file system interfaces are missing the ability to keep track of changes in the data and meta-data. To address this, we looked at version control systems (VCSs).

A version control system is a system that records changes to one or multiple files and can recall specific versions. Although the main usage comes from versioning of software source code it can be applied to nearly any type of files.

The first VCS implementations were local version control systems that usually kept a simple database with all the changes. One such example is RCS [Free Software Foundation 2015].

To facilitate developer collaboration on projects, centralized version control systems like CVS [Vesperman 2007; Free Software Foundation 2016], Subversion [Fitzpatrick et al. 2008; The Apache Software Foundation 2016b] and Perforce [Wingerd 2006] were implemented. These systems feature a single server that contains all the versioned files, and a number of clients that check out and commit files. Recently, this centralized systems started to be replaced by distributed version control systems [de Alwis and Sillito 2009; Muşlu et al. 2014] mainly because the central server provided a single point of failure for the whole system.

In distributed VCS like Git [Chacon and Straub 2014], Mercurial [O'Sullivan 2009] or Bazaar [Gygerik 2013]) clients fully mirror the whole repository instead of just checking out latest snapshot of few files. These systems allow developers to work offline or during periods when the central server is unavailable. Furthermore, in case of the server crash every client has the complete repository and the data can be easily recovered. Also many of these systems can handle several remote repositories and enable developers to simultaneously collaborate with different groups of people in different ways within the same project.
2.2.1 *Git.* Git is a popular free and open source version control system designed for use by software developers. Although it excels at managing changes to software source code, it can track any kind of content. It is designed to handle both small and large projects very effectively and offers fast performance and flexible workflows [Chacon and Straub 2014].

Git was created by Linus Torvalds to support the development of the Linux kernel. It replaced the previous version control system, BitKeeper, in managing the Linux kernel source code. As BitKeeper provided sophisticated operations that did not exist in other similar software available at the time, Git had to satisfy all the requirements and outperform the competition.

The new system allowed both independent and simultaneous development in private repositories as well as parallel development. The system enabled this without the need for constant synchronization with a central repository, as this could become a development bottleneck. Most of all, the tool allowed multiple developers in multiple locations to work in parallel even if some of them are temporarily offline. Each Git repository holds a complete copy of all the data. In order to achieve fast local updates and network transfer operations, Git uses compression and delta techniques to reduce transfer times and save space.

Git is using a cryptographic hash function to identify objects. This guarantees data integrity and allows trust to be established between all distributed repositories. Data objects stored in the Git repository database cannot be modified once created. As a consequence, the entire history of stored changes cannot be altered. Git also ensures that there is an accountability trail for every change by recording who committed which content.

Git also supports the branching and merging workflow, like most other VCSs, but also goes a step further in promoting and encouraging it mainly by ease of creating new and merging old branches. Being free and open, Git was quickly adopted by the developer community and many other tools were built on top of original functionality.

3. IMPLEMENTATIONS IN SVETOVID

The File Systems service is implemented in Java as an OSGi service, which brings many benefits. It allows other OSGi services in the same container to communicate with no efficiency penalty, as the communication is realised through normal method calls. Also, the service can be imported in other OSGi containers using OSGi Remote Services Specification. Furthermore, the same service can be consumed as a web service by other services and end consumers that can be implemented in any programming language and deployed on any platform.

3.1 Temporal File System

One of the main ideas in the project was to have a temporal dimension for almost all the data in the whole system, not just in the file system. Several interfaces and classes were developed as OSGi bundles to enable this, starting with Period (Figure 1) that represents a time period with a first and last instant, both of which are long values compliant with Unix time, that is the number of milliseconds since midnight January 1, 1970, UTC. An expansion of this interface is the ValuePeriod that attaches a value to a period. The default implementations for these interfaces are immutable since objects of these classes will be passed around quite often and need to guarantee the information they represent has not changed.

These first classes could be viewed as “atoms” in this systems that are used by the more complex classes. The interface TemporalValue defines multiple states of a variable, attached to different periods of time, and during some of them it can even be in a “not set” state, thus representing the change of a single value through time. Similarly there are TemporalSet and TemporalMap interfaces that work in the usual way and represent a set and a map that changes through time. To enable the persistence of the
package org.svetovid.core.util.temporal;

public interface Period {
    public long getFirstInstant();
    public long getLastInstant();
    public Period extendOver(Period period);
    public Period intersectWith(Period period);
}

Fig. 1. Interface Period

data another OSGi bundle was made that connects these temporal concepts with the Java Persistence API (JPA). For instance it features a JPATemporalMap class that is then used in many other classes as the main data storage with key-value pairs.

The implementation of the file system that is discussed here uses this temporal persisted map to store all the properties related to the files it needs to handle. Multiple versions of the files are stored in different root folders. The root folder for a particular moment in time is stored as a property. This way the teacher can define the root folder to change on every new assignment, or on predefined intervals, or even on every save, while at the same time the folder contents at any point is available to the compiler or any other external tool that needs to be run.

3.2 File System with a Git Back End

However, while implementing the mentioned solution some problem were encountered. Namely, any tools and environments that are needed or wanted had to support the new file system. In other words we had to integrate them and implement the needed glue code. This had to be done for each tool in order to support it so adoption was hampered. Because of this a different solution was tried.

Attributing to excellent features and abundant availability of numerous implementations and tools, Git was adopted as a foundation for our file system. Moreover it fulfils all laid out requirements and
its accessibility greatly helps consumers of the service consequently ensuring that it can be employed without difficulties. The only open issue is the choice and administration of the repository server that will be hosting the file systems.

Diverse open and free solutions are always welcomed, but every system brings slightly different concepts and ideas about the design and in our situation more importantly management. To offset this issue an abstraction layer was implemented on top of these servers to reconcile this diversity. This layer enabled our system to work with any type of Git repository. The operations themselves (such as saving a file) are implemented with additional calls to the adequate Git operations. The current version doesn’t fully implement the whole temporal file system interface. It does allow for reverting to older versions of the files, but these are saved as separate operations, therefore the complete history is perserved for future analysis.

3.3 Fused File Systems

The files owned by a student are contained in a virtual file system (Figure 3) that is accessible only from that student’s account. This virtual file system is a sum of three concrete file systems, located on the server. One file system is reserved for each student; content of which can be freely changed by the student. Figure 3(b) shows examples of such virtual file system. Every assignment is also designated to one concrete file system whose files are accessible by students solving the same assignment. This is meant to contain description of the assignment, test data, and other files needed by students working on the assignment. On top of that, there is an additional file system, shared by all the students. The information necessary for all students is stored here.

![Diagram of virtual directory](attachment:virtual_directory.png)

**Fig. 3.** Virtual directory as perceived by student (a), and as stored on server (b)
4. CONCLUSIONS

There are many real world practical problems with practical computer assignments, both with the collection of the solutions and with their grading. In general specialised tools are desirable for many of these tasks. In this paper we present some advantages of having a file system for storing this data that has a temporal dimension. This means that not only the final submitted version is stored, which leads to several advantages. The student could restore an older version of the assignment that was possibly more correct. The teacher can also have a better understanding of the development process of the solution and can evaluate its evolution and not only the final state.

Also, if any irregularities are detected there is the advantage of having a better overview of the whole process by which the student come to the submitted solution. For example, it can be seen whether a student tried many iterations of the solution, each building on the previous and eliminating some errors and implementing more and more of the assignment, or a student just wrote the whole solution and submitted it without ever trying it. However, the detection of cheating itself, whether automatic or manual, is not the responsibility of the filesystem and is left to other tools. The filesystem provides the necessary information to help in the further investigation if there is a suspicion.

The paper also presents two working solutions, one of which has been used in several courses with hundreds of students.

The presented solutions are generic file systems, meaning they are not meant only to keep students’ assignments. However these systems store historic data about the properties and contents of the files, unlike most “normal” file systems, and offer the ability to review or restore previous versions of files. The first implementation (Section 3.1) uses custom built OSGi bundles with temporal properties and Java Persistence API for long term data storage. The other implementation (Section 3.2) uses the widely used version control system Git for data storage. This has the advantage of being easily combined with other tools and environments, since the individual file systems can be exported as regular Git repositories.

Although there is positive feedback for the first temporal file system solution, it is hard to measure the impact on student cheating rates or teacher’s effort. There were a few informal interviews with our colleagues who stated an improvement, and there are plans to conduct a formal survey among both students and teachers in order to have a detailed description of their experiences and how they perceive the advances in all the challenges introduced in this system.

There is always room for improvements with any essential service such as a file system. Both presented solutions have their advantages and disadvantages. Many of them would be solved if the temporal file system would be implemented with a new bundle supported by the Git version control system. There are also options to integrate the file system to be directly usable by Eclipse IDE for courses that require the advanced features that an IDE can provide. Similar could be said for other such programs.

Specific tools could be developed that would use the history saved by this system to point out possible irregularities and give a more in depth view of the evolution of the students’ solutions.

REFERENCES


Scott Chacon and Ben Straub. 2014. *Pro Git* (2nd ed.). Apress, Berkely, CA, USA.


Case Study: Using Model Based Component Generator for Upgrade Projects

PETAR RAJKOVIC, University of Nis, Faculty of Electronic Engineering
IVAN PETKOVIC, University of Nis, Faculty of Electronic Engineering
DRAGAN JANKOVIC, University of Nis, Faculty of Electronic Engineering

To improve development of the medical information system components, we have introduced model based code generation tool. It was of great help when we needed to develop series of components sharing the same set of basic functionalities – predominantly data collection forms. Since we faced many functionality update requests, we tend to check the usability of our model driven approach in cases when legacy components developed under the older version of same programming framework need to be included in our system. In this paper we described the set of updates needed for our code generation components as well as two most important use cases – when existing functionality from legacy system has to be extended and when significant portion of legacy system needs to be adapted and included in new system. Overall conclusion is that model driven approach is still useful even with upgrade projects, but required side effort is higher, especially when the first component in a row is getting adapted.

Categories and Subject Descriptors: H.4.0 [Information Systems]: General; H.5.2 [User Interfaces]: User interface management systems (UIMS); I.6.5 [Simulation and modeling]: Model development
General Terms: Human Factors, Design
Additional Key Words and Phrases: Model driven development, Code generation, Upgrade projects

1. INTRODUCTION AND MOTIVATION

Working on the development of medical information systems (MIS) for a decade and a half, we got experience with many different kind of projects – from simple demonstrational pilots to complex upgrade and the integration with legacy systems. The projects that focus on the upgrade, migration and the integration with other systems brings a complete new set of organizational and technical problems. In this paper, we present the results of our case study focused on the usage of model driven development (MDD) approach in various types of MIS extension projects.

To improve overall MIS lifecycle, we introduced data modeling and generation tools [Rajkovic et al, 2015] that helped us mostly in development of the components sharing the same set of basic functionalities. In addition, we defined a framework around used software development methodologies to choose a proper approach in relation to the type of targeting project [Rajkovic et al, 2016]. Beside both suggested updates proved to be useful with new developments, the real challenges came with the upgrade projects [Gettinger 2012]. In the upgrade projects, we were often faced with the requests to integrate external pieces of software or even to extend them. In some cases, including legacy functionalities into our MIS was required. Since the targeting legacy projects can vary in terms of technology and standards, a lot of work was expected. Luckily in some cases, we noticed many common functionalities within legacy components and decided to try to use our existing data generator tool to help us with software upgrade. For specific parts of the upgrades, we had to develop the additional pieces of code to adapt targeting components to our MIS.

This work is supported by the Ministry of Education and Science of Republic of Serbia (Project number #312-001). Authors’ addresses: Petar Rajkovic, University of Nis, Faculty of Electronic Engineering – Lab 534, Aleksandra Medvedeva 14, 18000 Nis, Serbia, e-mail: petar.rajkovic@elfak.ni.ac.rs; Ivan Petkovic, University of Nis, Faculty of Electronic Engineering – Lab 523, Aleksandra Medvedeva 14, 18000 Nis, Serbia, e-mail: ivan.petkovic@elfak.ni.ac.rs; Dragan Jankovic, University of Nis, Faculty of Electronic Engineering – Lab 105, Aleksandra Medvedeva 14, 18000 Nis, Serbia, e-mail: dragan.jankovic@elfak.ni.ac.rs;

Copyright © by the paper’s authors. Copying permitted only for private and academic purposes.
Since many of upgrades are based on adapter and visitor patterns [Kim et al, 2017] we realized that the set of classes that have to be developed will significantly differ from the windows forms which are the primary output of our code generation tool. We choose to extend our generator tool with template based approach and tried to use common templates as much as possible. The additional focus was put on testing phase. Since we needed to verify that both sides of the system work as expected, we conducted the initial tests using generated test vectors and unit tests. Generator tool used to take the same part of data model and generate both component and tests which can be run immediately (having in mind known potential problems with automatically developed tests [Palomba 2016]).

As it has been stated, the proposed update relies on our existing model based generation tool which proved as efficient in modelling and implementing new components. It has been extended with the new set of templates and functionality. It was important not to improve only development, but also testing phases. As an evaluation, we will demonstrate the process and effects for two, from our point of view, common cases – extending existing functionality with new category of options and integrating parts of legacy project into our information system. It is important to say, that examined cases differ from our MIS in architecture and coding standards, but they used the older version of the programming framework, so they can be assumed as technologically compatible.

2. RELATED WORK

The problem with the integration of legacy systems is well known in the area of medical informatics. Choosing between the integration of new functionalities into legacy system, and including legacy functionalities into the new system cannot be assumed as a “happy flow” [Wiegers 2013], but as an exceptional case within software development. Many different approaches can be found in the literature, and many authors are aiming to contribute for future standardization. Current standards such are [HL7, 2013] and openEHR [Kalra 2005] are well descriptive in terms of entities and attributes, but there is not a clear streamline in the guidelines related to upgrade processes.

When the integration with legacy system is needed, a large group of authors focus on data exchange standards, such is [Khan et al, 2014] and [Sachdeva et al, 2012], avoiding full integration. They presented state of the art adaptive interoperability engine that ensures data accuracy. Generated data exchange models are designed in a way that can easily follow future changes and requirements. Introducing data exchange standards is a good way for a situation when end users tend to continue to use the old system, or even when they need to work with several standards [Schloeffel et al 2006].

Since we had requirements to integrate the old functionalities, we used their approach to map the data from the old system before they are merged to a new one. The additional problem with mapping is the fact that both databases are usually large enough that simple whole-dataset-copy approach requires a lot of time and cannot be used effectively [West 2013]. Thus, while working on mappings we used sample extraction method as suggested in [Pageler et al 2016].

Interesting project is described in [Duftschmid et al, 2013]. The authors tried to integrate some 27 new archetypes into legacy system. They managed to do so with 15 of them, but the amount of work needed was higher than expected. Initially we have an idea to first improve legacy software to match the current standard, but since the integration with the new system was required we decided to go with partial adaptation of existing functionalities and include them with the new software.

Eventually we decided to extract Entity-Attribute-Value items and use them as a data model for our modeling tool as suggested in [Kalra et al, 2007] and [Duftschmid et al, 2010]. For this purpose we used reverse engineering tool of our data modeling framework [Rajkovic et al, 2015]. After identifying main entities and their extension points, we were able to define proper mappings and adaptation scenarios.
3. EXTENDING DATA MODELING FRAMEWORK

Data modeling and code generation tools are the part of our information system development framework for many years. We have started with the development in 2009, and first results were published in [Rajkovic et al. 2010]. Further updates and usage overview are presented in [Rajkovic et al. 2014] and [Rajkovic et al. 2015]. So far, our data modeling framework proved as suitable for the new developments and prototype building. The usage in the upgrade projects was limited due to the lack of testing support. It leads to the set of updates shown in Figure 1. These updates include an additional code generation routine needed to support test generation and immediate automated testing of generated components. The idea was to update template based code generation. The concept is as following: the code generation tool will a load template code file where specific parts are represented by the placeholders. The code generation tool will then search the data model and replace the placeholders with mapped model attributes. This approach give less flexibility comparing with initially defined CodeDOM [Hinkel 2016] definitions, but due to its simplicity, we used to get faster results, especially in the cases when we needed to generate the adapter class various entities from the legacy project.

![Fig. 1 Extended concept of code generation tool](image1)

```csharp
public void ExecuteAddEntityTests(Type entityToTest, List<TestVektor> testVektori)
{
    object form = Activator.CreateInstance(entityToTest);
    IDocumentProperties dokument = form as IDocumentProperties;

    foreach (TestVektor vektor in testVektori)
    {
        try{
            object pregled = dokument.GetPregled();
            object referenceToPregledCollection = pregled.GetType()
                .GetProperty("ParentCollection").GetValue(pregled, null);
            MethodInfo mi = referenceToPregledCollection.GetType().GetMethod("Add");
            mi.Invoke(referenceToPregledCollection, new object[] {pregled.GetType()}
                .Invoke(null, new object[] {pregled});
            _context.SaveChanges();
        }
        catch (Exception ex) { MediisTestAssert.Fail(ex); }
    }
}
```

![Fig. 2 Example of a generic test vector execution method](image2)

The next improvement we needed was the integration with testing tool. Since our MIS is developed under .NET environment, we rely on the standard test suites. Testing tool is a .NET application, designed as a part of our modeling tool that is able to generate test models and test classes and run tests. Tests are immediately executed and results can be prompted or stored and later evaluated. Testing phase itself is even more important when the legacy code has to be included into a new project. After including old functionality, we must ensure that both sides work as expected. Running initial set of tests will speed up component validation and detection of mapping related issues.
Testing tool is designed to run unit, integration and regression tests. Actually, it will execute the code from any loaded library containing test classes. The test classes will load the list of test vectors and run them in the sequence. The example of generic test method for adding new type of medical examination is displayed in Fig. 2. The testing class is instantiated with a reference to the list of test vectors (variable TestVektori) and to the parent entities referencing test entities for patient and medical service. Variable _context references execution context of testing environment including all possible destinations where the data can be stored. For example, when _context.SaveChanges() is invoked, depending on configuration, data can end up either in database and/or in XML repository and/or as test result. In order to properly test generated or imported components, mentioned list of test vectors must contain representative values (Fig. 3). Values can differ significantly depending on data type and the scope. Also, data fields in the model can have various standard values which also have to be considered. Test vector generation process is based on the adaptive random [Shahbazi 2016] [Chen 2004] and model based testing [Jacky 2007] [ElFar 2001], but due to its complexity it will not be presented in details in the scope of this paper.

Fig. 3 Example of generated test vectors

4. CASE A: UPGRADING LEGACY COMPONENT WITH ADDITIONAL FUNCTIONALITY

One of the common requests we faced was upgrading the legacy components. These requests were usually not only about including legacy components into our project, but to extend them by supporting some of the features common for our MIS. One of these common functionalities is action-level configuration. In many cases, old MIS had not any role-based access control. Since new MIS developed by our research group (and called Medis.NET) supports possibility for defining accessibility for each action per single user, the feature was often incorporated into the update.

In this particular case, the users of our targeting institution already had an experience with a piece of software supporting pediatricians. The software allowed them not only options related to medical documents, but also the whole set of administrative actions. The mentioned set consists of actions such are defining new medical records for any department, updating demographic data or even change the existing insurance setup. After standardized MIS systems are introduced in Serbian primary care, this kind of actions were not allowed any longer for medical personnel. Since our client insisted to keep existing legacy functionality running, we needed to integrate it into the environment of our MIS. This kind of upgrade cannot be understood as code-generation friendly and it is easy to
run into the situation that more manual work is needed than in the case when a new form is built. The positive side of this specific case is that target form is developed in the same technology as the base application and can be directly imported to a project.

To achieve this kind of upgrade, we had to go with following steps:
- Define meta model of configuration parameters
- Include target form in the project and map actions with configuration parameters
- Add additional extension points in target form
- Run generator tool to generate configuration class and to update target form

Defining meta model of configuration parameters can be considered as the standard part of the process expected by our model based framework. The class containing configuration parameters would later be used to generate a configuration class whose instances would be loaded later by the MIS configuration tool (Fig. 4).

Fig. 4. Mapping target form with the additional tags and extension points that would be used by code generation tool is the step which requires the most of manual work and time. In this case the initial design of the form helped a lot.

Since every add/edit/delete action was triggered by the button, the direct mapping between Boolean configuration parameter and properties making action button Visible/Enabled can be established. Another part needed for the form that need to be included is the new constructor allowing the form to fit in the new project. Benefits in this case are that end users do not request the migration of only one form, but rather series of similar ones. Then, the adaptation of each of the successive forms will use, as the basis, the same set of templates and analytics tools defined for the first in the row. In our presented case we had total five forms to include. In 4 out of 5 cases we used defined templates without modifications.

5. CASE B: INCLUDING THE SET OF LEGACY FUNCTIONALITIES INTO TARGETING MIS

Including the set of legacy functionalities is another request we had multiple times to deal with. The main problem with this kind of requests is that the set of action during the integration with legacy system cannot be precisely defined in advance. The systems can be developed differently and ability to reuse the existing functionality can be limited.

In this case, we had a request to bring the functionality of existing MIS dedicated for neurologists to our Medis.NET. The existing solution was developed in the manner of document management
systems with clear distinction between layers. For example, all forms were just used for displaying data, business logic for processing and data layer for interaction with database. Since we had to migrate data and to integrate into Medis.NET by keeping complete business logic unchanged, we had to find the best possible way to use our model based data generator and reduce the time needed for development.

The following set of actions were required here:
- Use reverse engineering tool to extract all the entities related specific medical examinations
- Establish connections to entities from Medis.NET and generate new segments of data access layer
- Adapt business logic to match the new entities
- Adapt presentation layer to match the new entities

The alternative for adaptation of business logic and presentation layer would be generation of new components and referencing necessary parts from legacy code. But in some cases, end users would not be willing to accept such solution.

The result of mentioned set of actions is a new project (called Medis.Neuro) referencing legacy libraries. It was built as separate component. This component can be then loaded in Medis.NET using standard approach such as inversion of control and supporting Spring library.

---

One possible configuration of running instance of Medis.NET is shown on Fig. 5. Beside the fact that generated components should be tested and have lower number of potential bugs, all kind of testing has even higher significance than with regular development. Just for the illustration, basic instance of Medis.NET running for primary care institutions has around 160 different data collection forms supporting different medical examinations. The set of functionalities included from old solution for neurologist consists of 146 different forms and significant business logic supporting many different decision making routines and calculations. So, after such an extension get introduced into information system, it is not simple unit and integration testing that must be done, but also performance check on the amount of resources used.

6. RESULTS AND DISCUSSION

Our model based modeling and data generation tool proved to be useful when new parts of the software have been developed. We managed to significantly reduce used time especially until first prototype is done. In some cases we managed to reduce expected time to one third [Rajkovic et al, 2015]. Also, due to the generation of tested components we experienced lower number of bugs.
The story of using model based generation tools with upgrade projects is a bit different. Since significant adaptation is usually required, we could not achieve results as with newly developed. The realization of both presented cases was done with the help of MDD approach. Due the high level of customizability, running the process for the first component in the row will last even longer than standard development. We had an opportunity to test both cases of upgrade requests on several different types of components so we are able to present some relevant results.

The first of the effects that we get from our framework is reducing the time needed for component development (Table I). We are displaying the time needed for each of the steps in component development workflow. The same steps were applied both in cases when no optimization is used, and when we use MDD approach. The data that we present is gathered as a result of surveying our development team members. We cannot make them general, but they are indicative enough to compare the results with different approaches. As the basic measurement unit we define T, which is a time that needed to define the structure of one entity and to create corresponding database table.

<table>
<thead>
<tr>
<th>Process</th>
<th>Standard development</th>
<th>MDD – first in the row</th>
<th>MDD – with developed template components</th>
</tr>
</thead>
<tbody>
<tr>
<td>DB table definition</td>
<td>T</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Defining class in object model</td>
<td>T</td>
<td>T</td>
<td>2T</td>
</tr>
<tr>
<td>Developing/Adapting visual form</td>
<td>6T</td>
<td>10T</td>
<td>12T</td>
</tr>
<tr>
<td>Form specific logic</td>
<td>2T</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Defining tests</td>
<td>T</td>
<td>T</td>
<td>2T</td>
</tr>
<tr>
<td>Testing</td>
<td>T</td>
<td>2T</td>
<td>4T</td>
</tr>
<tr>
<td>Overall time</td>
<td>12T</td>
<td>14T</td>
<td>20T</td>
</tr>
</tbody>
</table>

Pursuing new development of data forms without help of MDD takes a lot time to complete, as we estimated this to 12T. Using MDD, we estimated this the time reduction to 20-30% (we set it to 3T). The most significant time spent in this case is implementation of form specific logic which cannot be derived from other places. Theoretically, this can vary a lot, but we estimated it to 2T.

Working on upgrade projects and component integration is significantly time consuming with help of MDD we estimated necessary time to 14T for case A and 20T for case B. The most time is consumed while writing the adaptation classes and methods. Defining classes for object model takes more time than with regular development because developer cannot use results of reverse engineering tool directly, but needs to adapt retrieved list of fields introducing new key constraints and defining extension template. Extension template is used for any consecutive component to replace its key components. This is applicable for case B. For the case A it is not since we still need to define separate mapping for each of the forms.

We choose these two cases to be, by our opinion, representative. Time needed for adaption of any of succeeding forms could vary from case to case, but for these implementing the same interfaces it is much easier to reuse any of developed mappings. If we compare time needed to develop series of components, case B proved even better results than new builds after 10th in the row (39T for new build, 38.9T for adapted). Case A always requires more time than new build since not only mapping on data is needed, but also unpredictable custom mapping on actions. Case B initially required the most of the time, but after 5th adapted form in the row case B started to be more effective from the point of time consumption.
7. CONCLUSION

Including parts of legacy code in one’s software project is scarce on every programmer’s wish list. Dealing with code developed under different standards, architecture and even different technology is not an easy task to be accomplished. In this paper we wanted to show the overall positive effects we got from using our model based code generation framework in upgrade projects.

We examined two cases – when existing functionality from legacy system has to be extended (case A) and when significant portion of legacy system need to be included in our system (case B). In both cases, significant time is needed when first component is developed. During that time, all necessary custom templates are built allowing effective use of code generation tool for next instances.

Even with this approach, MDD significantly saved time in cases when many components with common functionality had to be generated. In cases when target components differ significantly, as well as have different usage, model driven cannot be of much help. The MDD approach has its value even for this kind of projects, and it can be used in more effective way with series of similar components.

REFERENCES

Containerized A/B Testing

ÁDÁM RÉVÉSZ and NORBERT PATAKI, Eötvös Loránd University, Faculty of Informatics

Software version ranking plays an important role in improved user experience and software quality. A/B testing is technique to distinguish between the popularity and usability of two quite similar versions (A and B) of a product, marketing strategy, search ad, etc. It is a kind of two-sample hypothesis testing, used in the field of statistics. This controlled experiment can evaluate user engagement or satisfaction with a new service, feature, or product. A/B testing is typically used in evaluation of user-experience design in software technology. DevOps is an emerging software methodology in which the development and operations are not independent processes, they affect each other. DevOps emphasizes the usage of virtualization technologies (e.g. containers). Docker is widely-used technology for containerization. In this paper we deal with a new approach for A/B testing via Docker containers. This approach is DevOps-style A/B testing because after the evaluation the better version remains in production.

Categories and Subject Descriptors: K.6.3 [Management of Computing and Information Systems]: Software Management—Software selection; H.5.2 [Information Interfaces and Presentation]: User Interfaces—Evaluation/methodology; D.2.9 [Software Engineering]: Management—Software Management

General Terms: Software Quality Analysis with Monitoring

Additional Key Words and Phrases: Docker, containers, DevOps, A/B testing

1. INTRODUCTION

Nowadays A/B testing plays an important role in evaluation of different but very akin user-experience design. It is widely used among online websites, including social network sites such as Facebook, LinkedIn, and Twitter to make data-driven decisions [Xu et al. 2015]. A/B testing is an important method regarding how social media affects the software engineering [Storey et al. 2010]. A/B testing has been applied in many web portals successfully [Kohavi et al. 2009].

A/B testing is a powerful method because it is based on the behavior of the end-users. A/B testing of webpages or webapplications requires small changes in the user-experience design (e.g. colors, structure of a page, shape of buttons, etc.). As visitors are served either the control or variation, their engagement with each experience is measured and collected. It can be determined whether changing the experience had a positive, negative, or no effect on visitor behavior from the collected info. A/B testing of these applications takes time for collecting proper number of feedback.

Containerization is new directive in virtualization: this lightweight approach supports operating system-based isolation among different pieces of the application [Soltesz et al. 2007]. Containerization is on the crest of a wave since Docker has been developed. Docker provides a systematic way to automate the faster deployment of Linux applications inside portable containers. Basically, Docker extends casual Linux containers (LXC) with a kernel-and application-level API for improved isolation [Bernstein 2014]. Docker is emerging tool to start complex application in many virtual operating
system-separated parts and it ensures the communication among them. Docker has a comprehensive documentation [Docker Inc. 2017].

Docker containers are built up from base images, there are general images (e.g. Ubuntu 16.04) and specific images (for Python run environment). Dockerfiles describe how an image can be created and Docker is able to generate the image and save it to the repository. Many services of the Docker platform are available (e.g. Docker Engine, Docker Compose, etc.). The images are in Docker Registries. The Docker Engine is responsible for managing containers (starting, stopping, etc.), while Docker Compose is responsible for the configurations of containers on a single host system. Docker Compose is mainly used in development and testing environments. One can define which services are required and what are their configuration in the container. Docker Compose files can be created for this purpose. Orchestration of Docker containers are typically executed with Kubernetes or OpenShift [Vohra 2017].

Continuous Delivery (CD) is a software development discipline. This methodology aims at building software in such a way that the software can be released to production at any time. It is a series of processes that aims at safe and rapid deployment to the production. Every change is being delivered to a production-like environment called staging environment. Rigorous automated testing ensures that the business applications and service work as expected. Every change had been tested in staging, so the application can be deployed to production safely.

The DevOps approach extends the CD discipline and focuses on comprehensive CD pipelines: starts with building and followed by different kinds of testing [Schaefer et al. 2013]. Unit testing, component testing, integration testing, end-to-end testing, performance testing, etc. should be performed on the software [Roche 2013]. In the meantime, static analyser tools try to find bugs, code smells and memory leaks in the source code. 3rd-party compliance should be checked in the build pipeline. Automated vulnerability scanning of the software is mandatory to discover security gaps. The visibility of the whole process is guaranteed.

After this phase, the automatic deployment of application starts. Application Release Automation (ARA) tools are available that can communicate with the CI server and the deployment steps can be designed on a graphical user interface of these tools. The DevOps culture argues for the deployment automation at the level of the application [Cukier 2013]. The automatic upgrade and roll-back processes involve many difficult changes. Database schemas, configuration files and parameters, APIs, 3rd-party components (e.g. message queues) may be changed when a new software version is released. The deployment process has to cover these changes as well and requires automation and visibility.

DevOps considers the monitoring and logging of the deployed application in the production environment [Lwakatare et al. 2015]. The development team is eager for feedback from the application which is in the production environment: e.g. what are the unused features in the software, memory or other resource leak detection or performance bottlenecks. ELK-stack is a popular toolset for this purpose [Lahmadi and Beck 2015]. Elasticsearch is a distributed search and analytics engine, Logstash is a data processing pipeline and Kibana is responsible for the visualization. Docker out of box supports some logging drivers such as JSON log driver and GELF log driver to handle the log streams of each container. With GELF log driver the container logs can be forwarded to an ELK stack. Graylog Extended Log Format (GELF) is understood by most of the log aggregating systems like Logstash or more obviously Graylog. Developers have to get as much information as possible to be able to take care of a trouble [Prakash et al. 2016]. Problems may cause automatic roll-back of the application to the previous stable version in a seamless way. The analysis of logs and monitoring data is application-specific and their evaluation may be difficult. Therefore, using big data analysis and machine learning shall be involved.

In this paper we argue for a new DevOps-style A/B testing for an automated, user experience-based approach. We take advantage of logging and monitoring features to get feedback from the end-users.
Our approach works in Docker containerized realm, thus the webapplications and every tool which are used in the evaluation run in containers. After the specified duration the A/B test is evaluated and winner version of webapplication remains in the production environment automatically.

This paper is organized as follows: in section 2 we present our A/B testing approach from high-level and go into implementation details per components in section 3. Finally, this paper concludes and presents future work in section 4.

2. OUR APPROACH

We propose an approach for A/B testing of webapplications in Docker containerized way. This approach takes advantage of Docker, Nginx server, ELK stack and GrayLog. We have developed a script for controlling the A/B testing. This script is written in Python.

The two variants of the same webapplication are running in separate set of containers. The Nginx server is also running in container. The Nginx is routing the users to A or B version based on their IP hash. On the client side of both webapplications HTTP requests are submitted to the Nginx server. Two kinds of requests are in-use. The first one is a periodical one which states if the user still using the application. The second one is triggered by the end-user to check the user's activity. Both requests contains the origin of the application version as tag. We collect the logs of webapplications in an ELK-stack.

The Python script runs on the host machine. The script takes a duration parameter that specifies how long the A/B test is running. When this duration expires the script gets log information from the ELK-stack container and evaluates which version is the better one. The script controls the Docker to discontinue the running of the worse version and replaces it with the better one.

3. TECHNICAL DETAILS

3.1 Client side

For our research we created two versions of a simple website with different title and headlines clearly indicating which version we are looking at using our web browser. Both version has a link.

The page also contains a JavaScript script which acts like a subset of any other webpage analytics bundle. It generates a UUID on every page load and sends a HTTP GET request to the '/ping' route sending the generated client id as parameter in every 5th second. By these messages we can set a metric which can indicate how long our user stays on the page. We also send a HTTP GET request containing the client id on the click event of the link to the '/click' endpoint.

We do not create any relation between those UUIDs and session cookies to keep it anonymous like a good analytic tool anyway. We did not want to make any unnecessary (A or B) version specific code in the web page (nor the backend) because it could pollute the source code of the product itself and can be irrelevant of its aspects.

3.2 Backend side

The backend simply serves a static HTML file (which contains all of the client side code) and responds with status 200 on every request at route '/click' and '/ping'. It dumps every request to the standard output. All of these configurations have been done in a single nginx.conf file to keep this proof of concept project simple.

3.3 Docker containers

First of all, our testing stack has a load balancer container which listens on port 80 and forwards the requests to node1 or node2. The forwarding depends on the client IP hash in order to make sure the
clients click and ping requests are being forwarded to the very same node which served the HTML file earlier (so the load balancer will not switch up the version between two requests from the same client).

The load balancer references Node1 and Node2 by their aliases. Docker Engine has a solution to create virtual networks between containers so when there are multiple products up and running containers on the same host machine they do not interfere with each other connecting to separate virtual networks. Docker compose takes care about creating a network for our project defined in a `docker-compose.yaml` file (see below) by default. This default network is created with the name of the containing folder (assuming it is the same as the project name) with a `_default` prefix. This came handy when we created a new container and connected it to the same network by hand on the end of the test evaluation.

The Docker Engine takes care about DNS services on the virtual network that is why we can reference containers by their names. We do not need to change configurations on every startup and we do not have to save IP addresses in environment variables or hosts files on containers. It is more dynamic and more secure.

```yaml
error_log /dev/stdout info;

events {}

http {
    access_log /dev/stdout;
    upstream abtest {
        ip_hash;
        server node1;
        server node2;
    }

    server {
        listen 80;

        location / {
            proxy_pass http://abtest;
        }
    }
}
```

Node1 only differs from Node2 in its `index.html` file and more importantly in its tag. Node1 has “version-a” tag while Node2 has “version-b” tag at the beginning. The version tag is also sent in every log message to the Graylog server providing the identity of the version. As shown at the code snippet below Node1 and Node2 have not got any open ports they can receive requests only through the load balancer.

As we mentioned earlier the backend prints all of its requests to the standard output. The standard output is forwarded in GELF format to the GELF server.

```json
version: '3'

services:
    loadBalancer:
        image: nginx
```
3.4 Log aggregation

There are numerous ELK stack configurations available on the Docker community hub so we omit the details for now. We have a Graylog server up and running which receives the logs of Node1 and Node2. We have set up an extractor which checks the message property of the log and uses regular expression to extract ‘click’ or ‘ping’ from the request route to a separate field called `clientLogEvent` when it is present and an other extractor works in the same way and extracts `clientSessionId`. Making extractors and testing queries on the Graylog web interface is comfortable and can be done without the need of digging in Elastic search querying. It is suitable for anyone who wants to shape it to fit their own specific A/B test scenario.

3.5 Evaluation and replacement

We have decided this task has to be done on a host machine by a script which can interact with the Docker Engine (or Swarm, Kubernetes, etc). For security reasons we cannot (neither want to) give a container access to other containers on system level.

We have chosen Python as the most suitable script language for this task. Python has maturity, and most of the *nix boxes have Python environment pre installed and another good reason is that Docker has a solid Python SDK, actively used by the Docker Compose project.

In our example we have decided to measure the count of click metric ('/click' route requests) – the bigger the better. When we exceed the duration of the test the script sends one query per version to
the Graylog server API to count its clicks (clientLogEvent: click). We use Apache Lucene syntax for queries. The script compares the result and then with the power of the Docker SDK shuts down the loser version Node and replaces it with an instance of the winner version.

The Python script itself interacts with the Graylog Web API and gets a session token by sending login credentials. At this point we could use API tokens set up on the Graylog Web interface, but we have not wanted to increase the complexity of the configuration for this example. The query is sent to the Graylog REST API, but it is just like any other REST API call so we omit the details for now. The interesting part is how we replace the container running the worse version with a new container running the better one. We stop and remove the “loser” container at first to avoid naming conflicts later on. After that we create a new container with the same parameters as the “winner” container, but with the name of the loser one. We connect the new container to the projects network using the same alias as the removed container had. When we start up the new container the load balancing works the same as before and the new node can be reached by the same name its predecessor could be reached by.

```python
loser = client.containers.get(loserContainerName)
loser.stop()
loser.remove()

newNode = client.containers.create('nginx',
    name = loserContainerName,
    volumes_from = [winnerContainerName],
    log_config = {
        'driver': 'gelf',
        'options': {
            'gelf-address': 'udp://127.0.0.1:12201',
            'tag': winnerTag
        }
    }
)

bridgeNetwork = client.networks.get('bridge')
bridgeNetwork.disconnect(loserContainerName)

testNetWork = client.networks.get(self.networkName)
testNetWork.connect(
    loserContainerName,
    aliases = [loserContainerName]
)
newNode.start()
```

The script is just a proof of concept but we have created a command line interface for it because we have created some parameters so we can test it on different setups. Its help text tells us what parameters we can use for our test.

```
$ abtestCli -h
usage: abtestCli.py [-h] [--duration DURATION] [--aTag ATAG] [--bTag BTAG]
                    [--networkName NETWORKNAME] [--apiAddress APIADDRESS]
                    [--apiUser APIUSER] [--apiPass APIPASS]
aName bName
```
A CLI tool for runtest

position arguments:
  aName
  bName

optional arguments:
  -h, --help            show this help message and exit
  --duration DURATION   
  --aTag ATAG           
  --bTag BTAG           
  --networkName NETWORKNAME
  --apiAddress APIADDRESS
  --apiUser APIUSER
  --apiPass APIPASS

3.6 Running
Assume that we have the docker-compose.yaml file in our currently working directory.

    $ docker-compose up -d

After it started up our services we only have to start our Python CLI script. It has three mandatory parameters:
(1) Duration – in ISO 8601 duration format
(2) A version container name
(3) B version container name

    $ abtestCLI.py PT30M ab_node1_1 ab_node2_1

After thirty minutes the script will log the name of the better version and replace the worse with it.

4. CONCLUSION
A/B testing is a powerful method to improve software quality and user experience. It gains feedback from two akin versions of the same product (software, search ad, newsletter email, etc.) and it measures the end-user engagement.

We have developed an approach and related tools for executing A/B testing in Docker containerized environment. Our proof of concept implementation is working and has fulfilled our expectations but there is a lot of work to do and a numerous of choices to make before it becomes production ready. One of our goals was to keep the stack and the implementation simple to leverage the understanding of the conception.

We have mentioned that Docker Compose is for single host development and testing. And it did a great job providing us an initial state for our services. Also we have met the limitations of it such as dynamic configuration. Assume that we use the same stack, the A/B test is over and there are winner version containers everywhere then our system shuts down. Since Docker Compose cannot persist configuration changes to its compose file our configuration will be restored to the original one on the next docker-compose up command. There are great configuration management software tools like Puppet or Chef [Spinellis 2012]. Of course when it comes down to scalability we have to use Docker Swarm or Kubernetes client libraries, etc for managing version replacement on a multi-host system.
The concept is proven and we are excited to set it working on enterprise level. There could be a great A/B test deployment service on Amazon AWS or Microsoft Azure. Those companies have resources and technology to create a powerful analytics system with an integrated automatic deploy solution.

REFERENCES

David Bernstein. 2014. Containers and Cloud: From LXC to Docker to Kubernetes. *IEEE Cloud Computing* 1, 3 (Sept 2014), 81–84. DOI: http://dx.doi.org/10.1109/MCC.2014.51


Abdelkader Lahmadi and Frédéric Beck. 2015. Powering Monitoring Analytics with ELK stack. 9th International Conference on Autonomous Infrastructure, Management and Security (AIMS 2015). (June 2015). https://hal.inria.fr/hal-01212015


Energy Consumption Measurement of C/C++ Programs Using Clang Tooling

MÁRIO SANTOS and JOÃO SARAIVA, University of Minho
ZOLTÁN PORKOLÁB and DÁNIEL KRUPP, Ericsson Ltd.

The green computing has an important role in today’s software technology. Either speaking about small IoT devices or large cloud servers, there is a generic requirement of minimizing energy consumption. For this purpose, we usually first have to identify which parts of the system is responsible for the critical energy peaks. In this paper we suggest a new method to measure the energy consumption based on Low Level Virtual Machine (LLVM)/Clang tooling. The method has been tested on 2 open source systems and the output is visualized via the well-known KcacheGrind tool.

Categories and Subject Descriptors: B.8.2 [Performance and reliability]: Performance Analysis and Design Aids—Energy consumption measurement; H.5.2 [Information Interfaces and Presentation]: User Interfaces—Evaluation/methodology

General Terms: Energy consumption, Human factors

Additional Key Words and Phrases: Energy consumption, Code Instrumentation, Visualization

1. INTRODUCTION

While in the previous century computer manufacturers and software developers primary and single goal was to produce very fast computers and software systems, in this century this has changed: the widespread use of nonwired but powerful computer devices is making battery consumption/lifetime the bottleneck for both manufacturers and software developers. Unfortunately there is no software engineering discipline providing techniques and tools to help software developers to analyze, understand and optimize the energy consumption of their software! As a consequence, if a developer notices that his/her software is responsible for a large battery drain, he/she gets no support from the language/compiler he/she is using. The hardware manufacturers have already realized this concern and much work in terms of optimizing energy consumption by optimizing the hardware has been done. Unfortunately, the programming language and software engineering communities have not yet completely realize that bottleneck, and as consequence, there is little support for software developers to reason about energy consumption of their software. Although is the hardware that consumes energy, the software can greatly influence such consumption [Bener et al. 2014], very much like a driver that operates a car influences its fuel consumption.

In this paper we introduce an automated instrumentation-based method to measure the process level energy consumption for C/C++ programs. The source code is compiled by our Clang tooling based compiler to produce an instrumented code. The generated executable will measure the energy con-
sumption and emit the results in machine usable format. We convert the output to the well-known Kcachegrind [Weidendorfer 2015] format.

This paper is structured as follows: In Section 2 we overview how we can measure the energy consumption of a processor. Our code instrumentation is based on the LLVM/Clang compiler infrastructure, which we discuss in Section 3. In Section 4 our instrumentation is described in details. The results are evaluated in Section 5. We overview the related work in Section 6. Our paper concludes in Section 7.

2. ENERGY MEASUREMENT ON PROCESS LEVEL

In this section we overview how one can measure the energy consumption of a processor using the Intel’s RAPL interface. We also discuss our extensions to retrieve the necessary information in function-level.

Originally designed by Intel, RAPL (Running Average Power Limit) [Dimitrov et al. 2015] is a set of low-level interfaces with the ability to monitor, control, and get notifications of energy and power consumption data of different hardware levels. It is supported in today’s Intel architectures, like i5 and i7 CPUs. The architectures, that support RAPL, monitor energy consumption information and store it in Machine Specific Records (MSRs). These MSRs can be accessed by the Operating System. As you can see by Fig. 1, RAPL allows energy consumption to be reported in a practical way, by monitoring CPU core (pp0), CPU uncore (pp1) and DRAM separately.

Our extension of RAPL for C (CRAPL), can be viewed as a wrapper (example code below) to access the MSRs during the execution of a C/C++ programme. Through this interface we are able to have
an estimate of the power consumption in order to study which components (methods\functions) have absurd energy spikes in our source code.

```c
CRapl rapl = create_rapl(0);
rapl_before(rapl);
doSomething();
rapl_after(0, rapl);
```

3. CLANG TOOLING

The LLVM project, started at the University of Illinois, is a collection of modular and reusable compiler and tool-chain [Lattner 2006]. LLVM has grown into an umbrella project and now includes various open source activities from compilers to static analysis. The flagship compiler for the LLVM project is Clang, the “native” compiler of LLVM. Clang supports C, C++, Objective-C and Swift languages in the advanced level [Groff and Lattner 2015]. The modular, object-oriented design of Clang make it ideal for research projects require compiler-level understanding of the source code [Lattner 2008]. Having a well-defined interface for building the Abstract Syntax Tree (AST), exploring it in various ways and even on-the-fly modify it, we can apply the tool-chain for instrumenting the source.

In the center of our activity is the Abstract Syntax Tree (AST). The AST contains all important information (even the formatting informations via the stored positions of every element). The structure of the AST is representing the logical structure of the original program. For example the node which belongs to a for loop has four children: a declaration statement to introduce the loop variable, a logical expression as loop condition, an iteration expression and the body. Note that the parentheses and the semicolons in the loop header are excluded.

In the AST there are different type of nodes such as ForStmt, FunctionDecl, BinaryOperator, etc. These types are organised to an inheritance hierarchy which has three roots: Decl, Stmt and Type. Since the fundamental part of build process is compilation of translation units, the type of the root node is TranslationUnitDecl.

One way of using the Clang AST is to visit its nodes [Horváth and Pataki 2015; Clang 2016]. The visitor design pattern can be used to reach every node of the tree and perform some action when the process comes to a given type of node. Clang compiler provides a very efficient way of tree traversal by RecursiveASTVisitor template class. Our visitor class has to inherit from this template class of which the template parameter is our class itself. The reason of this is that with this solution our class also becomes an AST visitor by the inheritance, but we do not have to pay for virtual function calls every time when running the given visitor function for the next AST node.

4. THE INSTRUMENTATION

Based on LLVM/Clang, our LibTooling tool starts by reading the input files and will run them up our FrontendAction. It will create an Abstract Syntax Tree (AST) with the parsed text of each file. For each of these generated trees we will recursively go through each node so we can make the necessary modifications to include our CRapl interface in the source code of the program that we want to analyse.

The main nodes to be visited are the following:

— **VisitFunctionDecl**: It visits all the nodes that are functions. If the analysed function does not refer to a header file and has the minimum number of statements (lines of code) that the user requested (\(-l = N\)) then the tool will insert the information of that function into an index (array) of functions that will suffer the respective modifications until the end of the recursive reading of their child nodes.
VisitIfStmt: to maintain code consistency we need to insert braces in each If or Else statement that they are not already limited by them. This because we always need to insert a rapl_after before every single return statement of the given function.

VisitReturnStmt: If it catches a return statement anywhere in the code for that function, it will insert a "rapl_after" to end the analysis of the power consumption in that call.

VisitCallExpr: When we tested for the first time the totality of the Plugin (Instrumentation + CRapl) we verified that there were functions to consume more energy than the main function itself, which is impossible since main is the first to be executed and the one that finishes the program. With this we realized that we were not handling recursive functions in the best way. So the best solution we found for this was to limit blocks of code before and after the recursive call (i.e rapl_after and rapl_before instrumentations for each of these calls). This was one of the biggest challenges until we came up with a good generic solution, regardless of what kind of recursive call it is.

In the end of crossing each tree, it will also insert the dependencies of the CRapl libraries and save the changes in the corresponding file (or create a new case the -o = "example.cpp" flag is enabled).

When it finish all the instrumentation of the files, it will create the index.txt file with all the information of all the modified functions so that later they will be analysed by the CRapl:

```
0:/home/name/ClangRapl/xerces-c/tests/src/XSTSHarness/XSTSHarness.cpp:102:1:printFile
1:/home/name/ClangRapl/xerces-c/tests/src/XSTSHarness/XSTSHarness.cpp:138:1:error
2:/home/name/ClangRapl/xerces-c/tests/src/XSTSHarness/XSTSHarness.cpp:145:1:fatalError
3:/home/name/ClangRapl/xerces-c/tests/src/XSTSHarness/XSTSHarness.cpp:240:1:main
```

5. EVALUATION

After the program execution, a file will be written with the values of the energy consumed by each function and the number of times it was called.

Currently our tool has been tested on two projects, tiny xml (small) and some xerces-c-3.1.4 (medium) samples / tests, both XML parsers written in C++.

Due to a high number of calls from each function (total of 181889), we received a huge overhead from CRapl. In order to remove this overhead first we would have to test only the main function to collect the true value of the energy consumed by the program and then run our script in python (mainly converts the format of the output file of the CRAPL in the Callgrind format to be read by Kcachegrind) to do the necessary normalizations (rule of three simple) of the energy results.

In this phase we will mainly present the results with this overhead, and since the Kcache grind reads only number interns the units will be presented in Milijoules \((10^{-3} \text{ J})\). mJ.

As we can see from Figure 2 when we run the CRapl in all functions of tinyxml we get a huge overhead (the number of calls to rapl is too big) but being the main objective of the project to realize which are the functions that spend more energy this process is necessary. Regardless, overhead does not prevent us from doing an analysis on the project (by percentage of energy consumed per function). We can verify that the Parser function (at line 1043) and its child nodes (including other functions) represent about 97% of the energy (package) consumed during the execution of the xmltest. Since the Parser function uses recursion, which means that at the beginning of each function you have to push the arguments to the stack and at the end pop them, it causes the function to have a longer execution
energy and consequently a higher consumption of energy [Fakhar et al. 2012].

With about 4926 instrumented functions we performed 6 tests of the xerces project and these were the results (package energy) obtained for some random functions with the overhead (in Joules):

<table>
<thead>
<tr>
<th>Functions</th>
<th>Package Energy (J)</th>
<th>%</th>
<th>Calls</th>
<th>Energy/Calls (J)</th>
<th>Time (S)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Main:840</td>
<td>973</td>
<td>100%</td>
<td>1</td>
<td>973</td>
<td>90</td>
</tr>
<tr>
<td>testRegex:5393</td>
<td>569.3</td>
<td>58%</td>
<td>1</td>
<td>569.3</td>
<td>55</td>
</tr>
<tr>
<td>matches:517</td>
<td>542.9</td>
<td>56%</td>
<td>84</td>
<td>6.5</td>
<td>52</td>
</tr>
<tr>
<td>match:995</td>
<td>399.2</td>
<td>41%</td>
<td>52448</td>
<td>0.008</td>
<td>39</td>
</tr>
<tr>
<td>Initialize:162</td>
<td>329.5</td>
<td>34%</td>
<td>1</td>
<td>329.5</td>
<td>30</td>
</tr>
</tbody>
</table>

From these results (Table I) we can not draw great conclusions except that most of the functions with high energy consumes are child nodes of the Initializer:162 because it is a function with only one call and with the highest percentage of energy spent. Also we can verify that the best and most interesting case to be analyzed in detail is the DTest because of the irregularity of the function match:995.

We will show some results about the most expensive functions in this test:

<table>
<thead>
<tr>
<th>Functions</th>
<th>Package Energy (J)</th>
<th>%</th>
<th>Calls</th>
<th>Energy/Calls (J)</th>
<th>Time (S)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Main:840</td>
<td>973</td>
<td>100%</td>
<td>1</td>
<td>973</td>
<td>90</td>
</tr>
<tr>
<td>testRegex:5393</td>
<td>569.3</td>
<td>58%</td>
<td>1</td>
<td>569.3</td>
<td>55</td>
</tr>
<tr>
<td>matches:517</td>
<td>542.9</td>
<td>56%</td>
<td>84</td>
<td>6.5</td>
<td>52</td>
</tr>
<tr>
<td>match:995</td>
<td>399.2</td>
<td>41%</td>
<td>52448</td>
<td>0.008</td>
<td>39</td>
</tr>
<tr>
<td>Initialize:162</td>
<td>329.5</td>
<td>34%</td>
<td>1</td>
<td>329.5</td>
<td>30</td>
</tr>
</tbody>
</table>

As we can see in Table II, the function match:995 has a very high power consumption compared to the other tests due to be used many times (52448). So, from this table we can say that the function match:995 is a function that spends less energy in relation to the rest (only 0.008J). Knowing that
this function is used mostly by the function matches:517 we can conclude that 70-75% of the energy consumed by matches:517 comes from match:995.

6. RELATED WORK

The energy consumption of software systems is a concern for computer manufacturers, software developers and regular users of computer devices (include users of mobile (phone) devices). While computer manufacturers began developing energy efficient hardware since the wide adoption of non wired computers, only recently energy became a concern for software developers as shown by the questions addresses on stack overflow report in [Pinto et al. 2014]. In fact, nowadays the energy efficiency of software systems is an intensive area of research. CPU manufacturers already provide frameworks to analyse the energy consumption of their processors, namely the energy estimators provided by Intel - the Intel RAPL [Dimitrov et al. 2015; Weaver et al. 2012; Hähnel et al. 2012; Liu et al. 2015; Fu et al. 2015] - or by Qualcomm - the Qualcomm TrepN framework [Qualcomm 2014; Bakker 2014]. Together with the use of external energy measurement devices, such as [McIntire et al. 2012; Bessa et al. 2016], it is possible to instrument and analyse the energy consumption of software systems.

Indeed, several techniques have been proposed to reason about energy consumption in software systems. For example, in the area of database systems, one of the first approaches to evaluate the energy consumption is presented in 2009 on the Claremont report [Agrawal et al. 2008], which expressed clearly the importance of taking into account the energy consumption from the very beginning of designing a database system. In [Goncalves et al. 2014] it was presented a technique to infer the most energy efficient query execution plan.

Researching and designing energy-aware programming languages is an active area [Cohen et al. 2012]. For example, there are works to analyse the energy efficiency of Java and Haskell data structures [Pereira et al. 2016; Lima et al. 2016; Pinto et al. 2016], to analyse how different coding practices influence energy consumption [Sahin et al. 2014a], energy aware type systems [Cohen et al. 2012], and to study the impact of code transformation [Brandolese et al. 2002], code obfuscation [Sahin et al. 2014b], and testing techniques [Li et al. 2014] software energy consumption. Other researchers have defined techniques to analyse energy consumption in Android mobile applications [Nakajima 2013; Couto et al. 2014; Li and Mishra 2016].

7. CONCLUSION

Our tool can be a good complement for C/C++ programmers who are interested in reducing the energy consumption of their programs. This is a theme that may not emerge much when small programs are used by a single machine but can have a positive environmental impact, or even reduce energy costs economically, when we talk about large scale projects used in servers or millions of personal computers around the world. In summary, we can say that the results are quite convincing in terms of high energy consumption by recursive functions and we can also notice that in sequential programs, the longer it takes a function to perform, the higher its energy consumption. As future work it would be interesting to shape CRAPL so that it would pick up the paths of the executed functions, so that we could study the results in more detail through the visualization of more illustrative graphs in KcacheGrind. In addition, collection of energy measurements from standard libraries functions would also give us a good perspective on which functions we should choose depending on how much data our implementation will receive.
ACKNOWLEDGMENT
This work is financed by the ERDF European Regional Development Fund through the Operational Programme for Competitiveness and Internationalisation - COMPETE 2020 Programme and by National Funds through the Portuguese funding agency, FCT - Fundação para a Ciência e a Tecnologia within project POCI-01-0145-FEDER-016718

REFERENCES


Zhangjie Fu, Xingming Sun, Qi Liu, Lu Zhou, and Jiangang Shu. 2015. Achieving efficient cloud search services: multi-keyword ranked search over encrypted cloud data supporting parallel computing. IEICE Transactions on Communications 98, 1 (2015), 190–200.


Vincent M Weaver, Matt Johnson, Kiran Kasichayanula, James Ralph, Piotr Luszczek, Dan Terpstra, and Shirley Moore. 2012. Measuring energy and power with PAPI. In *Parallel Processing Workshops (ICPPW), 2012 41st International Conference on*. IEEE, 262–268.

Evaluating State Modeling Techniques in Alloy

ALLISON SULLIVAN, KAIYUAN WANG, and SARFRAZ KHURSHID, The University of Texas at Austin, USA
DARKO MARINOV, University of Illinois at Urbana-Champaign, USA

Software models help develop higher quality systems. The declarative language Alloy and its accompanying automatic analyzer embody a method for developing software models. Our focus in this paper is Alloy models of systems where different operations may mutate the system state, e.g., addition of an element to a sorted container. Researchers have previously used two techniques for modeling state and state mutation in Alloy, but these techniques have not been compared to each other. We propose a third technique and evaluate all three techniques that embody conceptually different modeling approaches. We use four core subjects, which we model using each technique. Our primary goal is to quantitatively evaluate the techniques by considering the runtime for solving the ensuing SAT formulas. We also discuss practical tradeoffs among the techniques.

Categories and Subject Descriptors: I.6.4 [Simulation and Modeling] Model Validation and Analysis; D.2.5 [Software Engineering] Testing and Debugging; D.2.4 [Software Engineering] Software/Program Verification

Additional Key Words and Phrases: Alloy, state modeling, SAT solving, empirical evaluation, predicate parameterization

1. INTRODUCTION

Building and analyzing software models plays an important role in developing higher quality software systems. The Alloy tool-set – including the declarative language Alloy and its accompanying automatic analysis engine called Alloy analyzer – embodies a method for developing software models [Jackson 2006]. The Alloy language is a relational, first-order logic with transitive closure that allows succinct formulation of complex structural properties. The Alloy analyzer performs scope-bounded analysis of Alloy formulas using off-the-shelf Boolean satisfiability (SAT) solvers. The analyzer can generate two forms of valuations for the relations in the model: (1) instances, i.e., valuations such that the formulas hold; and (2) counterexamples, i.e., valuations such that the negation of the formulas holds. The analyzer enables Alloy users to not only validate their models but also use the tool-set as a basis for various forms of software analyses.

Our focus in this paper is Alloy models of systems where different operations may mutate the system state, e.g., addition of an element to a sorted container. Researchers have used at least two techniques for modeling state and state mutation in Alloy [Jackson and Vaziri 2000; Jackson and Fekete 2001; Marinov and Khurshid 2001; Taghdiri 2003; Frias et al. 2005]. One common technique, which we call additional state type, is to introduce a set of state atoms and increase the arity of each relation to add a new state type [Jackson and Fekete 2001; Taghdiri 2003; Frias et al. 2005]. Another common technique, which we call relation duplication, is to duplicate relations in the model such that one set of relations

This research was partially supported by the NSF Grant Nos. CNS-1239498, CCF-1319688, CCF-1409423, and CCF-1421503. Authors’ addresses: Allison Sullivan, Kaiyuan Wang, and Sarfraz Khurshid, Department of Electrical and Computer Engineering, The University of Texas at Austin, Austin, TX 78712, USA; Darko Marinov, Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, IL 61801, USA. Email correspondence: khurshid@utexas.edu

Copyright © by the paper’s authors. Copying permitted only for private and academic purposes.
represents one state, say pre-state, and another set identifies another state, say post-state [Jackson and Vaziri 2000; Marinov and Khurshid 2001]. A shared intuition at the basis of these techniques is to (explicitly or implicitly) create a representation of each desired state in the model, and write formulas that constrain specific states individually, or sets of states collectively, e.g., to encode post-conditions that relate pre- and post-states. Despite the common basis, these techniques are technically quite different – not only in terms of syntactic and semantic representation but also in terms of the state spaces that ensue for SAT exploration.

While state modeling techniques have allowed effective applications of Alloy in various domains – including software design [Jackson and Fekete 2001; Taghdiri 2003; Frias et al. 2005], analysis [Jackson and Vaziri 2000; Dennis et al. 2006; Milincevic et al. 2011; Galeotti et al. 2013], testing [Marinov and Khurshid 2001], and security [Kang et al. 2016] – these techniques have not been compared to each other. We propose a third technique, called predicate parameterization, and compare all three techniques that embody conceptually different modeling approaches. We use four core subjects that we chose because they represent two broad classes of problems – two subjects are data structures representative of many evaluations done with Alloy [Jackson and Vaziri 2000; Marinov and Khurshid 2001; Galeotti et al. 2013] and two subjects are from the standard Alloy distribution. We are not aware of any common benchmark set of Alloy models for evaluating performance of the Alloy analyzer. We model each subject using each technique. We do not use more or bigger subjects because translating each model from one technique to another currently requires a substantial manual effort. Our primary goal is to quantitatively evaluate the techniques by considering the runtime for solving the ensuing SAT formulas. (In other words, we do not consider the asymptotic algorithm complexity but the actual practical performance.) We also discuss practical trade-offs among the techniques.

2. TECHNIQUES

This section describes the three state modeling techniques that we evaluate. We first introduce an illustrative example and some basic concepts of Alloy (Section 2.1). We then describe the three techniques and illustrate them using our example (Section 2.2).

2.1 Illustrative example and Alloy basics

Consider modeling an acyclic, sorted, singly-linked list with unique elements in Alloy. The following snippet declares the basic Alloy data-types:

```
sig List {
   header: lone Node
}
sig Node {
   elem: Int,
   link: lone Node
}
```

The `sig` declaration introduces a set of atoms and optionally declare fields, i.e., relations. The field `header` is a binary relation of type `List × Node` and represents the list's first node; `elem` has type `Node × Int` and represents the node's integer (Int) element; and `link` has type `Node × Node` and represents the node's next node. The keyword `lone` declares the binary relation to be a partial function, e.g., each list has at most one header node, and each node has at most one next node. By default, each binary relation that is declared is a total function, e.g., each node contains exactly one integer element.

Consider expressing acyclicity. The following snippet is an Alloy predicate (`pred`), i.e., a named, parameterized formula that may be invoked elsewhere, which defines acyclicity using universal quantification (`all`):
Evaluating State Modeling Techniques in Alloy

•

16:3

pred Acyclic(l: List) {
    all n: l.header.*link | n !in n.^link
}

The operator ‘.’ is relational composition; ‘*’ is reflexive transitive closure; and ‘~’ is transitive closure. Note that ‘*’ and ‘~’ are used as prefix not suffix operators. The (infix) operator ‘!in’ denotes that the left-hand expression is not a subset of the right-hand expression. Note that this operator does not denote just “not an element” because all Alloy expressions are semantically relations (even if of arity only one, i.e., sets) and not scalar atoms [Jackson 2006]. The expression l.header.*link represents the set of all nodes reachable from l’s header along link (including the header itself). The predicate encodes that for any node n in the list, the set of nodes reachable from n does not contain n, hence no cycle.

The following predicate defines sortedness (with unique elements):

pred SortedUnique(l: List) {
    all n: l.header.*link | some n.link => n.elem < n.link.elem
}

The operator ‘=>’ is logical implication. The formula some n.link encodes that the expression n.link is a non-empty set. The predicate encodes that for any node in the list, if the node has a next node, the elements from the two nodes are in the ascending order; the operator ‘<’ is integer comparison.

The following Alloy snippet defines the predicate RepOk that is a conjunction of Acyclic and SortedUnique, and uses the run command to instruct the analyzer to create an instance in the scope of 1 list, 3 nodes, and bit-width of 2 for integers:

pred RepOk(l: List) {
    Acyclic[l]
    SortedUnique[l]
}
run RepOk for 1 List, 3 Node, 2 int

2.2 Additional state type

The most widely used technique for modeling state in Alloy is to introduce a new sig, commonly called State, and add it to each relation, increasing the relation’s arity by one [Jackson and Fekete 2001; Taghdiri 2003; Frias et al. 2005]. For example, the following snippet shows this technique applied to the list declaration:

abstract sig State {}
sig List {
    header: Node -> State
}

fact {
    all l: List | all s: State | lone l.(header.s)
}

State is an abstract sig, i.e., it contains only atoms that are strictly necessary for the constraint solved. The symbol ‘->’ denotes the Cartesian product in expressions and adds arity in declarations. The field header is now a ternary relation of type List × Node × State, which allows a list to have different nodes as its header in different states. Note that the state need not be the last type; it can be in any position, e.g., in the first position where the sig State would have other relations (such as header and link) as its fields. We use state in the last position because it allows us to preserve the declaration structure of the original model. A fact in Alloy is a formula that must always hold. We use a fact to require each list to have at most one header node in each state to conform to the partial function relation in the original model (without state).
The following snippet shows how the predicate `Acyclic` can be written in the presence of state:

```java
pred Acyclic(l: List, s: State) {
    all n: l.(header.s).*!(link.s) | n !in n.^!(link.s)
}
```

Note the new state parameter `s` for which the predicate holds, and also the new composition of each relation with a state to represent the field values in the desired state, e.g., `l.(header.s)` is the header of the list `l` in the state `s`.

Consider next modeling state mutation. This snippet defines removal of the first node from the list:

```java
pred RemoveFirst(l: List, s: State, s': State) {
    s != s' -- states are unique
    RepOk[l, s] -- l satisfies RepOk in s
    l.(header.s).*!(link.s).(elem.s) = l.(header.s).(elem.s) = l.(header.s').*(link.s').(elem.s')
    RepOk[l, s'] -- l satisfies RepOk in s'
}
```

```java
run RemoveFirst for 2 State, 1 List, 3 Node, 2 Int
```

The predicate has two state parameters: `s` represents the pre-state, and `s'` represents the post-state. The operator `-' is set difference. (The symbol `-' is used for comments.) The predicate encodes that the two states are distinct; `l` satisfies `RepOk` in the pre-state; the set of elements in the pre-state minus the header element in the pre-state is the set of elements in the post-state; and `l` satisfies `RepOk` in the post-state. Figure 1 graphically illustrates an instance for `RemoveFirst`.

![Diagram](image)

Fig. 1. Example RemoveFirst (a) pre-state and (b) post-state visualized using Alloy analyzer.

Consider next using the analyzer to check whether `RemoveFirst` has a specific property. The following snippet uses an Alloy assertion (assert) to encode that `RemoveFirst` implies that the header element in the post-state is the second element from the pre-state:

```java
assert PartialCorrectnessOnce {
    all disj s, s': State | all l: List |
    RemoveFirst[l, s, s'] => l.(header.s').(elem.s') = l.(header.s).(link.s).(elem.s)
}
```

```java
check PartialCorrectnessOnce for 3
```
The keyword `disj` requires \( s \) and \( s' \) to be distinct. The command `check` instructs the analyzer to find a counterexample to the named assertion, i.e., `PartialCorrectnessOnce`. However, the analyzer does not find a counterexample for this command in this example for the given scope of 3. (There could exist a counterexample in a larger scope.)

### 2.3 Relation duplication

Another technique for modeling state is to introduce a new copy of declared relations for each state and to model mutation by defining constraints across the relations for different states [Jackson and Vaziri 2000; Marinov and Khurshid 2001]. To illustrate, consider modeling pre-state and post-state for `RemoveFirst`. (In general, there could be more than two states, and the relations would need to be copied multiple times.) The following snippet shows this technique applied to the list declaration:

```alloy
sig List {
    header: lone Node, -- pre-state
    header': lone Node -- post-state
}
```

The mutation of the original header field is now modeled by two relations: `header` that represents the value in the pre-state, and `header'` that represents the value in the post-state.

Constraints on the relations are now written over appropriate groups of relations. The following snippet shows how two predicates can be written to represent acyclicity for the two states:

```alloy
pred Acyclic(l: List) { -- for pre-state
    all n: l.header.*link | n !in n.^link
}
pred Acyclic'(l: List) { -- for post-state
    all n: l.header'.*link' | n !in n.^link'
}
```

`Acyclic` represents acyclicity in the pre-state, and `Acyclic'` represents acyclicity in the post-state. Note that each predicate uses relations only from its corresponding state. Similar changes are made for `RepOk` and `RepOk'` (and `SortedUnique` and `SortedUnique'`).

Consider next modeling state mutation. This snippet defines `RemoveFirst` using this technique:

```alloy
pred RemoveFirst(l: List) {
    RepOk[l] -- RepOk in pre-state
    l.header.*link.elem - l.header.elem = l.header'.*link'.elem'
    RepOk'[l] -- RepOk in post-state
}
run RemoveFirst for 1 List, 3 Node, 2 Int
```

Moreover, the following snippet defines the assertion `PartialCorrectnessOnce` using this technique:

```alloy
assert PartialCorrectnessOnce {
    all l: List | RemoveFirst[l] => l.header'.elem' = l.header.link.elem
}
check PartialCorrectnessOnce for 3
```

### 2.4 Parameterization

The third technique we evaluate removes all relation declarations from sig declarations, adds the relations as parameters to all predicates, and adds a new predicate to express all the facts in the model. For example, the declaration of the list signature becomes just the following:
The following snippet illustrates adding the relations as parameters to the acyclicity predicate:

```alloy
def Acyclic(l: List, header: List -> Node, elem: Node -> Int, link: Node -> Node) {
  all n: l.header.*link | n !in n.^link
}
```

Similar changes are made for RepOk (and SortedUnique).

In addition to changing all the existing predicates, a new predicate is added to encode all the facts from the original model. The following snippet illustrates the new predicate, which should be appropriately invoked when commands are executed:

```alloy
def SigDeclFacts(header: List -> Node, elem: Node -> Int, link: Node -> Node) {
  all l: List | lone l.header
  all n: Node | one n.elem and lone n.link
}
```

In Alloy, 'one' holds if its expression denotes a singleton set/relation, while 'and' is the usual conjunction, expressed explicitly. (There is also an implicit conjunction among the formulas on different lines.)

Consider next modeling state mutation. This snippet defines RemoveFirst using this technique:

```alloy
  RepOk[l, header, elem, link]
  l.header.*link.elem - l.header.elem = l.header'.*link'.elem'
  RepOk[l, header', elem', link']
}
  SigDeclFacts[header, elem, link] and SigDeclFacts[header', elem', link']
  RemoveFirst[l, header, elem, link, header', elem', link']
}
run RunRemoveFirst for 1 List, 3 Node, 2 Int
```

In addition to the list parameter, RemoveFirst has 6 relations as parameters: 3 for pre-state (header, elem, and link) and 3 for post-state (header', elem', and link'). To run RemoveFirst, a new predicate RunRemoveFirst is introduced and run, which appropriately enforces the facts from the original model (without state). This new predicate RunRemoveFirst is not expected to be invoked elsewhere (in another predicate); its only purpose is to enable a run command that conforms to the semantics of facts in Alloy.

The following snippet defines the assertion PartialCorrectnessOnce using this technique:

```alloy
assert PartialCorrectnessOnce {
  all l: List | all header: List -> Node | all elem: Node -> Int | all link: Node -> Node |
  all header': List -> Node | all elem': Node -> Int | all link': Node -> Node {
    SigDeclFacts[header, elem, link] and SigDeclFacts[header', elem', link']
    RemoveFirst[l, header, elem, link, header', elem', link'] => l.header'.elem' = l.header.link.elem
  }
}
```

The assertion assumes the facts, once again to conform to the semantics of facts in Alloy.

3. EVALUATION

We use four core subjects – two data structures and two subjects from the standard Alloy distribution – as base models, providing us 11 constraint-solving problems with different complexities to quantitatively compare the three techniques:
(1) **Singly-linked list**, our running example; we derive four problems: (a) create an instance for removing the first element (`RemoveFirst`), our running example; (b) create an instance for removing the first element twice (`TwiceRemoveFirst`), which requires three states (unlike our running example that used only two states); (c) check that `RemoveFirst` implies that the header element in post-state is the second element in the pre-state (`PartialCorrectnessOnce`); and (d) check that if a list has two or more elements, removing the first element twice implies the number of nodes in the list reduces by two (`PartialCorrectnessTwice`);

(2) **Binary search tree**, we derive four problems: (a) create an instance for adding a given element to the tree (`Add`); (b) create an instance for removing a given element from the tree (`Remove`); (c) check that adding an element that is not in the tree followed by removing the same element leaves the set of elements originally in the tree unchanged (`AddRemoveNoOp`); and (d) check that two is the difference in the number of nodes between (i) adding a new element to the tree versus (ii) removing an existing element from the tree (`AddRemoveComparison`).

(3) **Farmer**, the classic puzzle on crossing the river, which describes that a farmer wants to move a fox, a chicken, and a bag of grain from one bank of a river to the other bank without losing any of them. This model comes with the standard Alloy distribution, where it already has a `State` signature that represents the object status for both river banks every time the farmer moves. The state is the first type in the corresponding relations. The model includes two problems: (a) solve the puzzle (`solvePuzzle`); and (b) check that no object is at more than one place at the same time (`NoQuantumObjects`).

(4) **Dijkstra**, a model of Dijkstra's mutual exclusion for processes, which is also in the standard Alloy distribution; similar to Farmer, state is the first type in the relations used to model mutation. The model includes three problems: (a) create an instance that shows a deadlock (`Deadlock`); (b) try to find a deadlock instance where the process mutexes are grabbed and released based on the Dijkstra algorithm (`ShowDijkstra`); and (c) directly check that the Dijkstra algorithm prevents deadlocks (`DijkstraPreventsDeadlocks`).

Table I. State techniques comparison; P.V. and Cl. are primary variables and clauses, resp., in the SAT formula.

<table>
<thead>
<tr>
<th>Model</th>
<th>Problem/Command</th>
<th>Additional state type</th>
<th>Relation duplication</th>
<th>Parameterization</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Ti[m]</td>
<td>P.V.</td>
<td>Cl.</td>
<td>Ti[m]</td>
</tr>
<tr>
<td>List</td>
<td>RemoveFirst</td>
<td>102</td>
<td>59</td>
<td>2706</td>
</tr>
<tr>
<td></td>
<td>TwiceRemoveFirst</td>
<td>83</td>
<td>89</td>
<td>4514</td>
</tr>
<tr>
<td></td>
<td>PartialCorrectnessOnce</td>
<td>13949</td>
<td>216</td>
<td>13709</td>
</tr>
<tr>
<td></td>
<td>PartialCorrectnessTwice</td>
<td>44555</td>
<td>219</td>
<td>18391</td>
</tr>
<tr>
<td>Tree</td>
<td>Add</td>
<td>27</td>
<td>81</td>
<td>3162</td>
</tr>
<tr>
<td></td>
<td>Remove</td>
<td>8</td>
<td>81</td>
<td>3162</td>
</tr>
<tr>
<td></td>
<td>AddRemoveNoOp</td>
<td>31</td>
<td>262</td>
<td>13583</td>
</tr>
<tr>
<td></td>
<td>AddRemoveComparison</td>
<td>230366</td>
<td>278</td>
<td>13698</td>
</tr>
<tr>
<td>Farmer</td>
<td>solvePuzzle</td>
<td>29</td>
<td>66</td>
<td>2239</td>
</tr>
<tr>
<td></td>
<td>NoQuantumObjects</td>
<td>45</td>
<td>78</td>
<td>2441</td>
</tr>
<tr>
<td>Dijkstra</td>
<td>Deadlock</td>
<td>134</td>
<td>260</td>
<td>826</td>
</tr>
<tr>
<td></td>
<td>ShowDijkstra</td>
<td>87</td>
<td>47</td>
<td>1520</td>
</tr>
<tr>
<td></td>
<td>DijkstraPreventsDeadlocks</td>
<td>546</td>
<td>210</td>
<td>9498</td>
</tr>
</tbody>
</table>

Table I shows the experimental results. For each model, we list the executed commands. The scope for **List** (resp., **Tree**) has values as shown in the example: 1 list (resp., 1 tree), 3 nodes, and bit-width of 2 for integers. The scope for **Farmer** has 8 states and 4 fixed objects (Farmer + Fox + Chicken...
Allison Sullivan et al

+ Grain). The scope for Dijkstra has 5 State, 5 Process, 5 Mutex for Deadlock; 5 State, 2 Process, 2 Mutex for ShowDijkstra, and 5 State, 5 Process, 4 Mutex for DijkstraPreventsDeadlocks. We leave it as future work to experiment with different scopes and models.

For each modeling technique, we tabulate time (in milliseconds) to solve the resulting SAT formula ($T_{ms}$), the number of primary variables (P.V.), and the number of clauses (Cl.) in the SAT formula. All the experiments were run on an Intel Celeron CPU N3060 1.60GHz x 2 processor with 1.8GB of memory using Alloy 4 (http://alloy.mit.edu/alloy/downloads/alloy4.jar). We initially tried to use Alloy 4.2, the latest Alloy release, but encountered an anomalous behavior: our list and tree models using parameterization created SAT formulas with 0 primary variables and 0 clauses; we confirmed that this is a bug in Alloy 4.2.

For the four problems where the solving time exceeds 500ms for any of the techniques, parameterization provides the most efficient solving, followed by relation duplication, and then additional state type. The performance difference is the greatest for PartialCorrectnessTwice in list, where parameterization provides a speedup of 8X over additional state type. While the time for SAT solving is determined by the complexity, not just the size, of the SAT formula, one reason for the performance difference can be the size. For each of these four problems, the number of primary variables is the smallest for parameterization, which is the same as the number for duplication with one exception (PartialCorrectnessOnce). Moreover, the number of clauses for parameterization and duplication is quite close for these four problems but noticeably smaller than the number for additional state type. Overall, parameterization enables a tight encoding that leads to efficient analysis for these problems.

For the problems where the solving time is below 500ms for all techniques, the difference in time among the techniques is not practically relevant, so any can be used for just one small problem. However, the techniques do differ, and more precise and extensive measurements would be needed to find the best technique for analyzing a large number of small problems. In particular, it would be important to understand the cases where parameterization is not the best technique.

Quantitatively, the models created using parameterization are the fastest to solve. Qualitatively, however, the models created using additional state type are most readable due to two reasons: (1) state can be conveniently referred to, e.g., a quantified formula can directly be written over the set of states; and (2) the type declaration structure of the original model (without state) can be largely preserved. The models with relation duplication are burdensome for (manual) maintenance because the predicates have to exist in multiple copies (e.g., Acyclic and Acyclic'). The models with parameterization require unwieldy predicate signatures because all predicates are parameterized; in addition, facts need to be explicitly handled in a special way. We note that different techniques are best suited to different purposes, e.g., additional state type for manual modeling, and relation duplication and parameterization for automated analyses where the models are mechanically generated. Indeed, future work should consider automatic translations that map a model built using one technique to conform to another technique for more efficient back-end analysis.

4. CONCLUSIONS

The Alloy software modeling tool-set has been effectively used in software design, analysis, and testing. Our focus in this paper was on comparing Alloy modeling techniques for systems where different operations may mutate the system state. Over the years, researchers have use at least two techniques for modeling state and state mutation in Alloy, but these techniques were not previously compared to each other. We proposed a third technique and evaluated all three techniques that embody different modeling approaches. We used four core subjects, which we model using each technique. The results show that the models created using the parameterization technique are the fastest to solve. However, such models are hard to write manually and should be automatically derived from different models.
REFERENCES
Measuring Mangled Name Ambiguity in Large C/C++ Projects

RICHÁRD SZALAY, Eötvös Loránd University, Faculty of Informatics, Dept. of Programming Languages and Compilers
ZOLTÁN PORKOLÁB, Eötvös Loránd University, Faculty of Informatics, Dept. of Programming Languages and Compilers, and Ericsson Ltd.
DANIEL KRUPP, Ericsson Ltd.

Symbol reference has an important role in various areas of software technology. Some modern object-oriented languages use it for implementing function and operator overloading, linkers use it for connecting objects between different translation units, and development environments need to accurately understand them to provide features like traversing all references. In case of the C++ programming language, the most current tools use mangled names to correlate symbols, e.g. when implementing actions like “go to definition” or “list all references”. However, for large projects, where multiple binaries are created, symbol resolution based on mangled names can be, and usually is, ambiguous. This leads to inaccurate behaviour even in major development tools. Does this problem inherently stem from the size of the codebase, or is it just another indication of possibly underachieving software quality? In this paper, we discuss the prevalence of this problem on five projects, from open source to industrial code bases with varying degree of code size, to give an input on whether or not the research of a better symbol resolution algorithm is necessary.

1. INTRODUCTION

The size and scale of software systems have grown rapidly over the course of the recent years. Large-scale projects amounting up to million lines of code aren’t uncommon – e.g. the source code of the Linux kernel, with drivers, is around 17 million lines of code. This presents a challenge when it comes to understanding and navigability of the project. It is always essential to understand the precise behaviour of a software system when we are fixing a bug, or extending the system with a new functionality, and the importance of this understanding is preeminent when a major refactoring task is being undertaken.

To enhance understanding of the project, we have to correlate the occurrences of symbols in the source code and how these symbols are present in the built binaries. This helps developers to understand the project they are working on better via code comprehension tools while helping automated defect detection via static analysis tools. Consider, though, that this correlation usually can’t be dis-
covered using simple name identity: in most programming languages, the same name could mean different symbols based on the occurrence's context, e.g. attributes or methods with the same name in different classes, overloaded version of functions, or functions with the same name in different namespaces/packages or build configurations.

Modern software tools, like various development frameworks [Microsoft 2016a; CLion 2016; NetBeans 2016; Eclipse 2016] and code comprehension tools [Woboq 2016; OpenGrok 2016] provide discovery functionality: the user of the tool can jump to the place of the definition or can iterate all places of reference of a given symbol. Naturally, we expect such tools to work not on simple name identity, but on exact symbol correlation. Unfortunately, we found that current development and comprehension tools fall behind and into ambiguity with the increase of the code base’s size. The reason is – as we will explain – that mangled names are allowed to be ambiguous when multiple executables are produced to form the released product.

This paper is organised as follows:
We describe the ambiguity problem in Section 2 in details. Current tools’ behaviour is evaluated in Section 3. Section 4 presents our measurement results on five projects. The related work with possible future directions are discussed in Section 5. Our paper concludes in Section 6.

2. PROBLEM DESCRIPTION

We used the following criteria on “When is a symbol unambiguously defined?” for the measurements discussed in this paper. We have devised to be sufficient in our C++ context of large projects. In the attached tables, we show our measurements on how many of such problematic symbols were found for our test projects.

A function symbol \( f() \) (mangled name in LLVM/Clang: _Z1fv) is considered ambiguous if neither of the following conditions hold true for all symbols having this particular mangled name in the entire project:

—The function has exactly one definition node
—The function does not have a definition node, but has exactly one declaration node
—The function has neither a definition node nor a declaration node

A type symbol \( T \) (mangled name in LLVM/Clang: T) is considered ambiguous if it is defined more than once in the entire project. Types can be declared as many times as needed. A header discovered to be included multiple times in the same translation unit does not constitute as multiple definitions, as header guards make sure the preprocessed text is well-formed.

In the case of the C++ programming language, usually the mangled name [Stroustrup 2013; Ellis and Stroustrup 1990] is used to distinguish between different symbols of the same name. The mangled name is constructed using (possibly multiple) namespace and class information by concatenation, but its exact form is compiler-dependent. Compilers use mangled names to enable operator overloading [Stroustrup 1994], i.e. they generate different mangled names for functions with the same name but different parameter lists. Linkers use these mangled names to resolve symbols [ISO C++ 2014a].

In the case of the C language, there is no such thing as namespace, class, or function overloading [ISO C 2011]. Still, in certain cases, such as specific optimisations on how a method is called, name decoration occurs [Microsoft 2016b]. In this paper we will refer to the name visible to the linker as mangled name, both in the context of C and C++ for the sake of simplicity.

While mangled names must be unique for all translation units linked into a single executable, this does not stand for large-scale projects where multiple executables are typical.

In a software comprehension activity the most frequent questions users ask are “Where is this method called, or type referenced?” and “What does the declaration or definition of this looks like?” [Sil-
Fig. 1. Jumping to the definition of $f()$ from the calls is ambiguous in this example project layout.

lito et al. 2008]. A software comprehension tool should be able to answer these questions as precisely as possible, as accuracy ensures more optimised usage of the developers’ time spent working. Both questions lead to the fundamental problem of correctly resolving references to the definition and usages of a type, a function, or variable, and other language components.

According to the One Definition Rule (ODR) of C++ [ISO C++ 2014b], only one definition of any variable, function, class type, enumeration type, or template is allowed in any translation unit. When resolving references to ordinary C functions, static and non-virtual C++ member functions, type names or non-polymorphic variables, the unique definition within a single translation unit can be found based on static information. Specifically, the function definition of non-virtual functions or ordinary C functions can be looked up based on function signature, which – according to [ISO C++ 2014c] – contains the name of the function, the enclosing namespace, class of which the function is member of (in the context of C++), the type of the parameters, template parameter list (in case of function templates in C++), cv- and ref-qualifiers, unique type names (qualified with namespace) and scope-correct variable names.

There can be, however, more than one definition belonging to the same unique-name or signature, but defined in different translation units that are not linked together. This is a typical scenario in large-scale programs consisting of multiple separate executables and build configurations, e.g. every executable having a $\text{main}()$ function as entry point.

Since the translation unit containing the reference and the set of translation units linked together is known for the linker, it is possible for the linker to look up the correct, unique definition for any given reference.

In contrast, for a software development or comprehension tool, while the user is browsing a source file, the linkage context (the set of translation units linked together) where the definitions should be resolved is usually unknown [Ferenc et al. 2002]. This leads to ambiguous type, function or variable references. In some cases this ambiguity can be resolved automatically, by taking into consideration the linkage information. For example in Figure 1, the reference $f()$ in entry.cpp can be resolved to the definition in other.cpp, since it is known that entry.cpp is only linked with other.cpp in any target binary.

In some cases the ambiguity can only be resolved by asking the user to specify the linking context, but asking the linking context should only be done when it is absolutely necessary, to avoid superfluous user interaction that causes disorientation in the navigation process [Alwis and Murphy 2006], as
developers have to resolve symbol references using their internal and external knowledge instead of a proper tool solving the problem for them.

3. STATE OF THE ART

Current well-known development, engineering, and code comprehension tools rarely provide good support for this problem. We have analysed some common tools in terms of their reaction to a “jump to definition” query:

**Microsoft Visual Studio** [Microsoft 2016a] shows a disambiguation page when encountering the problem described in this paper. If the entire solution (a group of projects handled together) is configured for a certain dependency (i.e. only one of the ambiguous definitions is compiled when the solution is built) and the user changes the internal settings of the solution, Visual Studio decides which symbol a “jump to definition” query jumps to. This fine-tuning on the users’ end seemingly does not affect “get calls”/“get usage”-like queries, which still show results with every possible option present, including those which are clearly not valid in the solution’s current state.

**JetBrains CLion** [CLion 2016] analyses and builds symbol information when a project is configured, and one project having multiple separately configured executable targets misleads the IDE. A certain file is designated as location where function \( f() \) is defined – our understanding currently reveals that the file which has a larger name in lexicographical order. The ambiguity is present even when a certain executable is being debugged by the IDE with “Step Into” showing the proper implementation being executed while “Jump to Definition” opening an entirely distinct source file.

**NetBeans** [NetBeans 2016] does not show a disambiguation page at all, seemingly jumping to a file first detected for a certain symbol after the last build – this can be overridden by manually setting certain files to “Exclude from Build” after which the file’s symbols won’t contribute to the set of potential results. The file to which NetBeans jumps for a definition varies between client restarts, seemingly in a non-deterministic fashion; the only exception is when the symbol is defined in the same file where it is used: in this case all queries jump to this location in particular.

**Eclipse** [Eclipse 2016] properly prioritises symbols explicitly defined in the same source file, but if the definition is not found in the file where a query is issued (see example in Figure 1), a disambiguation page is shown. Putting different builds into entirely different projects with their own Makefile solves this issue, but search queries do not traverse project boundaries.

**Woboq Code Browser** [Woboq 2016] shows the locations where a symbol is defined when viewing information about a particular symbol, but the problem of Section 2 is present. Jumping to definition by clicking on a usage location jumps the user to the definition that has been first, in the order of build commands discovered by the codebrowser_generator tool of Woboq. It binds usages and definitions in the same source file together.

**OpenGrok** [OpenGrok 2016] uses Apache Lucene to index source code and it considers source files as pure textual input with no knowledge of build relationships. Thus, it always has to provide a disambiguation page if a mangled name is ambiguously defined.

Software development and comprehension tools usually, at most, show a disambiguation page and put weight on the developer’s shoulders to make out which is the “real” definition said call is referring to.

4. MEASUREMENTS AND RESULTS

We used LLVM/Clang to extract information from the Abstract Syntax Tree (AST) generated by parsing our analysed projects. This information has been saved into a database, on which we ran our investigations.
We have measured the problem on different projects, such as Apache Xerces [Apache 2016], CodeCompass [CodeCompass 2016], LLVM/Clang infrastructure [Clang 2016], LLVM/Linux [LLVM/Linux 2016] (the GNU/Linux operating system ported to be parseable and compilable on LLVM/Clang) and an Ericsson platform product, TSP [Ferraro-Esparza et al. 2002].

Attached tables provide an overview on the quantities. To outline the occurrence of the problem, we must investigate the number of distinct symbol names, i.e. how many mangled-name equivalence groups can be found in the project. The number of nodes show how many times a particular symbol (function or type) appears in the source code. Each node has a name, the symbol it references. The total number of definitions, declarations and usages (function calls, the name of a type appearing in a function signature or a variable declaration) is also presented.

In theory, the number of definitions should equal the number of unique names (assuming that the project is a single binary, because, as ODR states each function, global variable, etc. must have exactly one definition), but this is clearly not the case: first of all, software projects aren’t comprised of a single one, but this is clearly not the case: first of all, software projects aren’t comprised of a single binary, and some symbols are resolved dynamically.

To understand the impact of the problem on developer effort and navigation costs, we have to visualise the individual groups and see the ambiguity in detail. For example, in Xerces, 19 unique function names are considered ambiguous, and these names are used in 348 locations. The distribution of this ambiguity is an important factor of impact, as each non-unique name having 18 nodes attached (348/19 = 18, so the ambiguity is averaged out) is not as powerful, as 1 important name (one that is frequently navigated by developers, for call sites and definitions) having ~ 300 usages would be. Figure 2. shows the distribution of the number of definition symbols for a mangled name group.

We have seen that while the problem appears to affect a considerable ratio of symbols (0.5 — 10%) if the project is viewed as a whole, the individual cases of ambiguity is resolvable by the developers easily. On the other hand, automated toolchains, such as static analysers, can fall into indecision when meeting this ambiguity, as better heuristics could help symbol resolution.

In the following subsections we will discuss in detail the findings in every project we have evaluated individually.

4.1 Xerces

Apache Xerces [Apache 2016] is an open-source XML parser library framework. The most prominent ambiguity is from the \texttt{StrX} class and its two member functions, which are defined in the \texttt{tests} and \texttt{samples} folder. They account for 55% (4 names having 15 definition each, and 182 call sites in total) of the duplication. The usage() function, existing only in these sample projects, accounts for the remainder of the duplication in Xerces, totalling 16 definitions with 37 call sites, which are individually located in their respective sample projects.
Table II. Distribution of ambiguity and symbol types amongst types

<table>
<thead>
<tr>
<th>Types</th>
<th>Names</th>
<th>Nodes</th>
<th>Decl.</th>
<th>Def.</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Xerces [Apache 2016]</td>
<td>2,849</td>
<td>60715</td>
<td>2,248</td>
<td>1,492</td>
<td>57,065</td>
</tr>
<tr>
<td></td>
<td>(9, 0.3%)</td>
<td>(329, 0.5%)</td>
<td>(4%)</td>
<td>(2%)</td>
<td>(94%)</td>
</tr>
<tr>
<td>CodeCompass [CodeCompass 2016]</td>
<td>346,387</td>
<td>779,523</td>
<td>182,413</td>
<td>169,124</td>
<td>427,986</td>
</tr>
<tr>
<td></td>
<td>(1,315, 0.4%)</td>
<td>(9,461, 1.2%)</td>
<td>(23%)</td>
<td>(22%)</td>
<td>(55%)</td>
</tr>
<tr>
<td>LLVM [Clang 2016]</td>
<td>1,628,232</td>
<td>4,206,743</td>
<td>860,245</td>
<td>789,886</td>
<td>2,556,612</td>
</tr>
<tr>
<td></td>
<td>(7,192, 0.5%)</td>
<td>(73,375, 1.8%)</td>
<td>(20%)</td>
<td>(19%)</td>
<td>(61%)</td>
</tr>
<tr>
<td>Linux [LLVM/Linux 2016]</td>
<td>19,883</td>
<td>159,581</td>
<td>1,990</td>
<td>20,029</td>
<td>137,562</td>
</tr>
<tr>
<td></td>
<td>(188, 1%)</td>
<td>(579, 0.3%)</td>
<td>(1%)</td>
<td>(13%)</td>
<td>(86%)</td>
</tr>
<tr>
<td></td>
<td>(2,478, 1.2%)</td>
<td>(74,511, 3.5%)</td>
<td>(3%)</td>
<td>(8%)</td>
<td>(89%)</td>
</tr>
</tbody>
</table>

Thus, in Xerces, a developer working on the “actual” project is not likely to be led off-road by the ambiguity, and in the test/sample projects, the only decision the developers need to take in case of a disambiguation page, is to select the referenced symbol within the same test/sample. However, this deeper inspection of the code revealed a smell: the implementation of StrX is duplicated in all the test projects – which could make changes to the test/sample infrastructure a hard task.

4.2 LLVM/Clang

In case of LLVM [Clang 2016], certain functions, such as fieldFromInstruction and getMnenomic, etc., related to different targeted architectures form the main mass of duplication. In case of the first function, we measured 6244 call sites to 7 definitions. Only 650 of the calls were not in the same source file as a definition, however, in this case, a disambiguation is easily resolvable by using the source file which corresponds to the compilation architecture the developer is currently working on or navigating.

A problem can, however, arise for static analysis tools or any automated tool which would expect mangled names to be unique on the project level, as in this case, the tool needs to be armed with a heuristic that can figure out which implementation to use – due to LLVM’s high-grade code quality, a simple “similar looking source code path” rule could be enough.

4.3 LLVM/Linux

The LLVM/Clang parser compatible version of the GNU/Linux kernel, called LLVM/Linux [LLVM/Linux 2016] (the original source code cannot be parsed by Clang due to usage of non-standard, GNU GCC specific extensions) contains type duplication which are almost all only a 2 definition kind. We need to point out that it is C, and not C++ code, and Linux is an operating system’s kernel, so the language rules and the developer intentions are different in this case. However, these duplicate definitions are entirely separate in their use and Application Binary Interface (ABI). Examples include module and file. Based on the name, it is not easily seen which instance should be used in a given context, as some member fields overlap.

Almost all ambiguous function definitions fall into the 2 definition category too. These result from the fact of straight code duplication, to which most headers and source files state Copied from <PATH> in their preamble. Either in case of fully copied files or copied bits of code, we did not find different implementation – in terms of source code text, and not program semantics! However, a modification to

1LLVM is an open source project backed by software companies worldwide providing a unified way of generating machine code from the LLVM Internal Representation (IR) language. Clang is a frontend suite, which translates C/C++ source code to LLVM IR. The two projects together make up for a versatile C/C++ compiler.
either of the ambiguous definitions most likely will require a change to the other, which could bring forth undocumented or unforeseen consequences to the kernel as a whole.

The Linux kernel has been debated for its code quality since it started gaining popularity. The above examples show, that a "happens-to-work" mentality stands as an obstacle for current and future developers, which can hinder the evolution of the project. Easy understanding, security and safety in an operating system should be a must, and we can just again talk about static analysis tools, which are majorly suffering due to ambiguity.

4.4 TSP

Ericsson’s Telecom Server Platform [Ferraro-Esparza et al. 2002] combines multiple well-known open source and industrial, proprietary technologies. Because of this, most of the Linux kernel’s issues are applicable for TSP too. Due to its nature of embedded and enterprise-grade usage, TSP combines the use of many different C and C++ standard library implementations, along with custom memory management and a modular interface. All these contribute to duplicate definitions for functions and types having general names, such as setup, module, teardown or Transaction. Most of these references can be resolved if the developer, or tool, selects the appropriate definition based on a “file paths look alike” heuristic.

In case of the different standard libraries and memory managers, it is not apparent which implementations are used. The developer must have the knowledge beforehand, or try every possible implementation: some of these program elements have \( \geq 6 \) different definition bodies to the same symbol name.

5. RELATED AND FUTURE WORK

Producing clusters by decomposing large-scale software. Richard C. Holt emphasises on the importance of linkage information when it comes to extracting software models in [Wu and Holt 2004]. Their approach uses the information retrieved from the linker to resolve dangling references in software models created by extractors, such as [Ferenc et al. 2001].

We would like to increase our data by investigating – perhaps with a computer-aided solution – more projects. The previous heuristics mentioned (such as “file paths look alike”) is a topic for further research, to view how powerful such an approach would be in aiding developers and static analysis tools.

Further heuristics should be discussed and developed that could further reduce the list of ambiguous symbols, e.g. parts of the Standard Template Library could be excluded from the ambiguity check process via recording the actual version of the STL used when building the project.

A problem similar to that described can be applied to other programming languages, e.g. in Java, different classpaths could be considered as “linkage” information and the question of “How symbols reference each other?” is applicable for packages built with different configuration in a large-scale project.

6. CONCLUSION

In this paper we investigated the ambiguity of symbol resolution based on mangled names in large C/C++ programs. Resolving symbol references is the base of many essential functionalities – such as navigating to the definition of a type or function, listing all call sites of a given function – implemented in software development tools. Most of the available development environments and comprehension tools implement these features for C/C++ based on mangled names. Unfortunately, using mangled names only may lead to ambiguity when different symbols with the same mangled name are defined in different translation units and assembled into different binaries.
We have shown that while current state of the art tools are ignoring this problem, and on a project level, a considerable ratio (0.5 − 10%) of symbols are affected, the problem can easily be resolved by human developers. On the other hand, automated tools, such as static analysers, could be in need of heuristics, such as source path similarity, to enhance their symbol resolution across translation units.

REFERENCES

Appendix

Fig. 2  Distribution of definition ambiguity amongst symbols that are ambiguously defined. The numbers shown are the node counts.
Comparison of Software Structures in Java and Erlang Programming Languages

ANA VRANKOVIĆ, TIHANA GALINAC GRBAC, University of Rijeka, Faculty of Engineering
MELINDA TÓTH, ELTE Eötvös Loránd University, Budapest, Hungary

Empirical studies on fault behaviour in evolving complex software systems have shown that communication structures among the software entities such as classes, modules, software units and communications among them, is significantly affecting the system fault behaviour. Therefore, we were motivated to further investigate software structures. One interesting question is to investigate software structures from software products written in different programming languages. In this work we present our preliminary study for which we developed tools to examine software structures of software products written in Java and Erlang programming language. We provide details on how we extract software structure from software product and provide preliminary results analyzing four Erlang software products and four Java software products.

Categories and Subject Descriptors: H.2.11 [Software Engineering]: Software Architectures—Languages

General Terms: Software structure

Additional Key Words and Phrases: Network graph, Subgraphs

1. INTRODUCTION

Today, network analysis is used in many scientific fields. It has proven to be useful in numerous problems. In medicine, physics, sociology, electrical engineering, it helped solve diverse issues. In computer science it is used as a tool to understand software behaviour by structuring software dependencies as a network graph. Milo in [Milo et al. 2002] proposed network motifs as patterns in complex networks with higher appearance than in random networks and elaborate its purpose as hidden structural property that can be used in characterizing various complex networks. In our previous study, we analyzed the software structure using network analysis on software written in the Java programming language. For the purpose of network graph extraction we developed the tool rFind [Petric et al. 2014a; Petric and Grbac 2014]. By extracting software graph structures we identified structural changes during the releases of evolving Eclipse software product. In an aim to better understand software behaviour in terms of structural changes, and influence of programming language on obtained conclusions we want to expand our study to other programming languages. In this study, we present preliminary study on the results obtained by analysing software structures in products implemented in Erlang. The subject of this work is to explore whether the programming language has any influence on software structure

This work has been supported in part by Croatian Science Foundations funding of the project UIP-2014-09-7945 and by the University of Rijeka Research Grant 13.09.2.2.16 and by the Hungarian Government through the New National Excellence Program of the Ministry of Human Capacities.

Author's address: A. Vranković, Faculty of engineering, Vukovarska 58, 51000 Rijeka, Croatia; email: avrankovic@riteh.hr; T. Galinac Grbac, Faculty of engineering, Vukovarska 58, 51000 Rijeka, Croatia; email: tgalinac@riteh.hr; M. Tóth, Faculty of Informatics, Pázmány Péter setány 1/C, 1117, Budapest, Hungary; email: tothmelinda@elte.hu

Copyright © by the paper’s authors. Copying permitted only for private and academic purposes.

in terms of network graphs. Like we did in our previous work [Milo et al. 2002; Petric and Grbac 2014; Petric et al. 2014a], we used thirteen subgraph types to study software structure, as presented in Figure 1. These subgraphs cover all three-node connections. All subgraph types present directed graphs where each node is a class/module and every edge is a connection between them. This preliminary study is based on simple comparison of subgraph counts present in software structures obtained from different Erlang and Java software products.

![Fig. 1. Subgraph types](Image)

The rest of the paper is organized as follows. At first, in Section 2 is description of background, in Section 3 we present the used tools to extract the software structures in different programming languages. Then in Section 4 we present the preliminary results obtained by simple comparison of results obtained for Erlang and Java code. Finally, in Section 6 we conclude the paper.

2. BACKGROUND

To analyze software structure we can define different types of structures, modules, classes and different types of software unites. Graph theory is a field of study that looks into the formal description and analysis of graphs [Bullmore and Sporns 2009]. Part of graph theory study are also complex networks, graphs that are based on real world networks, they are discussed in [Simon 1991]. Analyzing system using network graphs and complex networks has been used in many scientific fields for a long time: in medicine, for protein analysis [Aristóteles Góes-Netoa and et al. 2010], in logistics [Carlos PaisMontes and Laxe 2013], crime analyses [Colladon and Remondi 2017], electrical system analyses [Alexandre P. Alves da Silva and Souza 2012] and many more. In computer science there has been a few ideas of using complex networks as a tool to better understand the software behavior. In [S.Jenkins and S.R.Kirk 2007] software architecture graphs were presented as a complex networks using Java written applications. Interesting finding was that as the software ages, more out-going calls than incoming calls are present. In paper [Chong and Lee 2015] complex networks are used as a tool for analyzing the complexity of software system based on object oriented approach. They used a weighted complex network on a system to help them understand its maintainability and reliability. They also managed to identify violations of common software design principals. In [Luis G. Moyanoa and Vargas 2011] the community structure of a real complex software network is explored. The results of this paper shows a significant dependence between community structure and internal dynamical processes. Relationships between Erlang processes have been discussed in [Bozó and Tóth 2016]. No work has been found on comparison between the Java and Erlang software structure. Since Java and Erlang are not similar in paradigm and are not usually used in the similar products, the comparison between them is not often explored. Therefore we wanted to compare them because of their differences to see if there is any variation in the way they communicate in terms of the subgraph type.
3. TOOLS

In this work we used four different tools.

For analyzing Java written applications we used the tool rFind [Petric and Grbac 2014; Petric et al. 2014a]. The input of rFind is an application code written in Java.

As an output we receive two files that allow us to see all calls between classes. One of those files is .classlist where a list of all classes is displayed using class ids for easier reading. Each class represents a node in the network. The other file is a .graph where all connections between class ids are presented. Every connection is viewed as an edge between nodes.

For getting the same information from Erlang applications we used the tool RefactorErl [Bozó et al. 2011]. RefactorErl is an open source static source code analyser and transformer tool for Erlang. RefactorErl supports dependency examination both on module and function level, and is able to present it as a graph to the user. The input of the tool were applications written in Erlang, and it was able to produce a textual representation of dependencies as an output. Although the presentation of the result was quite different from the output of rFind, but the main idea was the same: present communication between modules.

The SuBuCo tool [Petric et al. 2014b] is an application that expects an Rfind output as input: the .classlist and .graph. Then it searches for three-node subgraph structures inside .graph file. Its output is a list of all subgraphs that appear in the given code. The file created contains a list of all subgraphs separated by subgraph type and ids of every class/module contained in specific subgraph.

![Subgraph analysis process graph](image)

Since the output of RefactorErl was not in the form for SuBuCo analysis, we wrote a parser to adjust the result so that it also contains .classlist and .graph. The parser was implemented in Java where the input files were files gathered from RefactorErl and the output files were the two needed files. The whole analysis process can be seen in Figure 2.

4. RESULTS

Our tests were conducted on four different software implemented in the Erlang programming language and four written in Java. The analysed Erlang software are: Mnesia for distributed telecommunications database; Dialyzer that allows static analysis for identifying software discrepancies; Cowboy which is a http server for Erlang/OTP; and RabbitMQ server that runs a multi-protocol messaging
broker. The former two are part of the standard Erlang/OPT distribution, the latter two applications were taken from open git repositories. For analyzing software written in Java we used Java Development Kit (JDT) and Plug-in Development Environment (PDE) projects from Eclipse project, Open Microscopy Environment that is an open-source software and data format standards for the storage and manipulation of biological microscopy data, and Ultimate Android, development framework, from git repository.

Table I. Tested data

<table>
<thead>
<tr>
<th>ERLANG PROJECT</th>
<th>NUMBER OF NODES</th>
<th>NUMBER OF EDGES</th>
<th>LOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mnesia</td>
<td>1914</td>
<td>6092</td>
<td>21417</td>
</tr>
<tr>
<td>Cowboy</td>
<td>510</td>
<td>948</td>
<td>4666</td>
</tr>
<tr>
<td>Dialyzer</td>
<td>1380</td>
<td>4089</td>
<td>14757</td>
</tr>
<tr>
<td>RabbitMQ</td>
<td>3416</td>
<td>6648</td>
<td>23472</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>JAVA PROJECT</th>
<th>NUMBER OF NODES</th>
<th>NUMBER OF EDGES</th>
<th>LOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>OpenMicroscopy</td>
<td>3127</td>
<td>10775</td>
<td>438107</td>
</tr>
<tr>
<td>Ultimate Android</td>
<td>1893</td>
<td>9286</td>
<td>224442</td>
</tr>
<tr>
<td>JDT</td>
<td>3202</td>
<td>16923</td>
<td>606767</td>
</tr>
<tr>
<td>PDE</td>
<td>2542</td>
<td>9834</td>
<td>333390</td>
</tr>
</tbody>
</table>

Table II. Mnesia Results

<table>
<thead>
<tr>
<th>ID</th>
<th>Appearance</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>36</td>
<td>40999</td>
<td>67.24%</td>
</tr>
<tr>
<td>6</td>
<td>13053</td>
<td>21.407%</td>
</tr>
<tr>
<td>12</td>
<td>5994</td>
<td>9.83%</td>
</tr>
<tr>
<td>38</td>
<td>705</td>
<td>1.15623%</td>
</tr>
<tr>
<td>14</td>
<td>177</td>
<td>0.2903%</td>
</tr>
<tr>
<td>74</td>
<td>36</td>
<td>0.059%</td>
</tr>
<tr>
<td>46</td>
<td>6</td>
<td>0.00984%</td>
</tr>
<tr>
<td>98</td>
<td>2</td>
<td>0.00328%</td>
</tr>
<tr>
<td>78</td>
<td>2</td>
<td>0.00328%</td>
</tr>
<tr>
<td>102</td>
<td>0</td>
<td>0%</td>
</tr>
<tr>
<td>238</td>
<td>0</td>
<td>0%</td>
</tr>
<tr>
<td>110</td>
<td>0</td>
<td>0%</td>
</tr>
<tr>
<td>108</td>
<td>0</td>
<td>0%</td>
</tr>
</tbody>
</table>

Table III. Dialyzer Results

<table>
<thead>
<tr>
<th>ID</th>
<th>Appearance</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>24398</td>
<td>51.0664%</td>
</tr>
<tr>
<td>36</td>
<td>16466</td>
<td>34.4643%</td>
</tr>
<tr>
<td>12</td>
<td>5207</td>
<td>10.8986%</td>
</tr>
<tr>
<td>14</td>
<td>926</td>
<td>1.9382%</td>
</tr>
<tr>
<td>38</td>
<td>588</td>
<td>1.2307%</td>
</tr>
<tr>
<td>74</td>
<td>87</td>
<td>0.1821%</td>
</tr>
<tr>
<td>46</td>
<td>52</td>
<td>0.10884%</td>
</tr>
<tr>
<td>98</td>
<td>33</td>
<td>0.0691%</td>
</tr>
<tr>
<td>78</td>
<td>14</td>
<td>0.0293%</td>
</tr>
<tr>
<td>102</td>
<td>4</td>
<td>0.00837%</td>
</tr>
<tr>
<td>108</td>
<td>2</td>
<td>0.004186%</td>
</tr>
<tr>
<td>238</td>
<td>0</td>
<td>0%</td>
</tr>
<tr>
<td>110</td>
<td>0</td>
<td>0%</td>
</tr>
</tbody>
</table>

The number of edges and nodes for each tested software can be seen in Table I. Number of edges seems to be much larger in Java software, even where number of nodes is lesser then in Erlang software. In examples where the number of nodes are similar, Mnesia and Ultimate Android applications, number of edges is still much greater in Java application than in Erlang. Communication is far more common in Java written software. We can see that in all tested applications number of edges grows with the number of nodes.

Subgraph ids discussed in this section are referring to the network subgraphs in Figure 1. In three out of four applications in Erlang subgraph with id 36 was the most common. The same subgraph id also was the most present in both Java projects and in projects gathered from git repositories. It seems that the communication in which multiple classes/modules heavily use one library is the most frequent one. Only one had different results, Dialyzer. We can see from the Tables II-V that in all Erlang applications subgraph ids 36,6 and 12 are the most common ones, most often in that exact order. Subgraph with id 6 presents communications where one node needs multiple resources from other nodes and the subgraph with id 12 could be the situation where communication flows from one node to the other and when the second node is triggered he calls for the third node. In Tables VI-IX.
Table IV. RabbitMQ Results

<table>
<thead>
<tr>
<th>ID</th>
<th>Appearance</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>36</td>
<td>31870</td>
<td>58.0955%</td>
</tr>
<tr>
<td>6</td>
<td>14671</td>
<td>26.7436%</td>
</tr>
<tr>
<td>12</td>
<td>7759</td>
<td>14.1444%</td>
</tr>
<tr>
<td>38</td>
<td>474</td>
<td>0.86405%</td>
</tr>
<tr>
<td>14</td>
<td>66</td>
<td>0.12031%</td>
</tr>
<tr>
<td>74</td>
<td>12</td>
<td>0.021875%</td>
</tr>
<tr>
<td>46</td>
<td>3</td>
<td>0.0054687%</td>
</tr>
<tr>
<td>108</td>
<td>2</td>
<td>0.003646%</td>
</tr>
<tr>
<td>102</td>
<td>1</td>
<td>0.001823%</td>
</tr>
<tr>
<td>98</td>
<td>1</td>
<td>0.001823%</td>
</tr>
<tr>
<td>238</td>
<td>0</td>
<td>0%</td>
</tr>
<tr>
<td>110</td>
<td>0</td>
<td>0%</td>
</tr>
<tr>
<td>78</td>
<td>0</td>
<td>0%</td>
</tr>
</tbody>
</table>

Table V. Cowboy Results

<table>
<thead>
<tr>
<th>ID</th>
<th>Appearance</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>36</td>
<td>1826</td>
<td>38.9755%</td>
</tr>
<tr>
<td>6</td>
<td>1388</td>
<td>29.6286%</td>
</tr>
<tr>
<td>12</td>
<td>1228</td>
<td>26.2113%</td>
</tr>
<tr>
<td>38</td>
<td>156</td>
<td>3.32968%</td>
</tr>
<tr>
<td>14</td>
<td>58</td>
<td>1.235994%</td>
</tr>
<tr>
<td>74</td>
<td>15</td>
<td>0.32017%</td>
</tr>
<tr>
<td>98</td>
<td>1</td>
<td>0.170758%</td>
</tr>
<tr>
<td>46</td>
<td>4</td>
<td>0.085379%</td>
</tr>
<tr>
<td>102</td>
<td>1</td>
<td>0.02135%</td>
</tr>
<tr>
<td>78</td>
<td>1</td>
<td>0.02135%</td>
</tr>
<tr>
<td>238</td>
<td>0</td>
<td>0%</td>
</tr>
<tr>
<td>110</td>
<td>0</td>
<td>0%</td>
</tr>
<tr>
<td>108</td>
<td>0</td>
<td>0%</td>
</tr>
</tbody>
</table>

we can see that Java projects behave similarly. In all projects it is the same order of frequency while in PDE id 38 is present more often than 12.

In terms of subgraph id appearance, we can see that subgraphs with id 238 and 110 do not appear in any of Erlang application and neither in Java applications. Subgraph with ids 102 and 98 were found in Erlang application, but not in any of Java applications. We can see that applications written in Erlang and Java have similar behavior in terms of subgraph id appearance, even though Java software products are greater in class/module size.
Table VI. Ultimate Android Results

<table>
<thead>
<tr>
<th>ID</th>
<th>Appearance</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>36</td>
<td>463879</td>
<td>96.14989%</td>
</tr>
<tr>
<td>6</td>
<td>17338</td>
<td>3.5937105%</td>
</tr>
<tr>
<td>12</td>
<td>924</td>
<td>0.19152%</td>
</tr>
<tr>
<td>38</td>
<td>273</td>
<td>0.0565857%</td>
</tr>
<tr>
<td>14</td>
<td>30</td>
<td>0.00622%</td>
</tr>
<tr>
<td>74</td>
<td>6</td>
<td>0.001244%</td>
</tr>
<tr>
<td>46</td>
<td>4</td>
<td>0.00083%</td>
</tr>
<tr>
<td>102</td>
<td>0</td>
<td>0%</td>
</tr>
<tr>
<td>238</td>
<td>0</td>
<td>0%</td>
</tr>
<tr>
<td>98</td>
<td>0</td>
<td>0%</td>
</tr>
<tr>
<td>110</td>
<td>0</td>
<td>0%</td>
</tr>
<tr>
<td>108</td>
<td>0</td>
<td>0%</td>
</tr>
<tr>
<td>78</td>
<td>0</td>
<td>0%</td>
</tr>
</tbody>
</table>

Table VII. Java Development Tool Results

<table>
<thead>
<tr>
<th>ID</th>
<th>Appearance</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>36</td>
<td>1349284</td>
<td>89.5352%</td>
</tr>
<tr>
<td>6</td>
<td>132736</td>
<td>8.808%</td>
</tr>
<tr>
<td>12</td>
<td>18535</td>
<td>1.2299%</td>
</tr>
<tr>
<td>38</td>
<td>4469</td>
<td>0.29655%</td>
</tr>
<tr>
<td>14</td>
<td>952</td>
<td>0.0632%</td>
</tr>
<tr>
<td>14</td>
<td>951</td>
<td>0.0632%</td>
</tr>
<tr>
<td>78</td>
<td>45</td>
<td>0.00299%</td>
</tr>
<tr>
<td>46</td>
<td>44</td>
<td>0.00292%</td>
</tr>
<tr>
<td>102</td>
<td>0</td>
<td>0%</td>
</tr>
<tr>
<td>238</td>
<td>0</td>
<td>0%</td>
</tr>
<tr>
<td>98</td>
<td>0</td>
<td>0%</td>
</tr>
<tr>
<td>110</td>
<td>0</td>
<td>0%</td>
</tr>
<tr>
<td>108</td>
<td>0</td>
<td>0%</td>
</tr>
</tbody>
</table>

Table VIII. OpenMicroscopy Results

<table>
<thead>
<tr>
<th>ID</th>
<th>Appearance</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>36</td>
<td>802077</td>
<td>92.1518%</td>
</tr>
<tr>
<td>6</td>
<td>54724</td>
<td>6.2873%</td>
</tr>
<tr>
<td>12</td>
<td>10547</td>
<td>1.21176%</td>
</tr>
<tr>
<td>38</td>
<td>2792</td>
<td>0.32078%</td>
</tr>
<tr>
<td>14</td>
<td>167</td>
<td>0.01919%</td>
</tr>
<tr>
<td>74</td>
<td>45</td>
<td>0.00517%</td>
</tr>
<tr>
<td>46</td>
<td>17</td>
<td>0.00195%</td>
</tr>
<tr>
<td>108</td>
<td>14</td>
<td>0.00161%</td>
</tr>
<tr>
<td>78</td>
<td>4</td>
<td>0.00046%</td>
</tr>
<tr>
<td>102</td>
<td>0</td>
<td>0%</td>
</tr>
<tr>
<td>238</td>
<td>0</td>
<td>0%</td>
</tr>
<tr>
<td>98</td>
<td>0</td>
<td>0%</td>
</tr>
<tr>
<td>110</td>
<td>0</td>
<td>0%</td>
</tr>
</tbody>
</table>

Table IX. Plug-in Development Environment Results

<table>
<thead>
<tr>
<th>ID</th>
<th>Appearance</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>36</td>
<td>1064393</td>
<td>96.2593884%</td>
</tr>
<tr>
<td>6</td>
<td>36556</td>
<td>3.309%</td>
</tr>
<tr>
<td>12</td>
<td>3671</td>
<td>0.31199%</td>
</tr>
<tr>
<td>38</td>
<td>1042</td>
<td>0.09433%</td>
</tr>
<tr>
<td>14</td>
<td>81</td>
<td>0.00735%</td>
</tr>
<tr>
<td>74</td>
<td>11</td>
<td>0.0009948%</td>
</tr>
<tr>
<td>46</td>
<td>1</td>
<td>0.00009044%</td>
</tr>
<tr>
<td>102</td>
<td>0</td>
<td>0%</td>
</tr>
<tr>
<td>238</td>
<td>0</td>
<td>0%</td>
</tr>
<tr>
<td>98</td>
<td>0</td>
<td>0%</td>
</tr>
<tr>
<td>110</td>
<td>0</td>
<td>0%</td>
</tr>
<tr>
<td>108</td>
<td>0</td>
<td>0%</td>
</tr>
<tr>
<td>78</td>
<td>0</td>
<td>0%</td>
</tr>
</tbody>
</table>

Looking at Pareto diagrams on Figures 3 and 4 we can see that there is significant growth only for subgraph types 36, 6, 12 and 38 in both Java projects and Erlang projects.

5. THREATS TO VALIDITY

Data collection and analysis is possible on any code written in Erlang or Java. Erlang applications that were tested are server implementations, database and static analytic tool. Java software applications were frameworks for developing Java software and software for working with specific types of data. Software function is not the same in Erlang and Java applications. Since Erlang is a language used for scalable soft real-time systems and Java is general purpose programming language, comparing software applications written in each of them could not give us generalized conclusions. Comparing similar types of languages could be a better approach.

6. CONCLUSION

In this study our main focus was to analyze code structure on software written in Erlang and compare it to the software written in Java.

To do that we used several tools and combined them together to get the appropriate output that we can analyze. We represented class/module communication using thirteen subgraph types.
In code written in Java, there was much larger number of communicating classes in comparison to communicating functions in Erlang code. Subgraph types 38, 36, 6, 46, 12, 74 and 14 were present in all tested code. Types 108 and 78 were present in two software applications. Besides id 46, they are the only ones present that have the number of edges higher than 3. The one with the highest occurrence was subgraph type 36 in every tested application, followed by types 6 and 12. Subgraph ids 102, 238, 98 and 110 did not appear at all. There is no communication where more than four interactions between three classes are existent.

Unlike in Java applications, in Erlang applications subgraph id 98 occurred in all tested applications and id 102 appeared in two tested applications. Subgraph with id 98 is the only one where communication is circular, it starts and ends in the same node with just one interaction between each node. Just as in Java application, in Erlang applications ids 36, 6 and 12 had the highest occurrence but in different percentage. While in Java applications subgraph id 36 occupied over 89% of all subgraphs, in Erlang the same id occupied between 58% and 68% while in Dialyzer it had appearance of only 34%. On tested java software, had a low appearance rate of under 10%. In Erlang applications result was different. Id 6 had a presence of around 20% in Mnesia and RabbitMQ. In Dialyzer, it had the largest number of appearance, 51%.

We can see that in Java written code, subgraph id 36 occupies more than 90% of all the communication while in Erlang code, ids 36 and 6 together occupy 80-90%.

Based on the code analysis, we can conclude that although there is similar behavior between languages, there are some differences. There are structures that appear in Erlang, but not in Java. Specifically structures where there is more communication edges between modules and id 98 where communication is circular. It is possible that those types are specific to that language. There is also a difference
in percentage of the subgraph. While id 36 is in an extensive number of subgraph in Java, types of communication where one library is being heavily used by other classes, in Erlang that number is much lesser. We can see that the usage of libraries is greater in Java programs. There is also a big difference in number of communicating classes/modules. It seems that classes in Java programming language tend to communicate more often than Erlang modules. It is possible that those results are because of the fact that Java is an object oriented language and is based on object communication.

In our future work we aim to do the analysis on code written in other programming and scripting languages, both functional and object oriented. Doing that we can come to the determinant conclusion in aspect of which subgraph types are specific for individual programming languages or applications.

REFERENCES


The Case for Non-Cohesive Packages

NICOLAS ANQUETIL, MUHAMMAD USSMAN BHATTI, STÉPHANE DUCASSE, ANDRE HORA
and JANNIK LAVAL, Inria - Lille Nord Europe – Cristal - University of Lille – CNRS

While the lack of cohesiveness of modules in procedural languages is a good way to identify modules with potential quality problems, we doubt that it is an adequate measure for packages in object-oriented systems. Indeed, mapping procedural metrics to object-oriented systems should take into account the building principles of object-oriented programming: inheritance and late binding. Inheritance offers the possibility to create packages by just extending classes with the necessary increment of behavior. Late binding coupled to the “Hollywood Principle” are a key to build frameworks and let the users branch their extensions in the framework. Therefore, a package extending a framework does not have to be cohesive, since it inherits the framework logic, which is encapsulated in framework packages. In such a case, the correct modularization of an extender application may imply low cohesion for some of the packages. In this paper we confirm these conjectures on various real systems (JHotdraw, Eclipse, JEdit, JFace) using or extending OO frameworks. We carry out a dependency analysis of packages to measure their relation with their framework. The results show that framework dependencies form a considerable portion of the overall package dependencies. This means that non-cohesive packages should not be considered systematically as packages of low quality.

Categories and Subject Descriptors: D.2.8 [Software Engineering] Metrics; D.2.9 [Software Engineering] Software quality assurance

1. INTRODUCTION

Cohesion and coupling principles have been first defined for procedural languages with a black box model of control flow in mind [Stevens et al. 1974; Yourdon and Constantine 1979]. These principles state that a module should have high (internal) cohesion and low (external) coupling so that it implements a well defined functionality and can be easily reused [Briand et al. 1998; Briand et al. 1999].

We argue that when porting the principles to the object-oriented paradigm, some intrinsic properties of the object-oriented paradigm got ignored. We claim that packages with low cohesion are not necessarily packages with low quality. This is due in particular to the presence of inheritance and late-binding, which are the cornerstones of incremental definitions of classes. Classes in one package can extend the behavior of classes defined in other packages and just define a small increment in functionality. Late-binding is also the key mechanism to build white-box frameworks [Pree 1995] using the Hollywood Principle (“don’t call us, we’ll call you”). The application of this principle leads to the situation where an extending package may exhibit low cohesion and still be a well designed package.

In this paper, an experiment is performed on various software systems to understand package cohesion and coupling when they are developed as extensions of frameworks. The analysis is performed by studying package dependencies of extender applications with their frameworks.

The contributions of the paper are the following: (1) identification of a common cohesion/coupling misunderstanding for object-oriented programming; (2) a validation of the hypothesis on several real cases; and (3) a simple model to represent cohesion/coupling in a context of frameworks where no
universally accepted metrics are available (e.g. [Allen and Khoshgoftaar 2001; Bieman and Kang 1995; Ponisio 2006]).

The rest of the paper is structured as follows: Section 2 describes the building principles of OO frameworks and Section 3 presents limitations of existing coupling and cohesion metrics. Section 4 describes our experimental settings. The results are presented in sections 5 and 6. Section 7 discusses the experiment and the results of the paper and Section 8 presents the work related to this study. Finally, Section 9 concludes the paper.

2. LATE-BINDING AND FRAMEWORKS

Late-binding is the fact that the receiver of a message is used to determine the method that will be executed in response to this message. It is the cornerstone of object-oriented programming. Late-binding is the key mechanism behind the Hook and Template Design Patterns [Gamma et al. 1995] and the building of white box frameworks [Fayad et al. 1999; Pree 1995; Roberts and Johnson 1997].

A framework is a set of interacting classes (possibly abstract) defining the context and the logic of behavior. Frameworks are reusable designs of all or part of a software system described by a set of abstract classes and the way instances of those classes collaborate [Roberts and Johnson 1997].

Frameworks are often built around a core package with interfaces and abstract classes. Such core classes interact among themselves to define the logic of the domain and they exhibit hook methods that extender applications will be able to specialize (see Figure 1). These classes can be subclassed/specialized in other framework packages but they are also extended by concrete extender applications.

The idea of cohesive modules is based on the notion of blackbox reuse (Figure 1, left). Inheritance is a white box reuse mechanism, which means a module can be plugged into another module (from which it inherits) without specifying its whole behavior [Fayad et al. 1999; Pree 1995; Roberts and Johnson 1997]. In particular OO frameworks are based on the Hollywood Principle (“don’t call us, we’ll call you”), which supposes that the extender application can “inject” code into the underlying framework through inheritance (Figure 1, right). In this case, the extender application (class D) may call a method offered by an internal class (class C) but actually provided by a framework class (class A). Conversely, the framework (class B) may send a message to an instance that ends up being implemented in the extender application (an instance of class C). Bieman and Kang [Bieman and Kang 1995] recognize two ways to reuse a class: by instantiation and by inheritance. Similarly, we consider that a framework class can be used in these two ways. Defining what is a cohesive module in this case is more complex than for black box reuse.

We will distinguish two layers in the systems studied:

—**Provider Framework**: (abbreviated f/w) any software application that is extended to create another software system. In this sense, the provider framework could be a complete software application.

—**Extender Application**: the application that is being analyzed, this is the subject of the experiment. We consider that the extender application is developed using framework classes.

**Working hypothesis.** There exists in the common perception of package cohesion a rampant misunderstanding: *In an object-oriented language, a well designed package does not have to be necessarily cohesive.*

Our claim is that in the presence of late-binding, weakly cohesive packages are not necessarily of bad quality. Over this article we will refine this claim and show evidence of this fact.
The Case for Non-Cohesive Packages

19:3

Fig. 1. Differences in accessing an underlying library (or framework). In the procedural world (left) the accesses are well known and controlled; in the OO world (right), inheritance and late binding may branch extender classes into the framework, creating new interactions between them.

3. COHESION AND COUPLING OF PACKAGES

Stevens et al. [Stevens et al. 1974], who first introduced coupling in the context of structured development techniques, define coupling as “the measure of the strength of association established by a connection from one module to another”. From then on, the mantra of good design has been that a module (package, class, function) must exhibit high cohesion and low coupling. The importance of this belief in the software engineering culture might be measured by the quantity of work on the topic of measuring cohesion and coupling. In OOP (Object-Oriented Programming) for example, this is attested by the study of Briand et al. [Briand et al. 1999].

In this section, we review existing metrics at the package level and show that they are based on dependencies between the classes of the packages. We will further show that many metrics at the class level are also based on dependencies and those that are not, cannot be made to measure cohesion and coupling at the package level. We will conclude that we are lead to use a metric based on dependencies. We will further examine the existing package metrics and highlight some shortcomings that drove us to dismiss them. These conclusions will serve as bases to choose cohesion/coupling metrics in Section 4.

3.1 Cohesion and coupling at package level

At the level of packages (or groups of classes) the following publications were found to propose cohesion/coupling metrics:

—COF (Coupling Factor) [Brito e Abreu et al. 1995] is a metric defined at program scope, it is a normalized ratio between the number of client relationships and the total number of possible client relationships of the classes. A client relationship exists whenever a class references a method or attribute of another class.

—Bunch [Mitchell and Mancoridis 2006] is a tool that remodularizes automatically software, based on the metrics of module cohesion and coupling defined by the approach. Similarly to COF, cohesion and coupling are defined as a normalized ratio between the existing dependencies of a package’s classes and the maximum number of such dependencies, but Bunch considers incoming and outgoing dependencies.

—Ponisio et al. [Ponisio and Nierstrasz 2006] propose a technique to measure package cohesion by analyzing how client packages access the classes of a given provider package.
A similar approach is presented in [Mišić 2001] that calculates package cohesion based on the fan-in of the contained objects.

Abdeen [Abdeen 2009] presents a set of metrics to capture the quality of a package in the context of an existing structure (set of packages).

Bavota et al. propose to analyze the structural and semantic relationships between classes in a package to define new packages with higher cohesion [Bavota et al. 2010].

With Relational Cohesion [Martin 2002], Martin defines the cohesion of packages as the average number of internal relationships per class, and efferent coupling looks at the number of classes outside the package that classes inside depend upon (Afferent coupling is the number of classes external to the package which depend upon classes in the package).

We conclude that, at the package level, all metrics are based on connectivity between classes inside the package with classes in the same package (cohesion) or in other packages (coupling). This connectivity is computed from dependencies between the classes.

3.2 Cohesion and coupling at class level

There has been a lot of work on cohesion/coupling at the class level too:

With CBO (Coupling Between Object) [Chidamber and Kemerer 1994], two classes are coupled together if one of them uses the other, i.e., one class calls a method or accesses an attribute of the other.

MPC (Message Passing Coupling) [Li and Henry 1993] is defined as the “number of send statements defined in a class”. The authors further refine the definition by indicating that calls to the class’s own methods are excluded from the count, and that only calls from local methods are considered, excluding calls in inherited methods.

The highly debated LCOM (Lack of cohesion of Method) and followers LCOM* [Briand et al. 1998; Chidamber and Kemerer 1994] try to capture class cohesion by measuring how well methods access the class state.

TCC (Tight Class Cohesion) [Bieman and Kang 1995], is the normalized ratio between the number of methods directly connected with other methods through an instance variable and the total number of possible connections between methods.

C3 (Conceptual Cohesion of Classes) captures the semantic information in code for cohesion computation i.e., textual similarity amongst methods [Marcus et al. 2008].

From this, we conclude that, again at the class level, cohesion/coupling metrics are based on some count of dependencies, internal for cohesion, external for coupling: CBO: two classes are coupled if one uses the other; MPC: number of send statements (i.e., method calls).

Some diverging metrics, consider shared dependencies (called sibling dependencies in [Anquetil and Lethbridge 2003]) instead of direct dependencies: TCC: number of methods connected with other methods through an instance variable; and LCOM*: how well methods access the class state. And a final one is based on textual similarity amongst methods: C3.

However, class metrics cannot easily be aggregated at the level of package: a package containing highly cohesive classes could be non-cohesive if each classes deal with a specific topic. A package containing highly coupled classes could be lowly coupled if its classes were all coupled together and not with other packages.
3.3 Cohesion and coupling from a higher point of view

We showed that cohesion/coupling is typically measured from the dependencies that software components have within themselves (cohesion) or with the outside (coupling). Even the research taking a higher point of view on the problem and trying to understand what is meant by cohesion/coupling or design quality (e.g., [Abreu and Goulão 2001; Bhatia and Singh 2006; Counsell et al. 2005; Taube-Schock et al. 2011]) fall back to the same basics: they measure the connectivity between members of the packages.

—Brito e Abreu and Goulão [Abreu and Goulão 2001] and Bhatia and Singh [Bhatia and Singh 2006] do criticize the hegemony of the “high cohesion, low coupling” mantra, but they still define module cohesion as intra-modular class coupling and module coupling […] as inter-modular class coupling, with 12 possible class coupling measures (direct inheritance, class parameter, attribute type, message recipient, …) that can all be summarized as: one member of a class needs to access another class or another class’ member.

—Counsell et al. tried to correlate some software metrics with human perception of class cohesion (for a small set of classes) [Counsell et al. 2005]. The metric that gave best result for expert software engineers was NAS: number of associations. “This metric includes coupling due to inheritance and […] aggregation, the return type of a method or the parameter of a method.”

—Finally, Taube-Schock et al. argue that high coupling cannot be avoided due to the power-law distribution of the connectivity metric [Taube-Schock et al. 2011]. “Connections between source code entities are modelled as links between nodes; these include parent/child relationships, method invocations, superclass/subclass relationships, super-interfaces, type usage, variable usage, and polymorphic relationships.” Again, they use some sort of dependency between classes to compute coupling at a higher level.

4. EXPERIMENT SETTING

Having reviewed existing possibilities to compute cohesion and coupling of packages, we now plan the details of our experiment according to the experimental setup suggested in [Wohlin et al. 2000].

4.1 Experiment definition

Analyze packages

with the purpose of comparing

with respect to their internal and external dependencies

from the point of view of researchers

in the context of real world, framework based, OO applications.

4.2 Context and subjects selection

The context of the experiment will be packages from real, well-designed, OO systems, which are based on a framework and for which the source code is available. The restriction to OO systems based on framework is dictated by our hypothesis. We need real systems to ensure that our experiment is meaningful.

To avoid unexpected bias in the results, it is considered best if subjects are selected randomly inside the entire population. It would be, however, difficult to identify exhaustively all systems that fit the context of our experiment. We will, therefore, rely on convenience sampling.

A difficulty of this research is to better define good and bad modularization. We cannot rely on traditional design metrics (“high cohesion, low coupling”) because our goal is precisely to show that good
modularization may result in low cohesion. So, we will resort to qualitative analysis of the applications and frameworks studied based on their developers' opinion to consider they are well modularized although they might exhibit low cohesion. We will then assume in our formal hypothesis that the quality of the modularization is established, and we will explain how we establish it for each subject system.

We selected the following systems that are known to be based on some framework and for which we found some arguments as to their adequate architectural design: JHotDraw, Eclipse, and JEdit.

**JHotDraw** is itself a framework for structured, two-dimensional, drawing editors. It is developed in Java and is based on the Swing+AWT framework. We analyzed a recent version (i.e., 7.6; 2011-01-09). In the experiment, we will exclude the samples package because it consists of small example applications developed to demonstrate the usage of the JHotDraw framework. As such they do not fit the experiment context. This system is considered well structured for two reasons:

—From its first version, HotDraw (the Smalltalk version) was developed as a “design exercise”. For example, it relies heavily on well-known design patterns. As a consequence, particular attention has been paid on its good design.

—Several notes in the documentation explicitly mention restructurings of the application: v. 7.0.4 (2006-06-25) “Reorganized package structure”; v. 7.0.7 (2006-11-12) “Reorganized file structure”; v. 7.5 (2010-07-28) “Some changes in the package structure and renamings of classes have been made in order to improve the clarity of the frameworks.”

**Eclipse** is an open source platform that is mainly known as an IDE for various languages. We will restrict ourselves to the user interface part (org.eclipse.ui) based on two toolkits: JFace and SWT. To ensure that the application has a good design, we selected version 3.0 of Eclipse (June 2004). This version was the first of the Eclipse Rich Client Platform and was the result of a large restructuring effort.

**JEdit** is a famous text editor for programmers. It is well known in research and served as a test bed for many experiments. Again, we will concentrate only on two specific packages of JEdit that compose the GUI part, org.gjt.sp.jedit.gui and org.gjt.sp.jedit.gui.statusbar. Because they are only two packages with a clear goal (documentation for “gui” states: “Various GUI controls and dialogue boxes”), we will suppose they are correctly designed in the sense that they don’t contain code not related to GUI features.

We give in Table I (page 10) some indication on the size of the three subject systems.

### 4.3 Cohesion and coupling metric selection

We want to study whether, under particular conditions, packages could be well designed although exhibiting poor cohesiveness. But we must first understand that there is no absolute value of a high or low cohesion. There is little meaning in a fully cohesive or fully non-cohesive package. All real, well designed, OO packages will most probably present some level of internal cohesion as well as some coupling with the rest of the system in order to achieve something of significance. How can we measure poor cohesiveness in such situation?

We can define no absolute threshold and there is no known per-system threshold either. We propose to work at the package level, considering that a package does not exhibit “high cohesion and low

1[^http://www.jhotdraw.org/]
2[^http://www.randelshofer.ch/oop/jhotdraw/Documentation/changes.html]
3[^http://www.eclipse.org/]
4[^http://eclipse.org/rcp/generic_workbench_structure.html]
5[^http://www.jedit.org/]
coupling” if its classes, collectively, are more coupled to the outside than to its own components. Conceptually, it is debatable whether this requirement is meaningful at all. One can argue that the two notions are independent. But we showed (Section 3) that the state of the practice usually bases the two metrics on some measure of dependencies between the classes. This makes them very close and even interdependent. One can argue that, given a constant amount of dependencies in a system, if the cohesion of a module increases, its coupling must decrease. In this practical view of cohesion and coupling, it makes sense to compare the two metrics, providing their definitions are based on the same dependencies and they are expressed in the same unit.

As for the metrics themselves, we already explained why existing cohesion/coupling metrics at the class level could not be used at the level of package (Section 3.2). We will also have to reject most of the existing metrics at the level of package:

—The metrics proposed by Ponisio et al. [Ponisio and Nierstrasz 2006] or Mišić et al. [Mišić 2001] are cohesion metrics without an associated coupling metric. So they do not satisfy our requirement of two comparable metrics.
—Martin’s afferent coupling, Abdeen metrics and Bunch metrics use incoming dependencies. This would bias the results against coupling since extender application classes cannot have incoming static dependencies from provider framework classes (see in Section 4.5 why we use static analysis).
—Bunch cohesion and coupling metrics were also found to depend too much on the size of the packages [Anquetil and Laval 2011]. Because coupling normalizes the number of external dependencies found by the maximum number of possible dependencies (ratio of number of external dependencies on number of classes inside times number of classes outside the package), it is impacted by the size of the system: the larger the system, the smaller the coupling.

We are left we two candidates: Martin’s Relational cohesion and Efferent coupling. Unfortunately, they are not expressed in the same unit, Efferent coupling counts the number of external dependencies, while Relational cohesion is the average number of dependencies of the package’s classes. We propose to use Efferent coupling as defined by Martin and Relational cohesion multiplied by the number of classes of the package, i.e., raw number of dependencies between classes within the package. Because we will be comparing between themselves cohesion and coupling of each package, we could as well have used the other solution (pure Relational cohesion and averaged Efferent coupling), the result of the comparison between the two metrics is independent of the package size in both cases. Note that this is very different of the Bunch metrics for which the coupling also depends on the size of the entire system, whereas the cohesion does not (reason why we rejected these two metrics).

The class dependencies we will consider are: inheritance between classes, invocation between methods, access to class attributes, or explicit references to classes (e.g., in the case of a “new”). We will consider dependencies between classes as a boolean property, without considering the possible strength of that dependency. That is to say, one class could inherit from another, call several of its method, access its attributes, we would still count only one dependency.

4.4 Variable selection and Hypothesis formulation

We will use as dependent variable the raw number of dependencies from classes within the studied packages.

The independent variable will be the destination of the dependencies (see Figure 2):

—Local Dependency: The dependency targets a class that resides in the same package as the referencing class. This corresponds to cohesion.
Framework Dependency: The dependency targets a class that resides in the framework. This corresponds to coupling with the provider framework.

We already introduced informally the experiment hypothesis, saying that we want to test whether package cohesion is inferior to package coupling. This is a strong hypothesis because one could assume that even if the cohesion of a package were only a little superior to its coupling, it would not qualify as “high cohesion, low coupling”. We will actually need to make it even stronger, because we are only interested in the coupling with the provider framework.

From this, we can now formalize the following null and alternative hypotheses:

\[ H_0: \text{package-local-dependencies} \geq \text{package-framework-dependencies} \]

\[ H_a: \text{package-local-dependencies} < \text{package-framework-dependencies} \]

4.5 Experiment design and instrumentation

The test will compare one variable (number of dependencies) on two treatments (local or framework dependencies). Both treatments apply to all the subjects (packages in the systems studied): for each package, we will look at its local dependencies and dependencies to the framework. This is a paired design.

—We will use static analysis to obtain the data related to the dependencies;
—We will use the Moose environment and its accompanying tools to gather the metrics for Java programs [Ducasse et al. 2005];
—In the analysis of results, we will distinguish inheritance from the other dependencies.

About static analysis, in Section 2, we introduced our working hypothesis in relation to late-binding used in OO framework. It is the goal of late-binding to identify the called method only at execution time according to the instance that receives the message. Therefore in the presence of late-binding, static analysis can have problems to identify the exact method invoked. For Java, the class implementing an invoked method is computed from the declared type of variables. This may give wrong results if the variable actually contains an instance of a sub-class of its declared type.
Yet we chose to use static analysis because the alternative (dynamic analysis) presented other problems. Dynamic analysis requires to define:

—Realistic usage scenarios: In our experiment this was not always easy, for example, when considering JHotDraw, which is not an application, but a framework itself and therefore cannot run on its own. We also experimented with just a selection of packages of the systems. In such cases, it is difficult to elaborate realistic scenarios that will use the framework and the part of the system we are interested in.

—Usage scenarios covering the whole application: We need usage scenarios in sufficient number to trigger all the features of the target application, so as to execute all the code and not miss any dependency. With complex applications and/or frameworks as our subjects, it is difficult to ensure that completeness.

About the Moose environment, we chose it because it provides tools to extract code information for different languages (including Java) and offers the needed infrastructure. Moose tools rely on the Famix meta-model to represent source code. It includes packages, classes, attributes, methods and their associated relationships [Demeyer et al. 2001]. We will therefore consider the following dependencies: method invocation, field access, class reference (e.g. “Collections.sort( . . . )”), and inheritance.

Finally, we distinguish inheritance from the other dependencies because inheritance is a strongly structuring relationship in OOP. We will term these other relationships (field access, method invocation, class references) “accesses”. For dependency counting, when inheritance between two classes exists, we ignore the other dependencies (accesses) between the two classes. Therefore, the total dependencies between two classes is not the sum of inheritance links and accesses.

4.6 Validity evaluation

We did not identify any conclusion validity threats. Validity of the results will be tested using the appropriate statistical test. Because the data do not follow a Normal Distribution, we will test our hypothesis using a one-tail Wilcoxon test.

We identified the following construct validity threats: our definition of low cohesion (local dependency is less than framework dependency) is a limited, syntactical, view on cohesion that does not fully measure what people mean by “high cohesion”, however this simplification is an accepted trade-off in architectural design measurement (see also Section 4.3).

We did not identify any internal validity threats to the experiment.

We identified the following external validity threats: We had to rely on convenience selection of subjects. As such, all our subjects were identified as being Java systems and based on UI frameworks (Swing, AWT, SWT). These two characteristics may be partly responsible for our results, for some unknown reason. We, however, do not see any reason why this could be the case. Another problem with our subject systems is that Eclipse represents about two thirds of the data point (see Section 5). We tried to compensate for this possible bias by presenting individual experimental results for each system studied.

Another possible external validity threat is that we had to accept an informal, qualitative, definition of “good architectural design” of the subject systems, based on their developers opinion. This was necessary because our goal was precisely to show that good modularization may result in low measured cohesion.
5. EXPERIMENT RESULTS AND ANALYSIS

Table I lists some size metric results for the three systems considered. One can observe a dissimilarity in importance of each systems, with Eclipse weighting 65% of all the packages and 71% of local+framework dependencies.

<table>
<thead>
<tr>
<th></th>
<th>JHotDraw</th>
<th>Eclipse (UI)</th>
<th>JEdit (GUI)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Packages</td>
<td>41</td>
<td>80</td>
<td>2</td>
</tr>
<tr>
<td>Classes</td>
<td>430</td>
<td>1,540</td>
<td>97</td>
</tr>
<tr>
<td>Lines Of Code</td>
<td>88,452</td>
<td>311,744</td>
<td>24,791</td>
</tr>
<tr>
<td>Dependencies</td>
<td>2,952</td>
<td>10,626</td>
<td>1,371</td>
</tr>
<tr>
<td>Local dep.</td>
<td>700</td>
<td>3,024</td>
<td>276</td>
</tr>
<tr>
<td>Framework dep.</td>
<td>2,252</td>
<td>7,602</td>
<td>1,095</td>
</tr>
</tbody>
</table>

“Dependencies” include the local and framework dependencies.

Figure 3 gives the box plot for the two tested variables. A box plot shows the shape of the distribution through five number summary of the data under analysis. The box is drawn from 25th percentile (on the left side) to 75th percentile (on the right side) values. The band in the middle depicts the median of the data. The whiskers on both sides of the box contain the non-outlier points i.e., the data points within the distance of 1.5 times the box size. Beyond non-outliers, data points are outliers: the values that are regarded as “too far” from the central value. In the figure, the 75th percentile value for framework dependencies is higher than those for local dependencies, thus supporting our hypothesis.

Table II gives a summary of some descriptive statistics for our experiment. There are 123 packages in the three systems considered. The distribution for the two dependent variables is not normal, it is skewed to the right (i.e. more data on the left, long tail on the right). For example the median is less than the mean and skewness is high. This is also confirmed by the box plots (Figure 3). For information, Spearman rank correlation coefficient shows that the two variables are positively correlated ($r = 0.80$): packages that have more dependencies within themselves also tend to have more dependencies with the underlying framework.

We also see that mean and median number of local dependencies ($mean = 32.5$, $median = 13$) are lower than for framework dependencies ($mean = 89$, $median = 19$), which again fits our hypothesis (packages have more dependencies toward their respective framework than inside themselves). This will need to be formally tested.
Table II. Descriptive statistics for our experiment with packages from Eclipse (user interface), JEdit (GUI) and JHotDraw

<table>
<thead>
<tr>
<th></th>
<th>local dependencies</th>
<th>fw dependencies</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mean</td>
<td>32.5</td>
<td>89.0</td>
</tr>
<tr>
<td>Median</td>
<td>13</td>
<td>19</td>
</tr>
<tr>
<td>Standard Deviation</td>
<td>55.5</td>
<td>172.4</td>
</tr>
<tr>
<td>Skewness</td>
<td>4.2</td>
<td>3.4</td>
</tr>
<tr>
<td>Minimum</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Maximum</td>
<td>436</td>
<td>977</td>
</tr>
<tr>
<td>Correlation</td>
<td>0.80</td>
<td></td>
</tr>
</tbody>
</table>

Total packages: 123

Because data are not normally distributed, we use the Wilcoxon test to test our hypothesis. As this test is known to be resistant to outliers, we will not take any special precaution for them and include all data in the test. The result of the test \((W = 3109.5, p = 1.3E-10)\) allow us to reject the null hypothesis with high confidence. We may therefore accept the alternative hypothesis that packages have more dependencies to the underlying framework than within themselves.

This experiment confirms that, for these systems, that are well designed, packages are not cohesive according to a definition of cohesiveness that is compatible with the ones used in literature.

6. DETAILED EXPERIMENTS

In this section, we present the results of additional experiments. One goal of these more detailed experiments is to evaluate the influence of Eclipse in the results of the previous section (as described in the threats to validity). We will therefore study each system independently and even parts of them. For each experiment, we define the “extender application” considered and its provider framework.

These experiments will also serve to gain a more in-depth understanding. For this we consider two additional dependent variables (inheritance relationships, and access dependencies = attribute access + method invocations + direct class references). The dependent variable used in the main experiment will be re-termed “total dependencies”.

6.1 JHotDraw (f/w=Swing+AWT)

Table III presents the results of the experiment. Note that Total dependencies is not the result of Inheritance+Access because we count multiple dependencies from one class to another as just one (see Section 4.5).

For JHotDraw, total number of local dependencies (700) is inferior to the dependencies to the framework (2,252) and the Wilcoxon test \((W = 58.5, p = 2E-06)\) allows us to reject the null hypothesis. We conclude that results for the main experiment were not solely caused by the preponderance of the Eclipse system since JHotDraw exhibits the same property.

Table III. Total Local and Framework Dependencies for JHotDraw (f/w Swing+AWT)

<table>
<thead>
<tr>
<th></th>
<th>Local</th>
<th>Framework</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inheritance</td>
<td>267 / 39.5%</td>
<td>290 / 42.9%</td>
</tr>
<tr>
<td>Access</td>
<td>700 / 17.6%</td>
<td>2,091 / 52.5%</td>
</tr>
<tr>
<td>Total Dep.</td>
<td>700 / 16.5%</td>
<td>2,252 / 53.2%</td>
</tr>
</tbody>
</table>

Because JHotDraw is well organized, we will single out some of its main parts. Figure 4 illustrates the individual dependencies of the JHotDraw packages. Most of the packages can be combined into
four groups based on their name as shown in the figure: app, GUI, draw and utils. Informally, one can see that in all the four groups, the total count of framework dependencies (gray) outnumber the count of local dependencies (black).

We now study more in depth the org.jhotdraw.gui package and its sub-packages to better illustrate how packages that are well designed may end up depending a lot on an underlying framework. For the GUI group of packages, the description of the main package org.jhotdraw.gui says: “provides general purpose graphical user interface classes leveraging the javax.swing package”. It seems only natural that it depends more on the framework (Swing+AWT).

Table IV presents the results of the experiment restricted to this set of packages. There are 8 packages in this experiment. Again, the number of local dependencies (148) is largely inferior to the dependencies to the framework (641) and the Wilcoxon test ($W = 2, p = 0.047$) allows us to reject the null hypothesis.

<table>
<thead>
<tr>
<th></th>
<th>Local</th>
<th>Framework</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inheritance</td>
<td>14 / 9.4%</td>
<td>131 / 87.9%</td>
</tr>
<tr>
<td>Access</td>
<td>148 / 19.7%</td>
<td>547 / 72.6%</td>
</tr>
<tr>
<td>Total Dep.</td>
<td>148 / 17.4%</td>
<td>641 / 75.4%</td>
</tr>
</tbody>
</table>

Looking deeper into this group of packages (again Figure 4), we see org.jhotdraw.gui.plaf.palette standing out as very dependent on the framework. We show the inheritance hierarchy of the classes
for this package in Figure 5. In the figure, black nodes represent framework classes and grey nodes represent local classes; white nodes represent classes in the other packages of JHotDraw. The class at the top is `java.lang.Object` and is included for illustration purposes. It should not be counted as a framework class. This figure illustrates well our working hypothesis by showing that many classes, local to this package (in grey), inherit from a black class (in the framework). Considering only inheritance dependencies, 19 local classes inherit directly from framework ones, 2 others inherit from application classes (in white) that themselves inherit from framework classes, and only six do not have framework classes in their ancestors.

![Fig. 5. Inheritance Hierarchy for the package gui.plaf.palette](image)

6.2 JHotDraw Samples (f/w: JHotDraw + Swing + AWT)

JHotDraw is itself developed as a framework so that applications can be developed by extending its classes and interfaces. For this purpose, some samples have been developed to demonstrate its usage. We analyze here these samples (grouped in the `org.jhotdraw.samples` package), where JHotDraw + Swing + AWT serve as provider framework. We hypothesize that the quality of these samples is equal or superior to that of JHotDraw because the samples are smaller and receive much less maintenance (these are not actual applications used by someone).
Table V summarizes the results of this analysis. Again, the total number of local dependencies (323) is largely inferior to the dependencies to the framework (2076) which supports our formal hypothesis. The Wilcoxon test gives a p-value=0.047 (under the 5% acceptance level) confirming the statistical validity of this result.

Table V. Total Local and Framework Dependencies for JHotDraw Samples (f/w: JHotDraw + Swing + AWT)

<table>
<thead>
<tr>
<th></th>
<th>Local</th>
<th>Framework</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inheritance</td>
<td>37 / 16.2%</td>
<td>192 / 83.8%</td>
</tr>
<tr>
<td>Access</td>
<td>323 / 13.6%</td>
<td>1949 / 82.3%</td>
</tr>
<tr>
<td>Total Dep.</td>
<td>323 / 12.9%</td>
<td>2076 / 83.2%</td>
</tr>
</tbody>
</table>

As expected, org.jhotdraw.samples demonstrates strong dependency relationship with the provider framework.

6.3 JEdit (f/w: Swing+AWT)

Table VI presents the results of the experiment with JEdit. Total number of local dependencies (1149) is well below the dependencies to the framework (2708), yet the Wilcoxon test gives a p-value=0.432 which does not allow us to reject the null hypothesis that the actual average number of local dependencies is more than the average number of framework dependencies.

Considering the large difference in numbers this seems surprising, but looking closer to the results, we see that two packages: org.gjt.sp.jedit.gui and org.gjt.sp.jedit.gui.statusbar cumulate 1095 dependencies to the framework, more than 45% of all the framework dependencies. If we look at these two packages, we obtain a total of 276 local dependencies and four times as much (1095) framework dependencies. Because they are only two packages, with a clear goal (documentation for "gui" states: “Various GUI controls and dialogue boxes”), we will suppose they are correctly designed in the sense that they don’t contain code not related to GUI features. We claim these two packages support our working hypothesis. Because there are only two packages, it would be meaningless to test the statistical validity of this conclusion.

6.4 Eclipse-UI (f/w: JFace+SWT)

Table VII presents the results of the experiment conducted on Eclipse-UI. Total number of local dependencies (3024) is well below the dependencies to the framework (7602), and the Wilcoxon test confirms the statistical significance of this result (p-value=4E-06), showing that Eclipse-UI supports our hypothesis.
Table VII. Total Local and Framework Dependencies for Eclipse (f/w: JFace+SWT)

<table>
<thead>
<tr>
<th></th>
<th>Local</th>
<th>Framework</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inheritance</td>
<td>627 / 26.1%</td>
<td>1261 / 52.3%</td>
</tr>
<tr>
<td>Access</td>
<td>5480 / 33.4%</td>
<td>6649 / 40.5%</td>
</tr>
<tr>
<td>Total Dep.</td>
<td>3024 / 19.8%</td>
<td>7602 / 49.7%</td>
</tr>
</tbody>
</table>

6.5 JFace (f/w: SWT)

As part of the Eclipse experiment, we also analyzed JFace with regard to its underlying framework: SWT. The JFace UI toolkit is an extension of SWT as implicitly indicated by the documentation:\(^6\) “the only prerequisites for JFace [were] reduced to SWT”.

Table VIII presents the results of the experiment. Total number of local dependencies (914) is below the dependencies to the framework (1539), and the Wilcoxon test confirms the statistical significance of this result (p-value=0.017).

![Table VIII. Total Local and Framework Dependencies for JFace (f/w: SWT)](http://en.wikipedia.org/wiki/Swing_(Java)#Relationship_to_AWT)

Table VIII. Total Local and Framework Dependencies for JFace (f/w: SWT)

<table>
<thead>
<tr>
<th></th>
<th>Local</th>
<th>Framework</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inheritance</td>
<td>305 / 44.3%</td>
<td>213 / 30.9%</td>
</tr>
<tr>
<td>Access</td>
<td>1593 / 43.4%</td>
<td>1331 / 36.3%</td>
</tr>
<tr>
<td>Total Dep.</td>
<td>914 / 27.5%</td>
<td>1539 / 46.3%</td>
</tr>
</tbody>
</table>

6.6 Swing (f/w: AWT)

Swing\(^7\) is the GUI toolkit of java. Java has another, prior, GUI toolkit: AWT. Swing inherits and use several of the classes of AWT, however AWT may not be seen as an underlying framework for Swing\(^8\). For example, AWT uses the native, operating system-supplied, widgets (menus, windows, buttons, ...) whereas Swing elements are entirely written in Java with no native code. We thought it would be interesting to do this experiment to see whether the good-design/low-cohesion property also holds in more general cases. We have no indication of the design quality of Swing, but this is not an issue in this case (see below).

The results of the analysis are presented in Table IX. Local dependencies (3492) are superior to framework dependencies (1815). There is no need to test the significance of this result since it does not support our formal hypothesis.

![Table IX. Total Local and Framework Dependencies for Swing (f/w: AWT)](http://en.wikipedia.org/wiki/Swing_(Java)#Relationship_to_AWT)

Table IX. Total Local and Framework Dependencies for Swing (f/w: AWT)

<table>
<thead>
<tr>
<th></th>
<th>Local</th>
<th>Framework</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inheritance</td>
<td>595 / 38.9%</td>
<td>257 / 16.8%</td>
</tr>
<tr>
<td>Access</td>
<td>2897 / 43.1%</td>
<td>1558 / 23.2%</td>
</tr>
<tr>
<td>Total Dep.</td>
<td>3492 / 42.3%</td>
<td>1815 / 22.0%</td>
</tr>
</tbody>
</table>

---

\(^6\)http://wiki.eclipse.org/index.php/JFace

\(^7\)http://java.sun.com/javase/technologies/desktop/

\(^8\)http://en.wikipedia.org/wiki/Swing_(Java)#Relationship_to_AWT
7. HYPOTHESIS TESTING AND DISCUSSION

We now summarize the results of all our experiments with regard to the initial working hypothesis.

In the systems studied, which are based on an OO framework, experimental data support our hypothesis that a well designed system (based on an OO framework) may have higher coupling than cohesion. This translated to packages having more dependencies towards classes in their underlying framework than their own classes.

When the numbers do not agree with our hypothesis as in the case of JEdit, we can argue that:

— We do not pretend that all well-designed packages will have this property, only that it is a possibility that one must take into account. In this case, JEdit has some packages, possibly well-designed, that do not depend on Swing. This should not be a surprise.

— We did find that the two GUI packages of JEdit have lower internal cohesion than coupling toward the underlying Swing framework. Again, this seems only natural, and concur with our hypothesis.

For Swing and AWT, again cohesion was higher than coupling towards the external library. We performed this experiment because AWT is not a framework on which Swing is based. This might be an indication that the scope of our good-design/low-cohesion rule is restricted to the case where a framework is used.

As already mentioned, our experiments targeted GUI related frameworks (Swing, SWT) which can be a bias of the study. GUI framework are among the more widely used and known because so many applications need to use one of them. It would be interesting to perform the same experiment with frameworks in more restricted domains: web application frameworks (e.g. Struts); ORB frameworks (providing implementation of Corba or RMI), etc. These examples are still generic and may apply to any business domain, other application domain frameworks may also be studied. Some systems use annotations to inject dependencies and the impact of the annotation dependencies needs to be studied through further experimentation.

8. RELATED WORK

Related work to the paper in the domain of program metrics is already presented in Section 3.

Dong and Godfrey [Dong and Godfrey 2007] propose a visualization approach to understand the usage dependencies between aggregated components formed through static object descriptions. The study does not take into account library (framework) classes and primarily serves as an approach for program understanding. Package blueprint takes the focus of a package and shows how such packages use and are used by other packages [Ducasse et al. 2007].

Abreu and Goulão demonstrate that the criteria used by practitioners to modularize object-oriented systems falls short to achieve the objective of minimal coupling and maximal cohesion [Abreu and Goulão 2001]. Anquetil and Laval present a study that shows that package cohesion and coupling metrics do not demonstrate the inverse correlation for successive restructured versions of eclipse platform [Anquetil and Laval 2011]. Furthermore, they describe that cohesion and coupling degrades with restructuring that aim to improve system structure. Recently Taube-Schock et al. conclude that high coupling is not avoidable and some high coupling is necessary for good design [Taube-Schock et al. 2011].

9. CONCLUSION

Cohesion and coupling is described for modules in the procedural paradigm to gauge the design quality of modules. A program is deemed to have a good design if it exhibits low (external) coupling and high (internal) cohesion. These concepts are then ported to the object-oriented paradigm. However, the notion of cohesion and coupling in object-oriented programs is not similar to procedural programs because
of the hollywood principle and late binding, which allow to leverage the benefits of the object-oriented paradigm. Object-oriented programs are developed on top of frameworks and this relationship results in tight coupling between an extender application and its underlying framework. In this paper, we present a study of package dependencies in object-oriented programs, which serves as a quantitative evidence that application is coupled to its underlying framework. Strong dependency of the extender application on its framework is perfectly acceptable. We believe that while calculating the package metrics for cohesion and coupling, these factors should be taken into account. Moreover, while deducing the results of these metrics, high coupling and low cohesion should not be considered harmful so that the measures reflect the reality of the object-oriented paradigm. We plan to continue investigating this phenomenon further so that adequate measure for object-oriented application can be developed to ascertain their design quality appropriately.

REFERENCES


Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional.


