



SPLST'15

Jyrki Nummenmaa, Outi Sievi-Korte, Erkki Mäkinen (editors)

Proceedings of the 14th Symposium on
Programming Languages and Software Tools

University of Tampere
School of Information Sciences

Tampere, October 9-10, 2015

Preface

This volume contains the papers of SPLST'15: 14th Symposium on Programming Languages and Software Tools held on October 9-10, 2015, in Tampere, Finland.

The symposium series started in Szeged, Hungary, in 1989, and since then, the symposium has been established as a bi-annual event. From the initial Finno-Ugric event, the symposia have developed into a conference series aimed to attract scientist from other countries as well, even though the majority of the authors still come from the Estonia, Finland, and Hungary - the countries where the symposia is normally organized.

The original profile "Programming Languages and Software Tools" has maintained its importance over the times, with the research topics evolving over time. This year's topics included metrics, metrics and testing, software tools, data types and structures, security, data collection and analysis, and products, models, and requirements.

The conference papers were selected through peer reviewing. Each paper had 2-3 reviews and the selection was strictly based on the outcome of the reviews. The paper submission, reviews, selection, and proceedings production were all performed with the help of the EasyChair system.

In addition to the peer-reviewed papers, the conference programme included two invited key note talks: One by prof. Aarne Ranta, University of Gothenburg, on the topic "Grammars for the Working Programmer: GF and BNFC" and the other one by Dr. Veli-Pekka Eloranta, from software company Vincit, on the topic "From trenches: Software development for medical devices".

The editors want to express their gratitude to the Program Committee and the external evaluators, and to the University of Tampere for providing the facilities for the conference.

September 28, 2015
Tampere

Jyrki Nummenmaa
Outi Sievi-Korte
Erkki Mäkinen

Steering Committee

Kai Koskimies	Tampere University of Technology (retired)
Jaan Penjam	Tallinn University of Technology
Horváth Zoltán	Eötvös Loránd University, Budapest

Program Committee

Eleni Berki	University of Tampere
Hassan Charaf	Budapest University of Technology and Economics
Tibor Gyimothy	University of Szeged
Pekka Kilpeläinen	University of Eastern Finland
Akos Kiss	University of Szeged
Tamás Kozsik	Eötvös Loránd University, Budapest
Ville Leppänen	University of Turku
Tommi Mikkonen	Tampere University of Technology
Erkki Mäkinen	University of Tampere
Jyrki Nummenmaa	University of Tampere
András Pataricza	Budapest University of Technology and Economics
Jari Peltonen	Cometa Solutions Oy
Jaan Penjam	Tallinn University of Technology
Attila Pethő	University of Debrecen
Outi Sievi-Korte	Tampere University of Technology
Antti Tapani Siirtola	University of Oulu
Kari Systä	Tampere University of Technology
Antti Valmari	Tampere University of Technology
Margus Veanes	Microsoft Research
Horváth Zoltán	Eötvös Loránd University, Budapest

Table of Contents

Metrics

Towards Proactive Management of Technical Debt by Software Metrics	1
<i>Anna Sandberg, Mirosław Staron and Vard Antinyan</i>	
Defining Metrics for Continuous Delivery and Deployment Pipeline	16
<i>Timo Lehtonen, Sampo Suonsyrjä, Terhi Kilamo and Tommi Mikkonen</i>	
Metrics for Gerrit Code Review	31
<i>Samuel Lehtonen and Timo Poranen</i>	

Metrics and testing

Test Suite Evaluation using Code Coverage Based Metrics	46
<i>Ferenc Horváth, Béla Vancsics, László Vidács, Árpád Beszédes, Dávid Tengeri, Tamás Gergely and Tibor Gyimóthy</i>	
Accounting Testing in Software Cost Estimation: A Case Study of the Current Practice and Impacts	61
<i>Jurka Rahikkala, Sami Hyrynsalmi and Ville Leppänen</i>	

Tools

ICDO: Integrated Cloud-based Development Tool for DevOps	76
<i>Farshad Ahmadighohandizi and Kari Systä</i>	
A State Space Tool for Concurrent System Models Expressed In C++	91
<i>Antti Valmari</i>	
Semantics analyzing expression editors in IP-XACT design tool Kactus2	106
<i>Mikko Teuvo, Esko Pekkarinen and Timo Hämäläinen</i>	

Products, models, and requirements

Internal Marketplace as a Mechanism for Promoting Software Reuse	119
<i>Maria Ripatti, Terhi Kilamo, Karri-Tuomas Salli and Tommi Mikkonen</i>	
Lean Startup Meets Software Product Lines: Survival of the Fittest or Letting Products Bloom?	134
<i>Henri Terho, Sampo Suonsyrjä, Ari Jaaksi, Tommi Mikkonen, Rick Kazman and Hong-Mei Chen</i>	
Model-based technology of software development in large	149
<i>Enn Tyugu and Jaan Penjam</i>	

Requirements management in GitHub with lean approach	164
<i>Risto Salo, Timo Poranen and Zheyang Zhang</i>	
Data types and structures	
Priority Queue Classes with Priority Update	179
<i>Matti Rintala and Antti Valmari</i>	
Two set-based implementations of quotients in type theory	194
<i>Niccolò Veltri</i>	
Security	
Preventing malicious attacks by diversifying Linux shell commands	206
<i>Joni Uitto, Sampsa Rauti, Jari-Matti Mäkelä and Ville Leppänen</i>	
Phishing Knowledge based User Modelling in Software Design	221
<i>Linfeng Li, Timo Nummenmaa, Eleni Berki and Marko Helenius</i>	
Securing Scrum for VAHTI	236
<i>Kalle Rindell, Sami Hyrynsalmi and Ville Leppänen</i>	
Data collection and analysis	
Collecting Issue Management Data for Analysis with a Unified Model and API Descriptions	251
<i>Otto Hylli, Anna-Liisa Mattila and Kari Systä</i>	
LOGDIG log file analyzer for mining expected behavior from log files	266
<i>Esa Heikkinen and Timo D. Hämäläinen</i>	
Mining Knowledge on Technical Debt Propagation	281
<i>Tomi 'Bgt' Suovuo, Johannes Holvitie, Jouni Smed and Ville Leppänen</i>	
Pattern recognition with Spiking Neural Networks: a simple training method	296
<i>Francois Christophe, Tommi Mikkonen, Vafa Andalibi, Kai Koskimies and Teemu Laukkarinen</i>	

Towards Proactive Management of Technical Debt by Software Metrics

Anna Sandberg¹, Mirosław Staron², Vard Antinyan²

¹*Ericsson*

Goteborg, Sweden

anna.sandberg@ericsson.com

²*University of Gothenburg*

Hörselgängen 11, Göteborg, Sweden

vard.antinyan@cse.gu.se, miroslaw.staron@cse.gu.se

Abstract. Large software development organizations put enormous amount of effort not only for responding to continuous requests of customers but also for reengineering and refactoring activities to keep their product maintainable. Often rapid and immature feature deliveries over long period of time gradually decrease the product quality, and therefore the refactoring activities become costly and effort-intensive. This situation is described by the concept of “technical debt”, which represents the accumulated rework that organization has to do in order to prevent the slowdown of the development. In this paper we report results of a case study at Ericsson on using software metrics for moving towards proactive management of technical debt. Our observations show that there are four distinguishable maturity phases of quality management over the eight years of development time of two large products: Start-n-stop, Reactive, Systematic, and Proactive quality management. Three sophisticated metrics are applied to help the organizations to move towards Proactive management of technical debt. These metrics are used on a systematic basis to provide information on the areas of the product that have tendency of accumulating technical debt. Software engineers use this information for making decisions on whether or not the pinpointed areas should be refactored.

Keywords: Software development, Software metrics, Software technical debt

1 Introduction

Large software developing organizations want to spend their time on feature development and innovations to be competitive and profitable. In practice, far too many instead spend a substantial part of their time on managing non-feature-delivering activities, such as defect handling, refactoring, and scope-cutting. This inefficient situation can be explained with help of “technical debt”, which is a known metaphor [1], but has still gained too little practical attention in the software development industry. Gradually accumulated technical debt in a long period of time can reach to scales that

require enormous effort for its control and management. Furthermore, it obliges organizations to stop feature development activities from time to time and focus only on defect handling and refactoring. This kind of development is labeled at Ericsson as Start-n-stop development with Start-n-stop quality management. In contrast with Start-n-stop development, every organization aims to move towards Proactive development, described by proactive management of technical debt in parallel with continuous feature delivery. Continuous delivery allows delivering new features to the customers all the time, thus making the organization competitive in the market [2] [3]. However, gradual increments of technical debt over long period of time slow down the development process. This situation requires continuous quality management as well which permit continuous control of technical debt. Approaches for technical debt management exist as well [4, 5]. However there is scarce data reported on continuous technical debt management, which ultimately permits Proactive development. The research question we raise in this paper is:

How can we use software metrics to proactively manage technical debt in a large software development organization?

The telecom company Ericsson has since many years worked with metrics to overcome the challenges around technical debt. In this article we have documented these experiences to visualize and present how large software development organizations can increase their understanding and manage the technical debt in four different maturity phases: Start-n-stop development, Reactive development, Systematic development and Proactive development. Proactive development is the want-to-be phase with its capability of proactive quality management and comparably high attention to feature development. We show how three sophisticated metrics can guide software organizations to proactively manage technical debt with the goal of spending valuable development time on feature-delivering activities and innovations.

2 Technical Debt Challenges

The price to pay for software development includes more than new features with added functionality or increased performance. Too often, poor quality, stemming out from rapid immature feature delivery, requires costly reworks during later software releases. In other words, if a customer has \$100 to spend, she cannot buy features for all of it. She has to spend a part of it on architectural features and defect handling. The size of this part is dependent on the size she spends in the previous releases. The challenge is to accept that fact and include long-term value thinking when managing her ongoing software development. A similar and understandable metaphor is linked to paying interests [1]. If she lends \$100 to zero % the first quarter, it is easy to understand that she will need to pay much more in the upcoming quarters.

Krutchén et al. [6] has divided the software development content to four categories based on the (non-) feature-delivering (i.e. negative and positive value) and visibility (see Figure 1). The guiding principle is – *what is possible to see is possible to manage*, especially when bringing a positive value. All software organizations tend to pay

attention to features and defects just because they are visible. It is the invisibility and lack of tangible customer value, which characterizes the *technical debt* that makes it harder for the software organizations to manage it. Therefore, it is extremely important to measure and visualize how technical debt can grow over time and latterly eat up the capacity for feature delivery if not kept under control.

	Visible	Invisible
Positive Value	New features Added functionality	Architectural, Structural features
Negative Value	Defects	Technical Debt

Figure 1 Technical Debt as an invisible negative value (Kruchten et al. 2012).

Over time Ericsson has used a variety of software metrics to manage different aspects of technical debt [7-9]. We investigated how software metrics are used in the studied organization which helped the organization move from Start-n-stop development to Proactive development.

3 Research Approach

The studied Ericsson case consists of two different products, each containing several millions lines of code developed over a 20 years period. Both products are developed in global multi-site environments with development sites in three different continents. Each development organization has ~500 developers. The general interest in software metrics at Ericsson has been high during a longer period [10], which has driven the interest of the organization to dive into challenging, but beneficial software metrics areas such as technical debt.

We studied metrics used by the two products over an eight years period. We used triangulation of multiple data sources (document analysis, observations and literature studies) in several iterations, so in the end we could identify four typical development phases with their corresponding quality management practices. We call them Start-n-stop development (correspondingly with Start-n-stop quality management), Reactive development, Systematic development and Proactive development. It is the Proactive development phase, in which the organization can invest the majority of its capacity on feature-delivering work and at the same time proactively manage technical debt. We studied the use of metrics and their evolution over time in the organization, which provided insights on how the best metrics were chosen for practical use. We also identified how metrics can be applied systematically and distributed across the organization, among smaller substituent development branches, which helped the organization to transform their quality management practices toward Proactive quality management.

4 Applied Metrics

Over time, Ericsson has used variety of metrics in different manners and for different purposes. With the development maturity the use of metrics and their role has evolved considerably in the organization. In the coming subsections we present the technical debt metrics and their use in the four distinguished maturity phases.

4.1 Using Metrics in Start-n-stop Quality Management

As in every large software development organization, Ericsson also measured and still measures such aspects as software defects, test coverage, size, velocity etc. Such elementary measurements were used long time in the company and are vital for decision making. In the first maturity level (Start-n-Stop) the organization only focuses on visible aspect of quality management. Namely, the organization can only see the software defects as a manifestation of software quality. Therefore, developers mainly focus on handling defects to improve the quality. In this maturity phase the organization had either very scarce use of technical debt metrics or did not have any use at all. As the product grows, the size and complexity of the product escalates, and therefore technical debt accumulates. This situation prompts developers to start irregular measurements (usually taken place right before the product release) and localized refactoring activities. This is the beginning of transition to Reactive quality management.

4.2 Using Metrics in Reactive Quality Management

As the development gains more maturity, among all aforementioned metrics the organization also uses metrics which provide insights on invisible aspects of quality (technical debt). All these metrics are summarized in Table 1.

Table 2 Early technical debt metrics used at Ericsson

Metric	Used to measure Tech. Debt of
Number of added LOC	Code
Number of deleted LOC	Code
Number of modified LOC	Code, tests
Number of developers	Code
Fan-in	Code, architecture
Fan-out	Code, architecture
Cyclomatic complexity	Code
Nesting	Code
Halstead measures	Code

The application of these metrics usually corresponds to Reactive quality management, when the organization needs to have an insight about the quality of their product just before the delivery (and therefore before getting defect reports of customers). The measures are usually applied in ad-hoc manner and by isolated individuals or

teams. The measures are not used by combination and with determined thresholds. Rather raw numbers of them is presented and the rest of the verdict is left to the observing developers. In such situations, usually small areas of the product turn out having obvious technical debt, and the organization makes decision on either taking the risk or refactoring the identified areas.

4.3 Using Metrics in Systematic Quality Management

In the phase of systematic quality management the emphasis is put not only on what metrics are used but on how they are used. In this phase a combination of several simple metrics are used to achieve a single more insightful indicator. Powerful visualization techniques are used for visualizing big data for organization. Most importantly, in order to monitor the quality of the product systematically and on all organizational levels, measurement systems are developed which can provide information on weekly or daily basis [11]. The value of the metrics is enhanced when they are visualized in appealing and simple diagrams.

Figure 2 and Figure 3 show three key metrics for systematic assessment of technical debt at Ericsson. The metrics are developed in an integrated dashboard which serves the whole development organization. Together they form a solid metrics system, which then can guide the software development to continuously build in quality from the start. More importantly, the solid metrics system increase awareness so that the organization immediately acts on problems to always secure high quality and a continuously improved software development base.

The source code stability metric (heat maps) are used to visualize the change frequency of code [12] (see Figure 2 left hand diagram). This is a comprehensive way of showing thousands of software changes in the code in a single figure. The dark areas draw the attention to and trigger discussions about the effectiveness of testing of these areas and also other negative development practices.

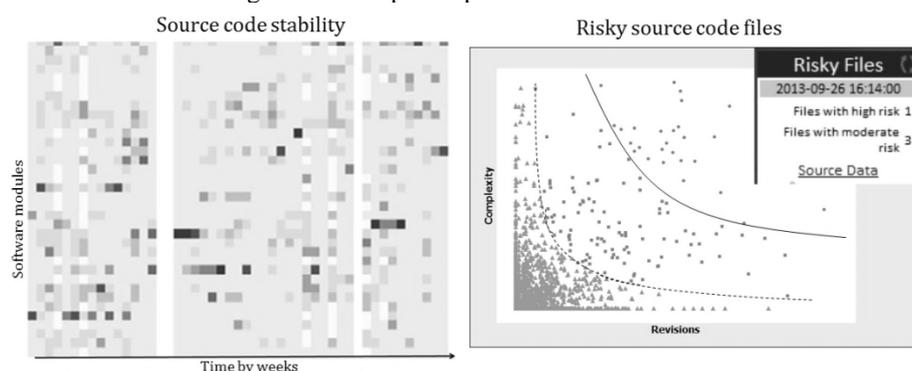


Figure 2 Key metrics and their visualization: Code stability heat map and dashboard of risky code

Studying the repetitive patterns in the heat maps allows predicting the changes, informing designers about potential need to update components and directing testing to reduce the risk of undiscovered defects.

Visualizing risky source code files [13] (see Figure 2 right hand diagram) permits developers to identify the areas of source code that are defect prone or difficult to maintain. Two metrics are combined for this assessment: cyclomatic complexity and number of revisions per file. Two thresholds are distinguished for this measure in order to separate files with high risk (upper right cloud of dots) and files with moderate risk (dots in between the two thresholds). In the upper right corner of the diagram the information product is presented, which shows the number of files with high risk and the number of files with moderate risk. The information product is integrated in the metrics dashboard of the organization alongside with other important metrics (not necessarily metrics concerned with technical debt). Files with high risk are refactored or additionally tested by all means. Depending on the organization's resources and risk appetite they can decide whether or not moderate risk should be mitigated or not.

Visualizing implicit dependencies [14] (see Figure 3) brings the invisible to become visible and thus draws attention to actions which shrinks the area of technical debt. When using historical data to predict events, the organization can reduce the negative repetitive patterns with help of for instance different preparation techniques.

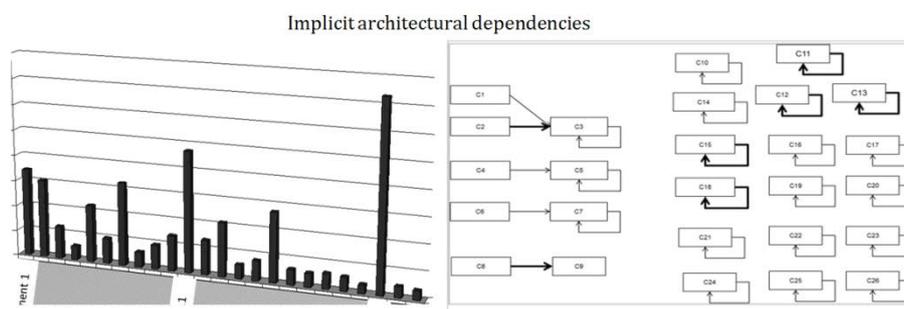


Figure 3 Key metrics and their visualization: Implicit architectural dependencies

The left side of the figure is the graphical representation of change waves, which allow detecting implicitly interconnected software modules (files in case of Ericsson). This is done by detecting how frequent changes in one file trigger frequent changes in another file after certain development time. In that diagram we can detect such patterns by observing different change-frequency picks. Based on this information the right hand diagram is developed which shows the names of modules and their implicit dependencies. By reviewing these dependencies the design architects make decisions on reengineering and avoiding unwanted dependencies.

4.4 Towards Using Metrics in Proactive Quality Management

Ericsson currently makes transition towards using metrics for proactive quality management. At the time of writing this paper the organization was adopting a princi-

ple of using a standard set of metrics for independent development teams and their development branches. The idea is that every team should have the possibility of interactively using the three metrics when developing the code. This permits to get the earliest possible feedback on their code quality and immediately take action if necessary. We also observed that with this kind of quality management a challenge emerges concerned with knowledge of developers on interpreting the metrics values. In order to have quality control of this level, the developers need to know what exactly the metrics' values show, and how they can use them on their own for improving the quality.

5 Action Principles for Staying in The Phase of Proactive Quality Management

Ericsson has longitudinal experiences of successful software improvement activities [15]. Their efforts to understand and manage technical debt come therefore rather natural. However, the organizational will to produce customer-value is also natural and understandable as adding no “customer-value” is equal to “no business”. When this will expands as a result of grasping over too many appetizing business opportunities, the outcome is a decreased feature-delivering capacity. The will needs to be accompanied by deep understanding of software development and its invisible technical debt. The experiences at Ericsson suggest the following action principles for moving towards and staying current in the optimal phase of Proactive quality management:

- **Embrace Technical Debt Existence.** Realize that all software development comes with a price for managing technical debt. Make use of metaphors and visualizations of metrics to create organizational acceptance of the term Technical debt. “Lending money to zero % interest... - the first quarter” is difficult to misinterpret. Seeing how the feature delivering capacity increases is the most inspiring to ensure action.
- **Understand Technical Debt.** Understand the deeply underlying factors for technical debt appearance. Continuously analyze the product and visualize metrics in all organizational levels in the earliest development phases. This reveals details about how neglecting design problems can turn into stinker-code over time. Understanding such cases allows the organization to plan and prioritize accordingly to avoid them and by that reduce their negative effect.
- **Start with the obvious.** Start with the small and most crucial set of problems. Usually the most of the problems in the product are concentrated in the few of artifacts. Use already identified and managed portions for understanding knowledge accumulating for the next step of action.
- **Learn and improve based on maturity phase.** Understand the maturity phase that is unique for your organization and by that implement and make use of the metrics most beneficial in that context.
- **Gradually increase towards more product-oriented metrics.** When the obvious metrics are in place, start with in-depth product oriented metrics like

complexity or implicit architectural dependencies in our case. These metrics prepare the organization to uncover the hidden technical debt in the product.

- **Provide Solid Metrics Systems.** Triangulate metrics to understand high complexity products in-depth. It is important to provide solid and visual metrics systems from which root-cause-analyses can be done. Appealing and easily accessible metrics facilitate the root-causes activities further. Combining the internal properties' metrics together allows keeping track of the influence of the technical debt on product performance.
- **Experiment with New Metrics.** Experiment with new metrics to find the most suitable and applicable ones. Experiences show that there are good practical metrics to re-use, but every organizations act in its own unique context, where different practical metrics can make tangible difference.
- **Do not stop paying attention.** Once in the want-to-be Proactive development phase, it is not equal to staying current there. Continuous attention to always act on trend changes is of highest importance. When allowing trend slippage, the organization allows the technical debt to grow and capacity for feature-delivering activities to shrink.

The notion of technical debt allows organizations to reason about the need to increase quality and organizing the road to understand and manage the technical debt. The four maturity phases provide a roadmap and improvement opportunity for large companies to manage the technical debt in practice. By using a solid metrics system, organizations can continuously monitor and act on negative trend deviations. When doing so, they can get the most out of the important feature-delivering activities – value delivery to customers and innovation!

6 The Journey Towards Proactive Quality Management

Along the evolution of organizational awareness of the invisible aspects of development (see Figure 1), the proportions of effort spent on each of these aspects change (see Figure 4). The effort spent on these aspects does not change by default along with increased awareness. It is the awareness itself that increases the will as well as a sense of urgency to act in ways where much attention is directed to manage the invisible aspects. Figure 4 visualizes the roughly estimated proportion of development effort spent on each of the four elements and through four maturity phases. Moving forward, we describe each maturity phase and typical practical metrics in more detail.

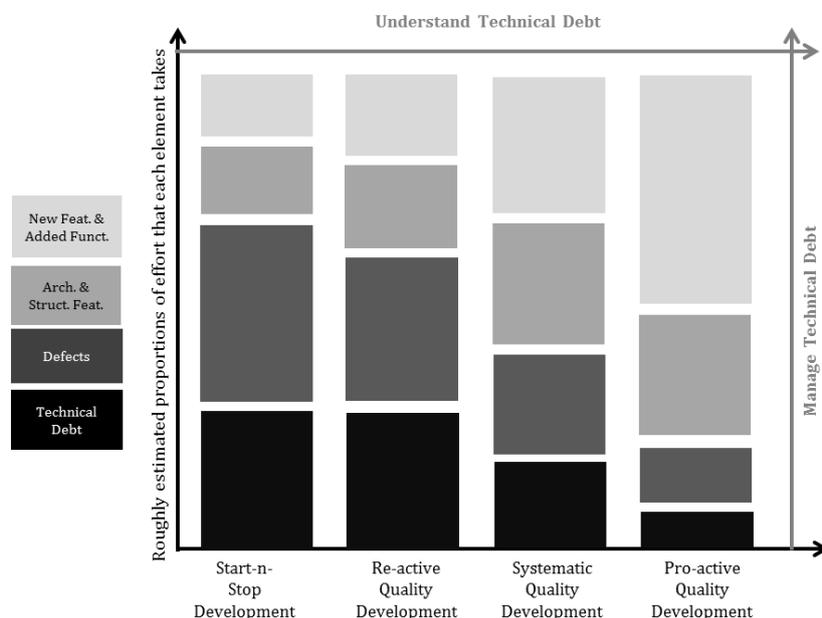


Figure 4. Estimated proportion of development effort per element in each maturity level

6.1 Start-n-Stop Quality Development

In the Start-n-stop development phase the organization typically focuses on feature development (i.e. the visible positive value) and then reacts on defects when the quality situation becomes critical. The technical debt is not managed at all and remains “hidden” until the development of the product decelerates to the extent when the organization needs to react immediately. This deceleration is usually observed by a high number of in-development defects which need to be fixed before progressing with the development. The organization “stops” new functionality development and focused on reduction of defects only [16].

The main metric focus of the organization in the start-n-stop development phase is the number of known defects measured from various perspectives (e.g. per severity, release, function, etc), which are perceived as the major problem. Defect volume metrics guide the development and the key mechanism used is to stop development when high defect levels prevent continued development. During such a stop, a ‘clean-up’ period is initiated before the development can start again. In this phase the technical debt is hidden and only the symptoms, defects of it, are managed. To continue the journey, the organization needs to develop fundamental awareness of its inefficient development practice.

6.2 Reactive Quality Development

Once the organization recognizes the need to make in-depth improvements it usually seeks methods to reduce the amplitude between the high and low number of defects. Using complexity analyses is one beneficial method as it is perceived to provide the organization with understanding of the underlying problems that cause the defects (i.e. allowing the organization to start managing the invisible values) [17]. In this phase, the organization starts to reactively act on its situation and typically improves its architecture management. Common efforts are focused on refactoring of stinker-code and redesign of interfaces. These improvements result in lower number of defects and thus more capacity for feature development. In this phase, the organization has a limited understanding of the technical debt although there is a shared perception that defects are only symptoms.

A few senior developers are constantly prioritized to do some localized discrete measurements and improvements here and there in the product to keep the development ongoing. Metrics in this phase draw the attention of the organization to quality assurance and product properties rather than quality problems. Typically used metrics are test effectiveness, test-requirement coverage, basic complexity measurements and product availability. We also discovered that software designers make ad-hoc attempts to visualize various aspects of product stability. Visualizing instability in this context is a means of searching for the areas where technical debt can be found.

In this phase the technical debt is recognized and attempts are made to understand it. To continue the journey, the organization needs to develop in-depth understanding of its actual problems that are explaining their current development situation.

6.3 Systematic Quality Development

In this phase the organization understands the need for managing the technical debt and has established measurement systems and visualization tools. The organization attempts to have a standard shared metrics dashboard or similar tools so all development teams, developers and architects can follow the evolving product condition. The main focus is to find new means for visualizing the invisible aspects of the product to be able to manage them. The organization spends effort on defining new metrics for measuring the invisible problems in order to remedy them before they even occur. We observed this type of behavior by studying architecture quality [14].

The metrics for quantifying the invisible values are organized in a solid metrics system. This metrics system guides the development to continuously act on potential problems which can become defects (e.g. monitoring implicit architectural dependencies and combination of several complexity metrics). Stability of the source code is also monitored using code change rate metrics to identify periods of development when too intensive development prevent quality assurance from being effective. An outspoken strategy to build quality from the start is growing and selected individuals lead by example.

In this phase the complete technical debt is understood and valuable efforts are made to manage it. Practical and solid metrics systems are used to guide the organiza-

tion. To continue the journey, the organization needs to develop skills to continuously and immediately act on the metrics describing their development situation.

6.4 Proactive Quality Development

In the Proactive quality management phase the organization possesses data and experience on which aspects of technical debt can be continuously monitored. The organization has an established metrics system to both understand and manage the technical debt and can focus the main part of their development capacity on feature development. The solid metrics system monitors deviations from the decreasing trend in technical debt, thus allowing the organization to come in control of their debt at all times.

A solid standardized (in the scope of organization) measurement system continuously provides information to the architects, managers, and technical leaders. Besides this every development team and individual developer has the standardized and approved measurement tool on his own computer in order to interactively follow his and teammates' code quality and control it in its development earliest phase. The designers of measurement systems and metrics (researchers and responsible engineers from the organization) set up systematic presentations and training session, so the developers can understand the meaning and interpret the metrics.

In this phase the technical debt is both understood and managed. Focus can be fully directed to feature growth and metrics are used to automatically keep the technical debt under control. To stay current in this phase, the organization needs to continuously pay attention to metrics showing negative trend changes and immediately address this with appropriate actions.

7 Related Work

An interesting discussion on technical debt is provided by Buschmann [18] who discuss the trade-off between paying or not paying accumulated technical debt. He claims that technical debt is similar to financial debt as it accumulates like a compound interest, but it is not always paid back if the organization decides to obsolete their old product and start with a completely new one. Lim, et al. [19] observes that measuring technical debt is a difficult task as it can have a variety of manifestations. Possibly that is the reason that many organizations including Ericsson strive for establishing the right metrics system for technical debt measurement. Tom, et al. [20] explore the causes of technical debt and found that it is mainly the decisions of non-technical people that prompt accumulating technical debt. Martini, et al. [21] studies the technical debt issues in large software development companies and concludes that the lack of knowledge is not a primary cause for accumulated technical debt. However, he founds that schedule pressure for feature delivery, using legacy systems, and small time allocated for refactoring are the main causes. In a later study Martini, et al. [22] develop a qualitative model for understanding technical debt in large software development projects. This study is also conducted in the same organization of Ericsson.

son as our study. Our study can be considered a complement to their study as we emphasize the use of measurement system for technical debt detection. Using such measurement system with their qualitative product specific models can be a powerful tool for technical debt management.

There have been a few studies proposing approaches for identifying one or another manifestation of technical debt: Marinescu [23] propose an approach for finding the technical debt of design flows based on eight types of flows found in code. Then he investigates how different software attributes, such as complexity and cohesion, influence each of the defined design flaw. The measurement systems proposed by us for technical debt management can be used to find several design flaws proposed by Marinescu. Nugroho, et al. [24] introduces a “return on investment” approach for managing technical debt. They estimate the maintainability of system by measuring several attributes of code and categorize the units of code by three different levels of risk. Then they calculate (to unclear precision) the return on investment for code maintainability in case the risky source code is refactored and improved to a level of optimal design. The study is interesting as it explicitly attempts to develop a connection between internal quality and business decisions in software development. Guo, et al. [25] explores the effect of technical debt in practice and observes that it has significant negative influence on the studied developed project. They conclude that business factors should be incorporated in the technical debt management model so the trade-offs between business opportunities and software quality can be considered. Probably one of the best known approaches to manage technical debt is proposed by Letouzey [26]. The approach is based on complexity measurement, dependencies, and coding violations. There is also a tool support for this approach (SonarQube tool). The tool offers effective visualizations, which can come to handy for large software products. It also attempts to derive the effort required for paying the technical debt, but this is not yet rigorously tested. The SonarQube has large amount of predefined rules for detecting coding violations and can categorize the severity of violations. However the tool relies on rather simplistic measures which are shown in literature not to be so good indicators of internal quality. We believe that if the tool could rely on more sophisticated measures (combination of measures), the assessment accuracy of SonarQube would increase significantly. We show such an attempt of using sophisticated measures at Ericsson, however such measures should be further evaluated rigorously which is the future work of this research.

Tom, et al. [20] investigates the main areas of technical debt and propose a framework for its management. This work is particularly valuable due to its extensive elaboration on different kinds of technical debts.

8 Conclusions

The primary aim of software development companies is to deliver value to their customer and be innovative. However they do not spend all of their resources for value delivering activities directly. In fact much time is spent on such activities as defect handling, re-architecting, and refactoring. A relevant metaphor to describe this

situation is the “technical debt”, which can be considered the time or effort that the organization should pay in order to improve the degrading quality of the product. This paper distinguishes four maturity levels of managing technical debt at Ericsson: Start-n-stop, Reactive, Systematic, and Proactive management. We observed that Ericsson has been using metrics differently through the four maturity phases. We investigated and presented the metrics and their evolution over four maturity phases of development. The investigations showed that there are three key metrics which are used for Systematic technical debt management at Ericsson: 1) change frequencies of files visualized by heat maps 2) implicit architectural dependencies and 3) risky source files visualized and reported daily on a dashboard. We also observed that in order for the organization to move towards Proactive management of quality and technical debt in particular, every software development team and individual developer is preferred to have a standard and organizationally accepted measurement dashboard on her own computer, so she can interactively enhance the quality of the product all the time.

Acknowledgement

The research has been conducted in Software Center, Chalmers | University of Gothenburg. <http://www.software-center.se/>

The researchers thank the manager of Research and Development organization at Ericsson, as well as all design architects and developers who supported the research.

References

1. W. Cunningham, "The WyCash portfolio management system," *ACM SIGPLAN OOPS Messenger*, vol. 4, pp. 29-30, 1993.
2. P. M. Duvall, S. Matyas, and A. Glover, *Continuous integration: improving software quality and reducing risk*: Pearson Education, 2007.
3. J. Bosch, *Continuous Software Engineering*: Springer, 2014.
4. Y. Guo and C. Seaman, "A portfolio approach to technical debt management," in *Proceedings of the 2nd Workshop on Managing Technical Debt*, 2011, pp. 31-34.
5. N. Brown, Y. Cai, Y. Guo, R. Kazman, M. Kim, P. Kruchten, *et al.*, "Managing technical debt in software-reliant systems," in *Proceedings of the FSE/SDP workshop on Future of software engineering research*, 2010, pp. 47-52.
6. P. Kruchten, R. L. Nord, and I. Ozkaya, "Technical debt: from metaphor to theory and practice," *IEEE Software*, pp. 18-21, 2012.
7. V. Antinyan, M. Staron, J. Hansson, W. Meding, P. Osterström, and A. Henriksson, "Monitoring Evolution of Code Complexity and Magnitude of Changes," *Acta Cybernetica*, vol. 21, pp. 367-382, 2014.
8. N. Ohlsson, M. Helander, and C. Wohlin, "Quality improvement by identification of fault-prone modules using software design metrics," in *Proceedings: International Conference on Software Quality*, 1996, pp. 1-13.

9. K. Pandazo, A. Shollo, M. Staron, and W. Meding, "Presenting software metrics indicators: a case study," in *Proceedings of MENSURA 20th International Conference on Software Product and Process Measurement*, vol. 20, no. 1, 2010.
10. A. B. Sandberg, L. Pareto, and T. Arts, "Agile collaborative research: Action principles for industry-academia collaboration," *Software, IEEE*, vol. 28, pp. 74-83, 2011.
11. I. I. 15939:2001, "Information technology — Software engineering — Software measurement process," 2001.
12. M. Staron, J. Hansson, R. Feldt, W. Meding, A. Henriksson, S. Nilsson, *et al.*, "Measuring and Visualizing Code Stability--A Case Study at Three Companies," in *Software Measurement and the 2013 Eighth International Conference on Software Process and Product Measurement (IWSM-MENSURA), 2013 Joint Conference of the 23rd International Workshop on*, 2013, pp. 191-200.
13. V. Antinyan, M. Staron, W. Meding, P. Osterstrom, E. Wikstrom, J. Wrangler, *et al.*, "Identifying risky areas of software code in Agile/Lean software development: An industrial experience report," in *Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), 2014 Software Evolution Week-IEEE Conference on*, 2014, pp. 154-163.
14. M. Staron, W. Meding, C. Hoglund, P.-E. Eriksson, J. Nilsson, and J. Hansson, "Identifying Implicit Architectural Dependencies Using Measures of Source Code Change Waves," in *Software Engineering and Advanced Applications (SEAA), 2013 39th EUROMICRO Conference on*, 2013, pp. 325-332.
15. A. Börjesson and L. Mathiassen, "Successful process implementation," *Software, IEEE*, vol. 21, pp. 36-44, 2004.
16. M. Staron, W. Meding, and B. Söderqvist, "A method for forecasting defect backlog in large streamline software development projects and its industrial evaluation," *Information and Software Technology*, vol. 52, pp. 1069-1079, 2010.
17. B. Boehm and V. R. Basili, "Software defect reduction top 10 list," *Foundations of empirical software engineering: the legacy of Victor R. Basili*, vol. 426, 2005.
18. F. Buschmann, "To pay or not to pay technical debt," *Software, IEEE*, vol. 28, pp. 29-31, 2011.
19. E. Lim, N. Taksande, and C. Seaman, "A balancing act: what software practitioners have to say about technical debt," *Software, IEEE*, vol. 29, pp. 22-27, 2012.
20. E. Tom, A. Aurum, and R. Vidgen, "An exploration of technical debt," *Journal of Systems and Software*, vol. 86, pp. 1498-1516, 2013.
21. A. Martini, J. Bosch, and M. Chaudron, "Investigating Architectural Technical Debt accumulation and refactoring over time: A multiple-case study," *Information and Software Technology*, 2015.
22. A. Martini, J. Bosch, and M. Chaudron, "Architecture technical debt: Understanding causes and a qualitative model," in *Software Engineering and Advanced Applications (SEAA), 2014 40th EUROMICRO Conference on*, 2014, pp. 85-92.
23. R. Marinescu, "Assessing technical debt by identifying design flaws in software systems," *IBM Journal of Research and Development*, vol. 56, pp. 9: 1-9: 13, 2012.
24. A. Nugroho, J. Visser, and T. Kuipers, "An empirical model of technical debt and interest," in *Proceedings of the 2nd Workshop on Managing Technical Debt*, 2011, pp. 1-8.

25. Y. Guo, C. Seaman, R. Gomes, A. Cavalcanti, G. Tonin, F. Q. Da Silva, *et al.*, "Tracking technical debt—An exploratory case study," in *Software Maintenance (ICSM), 2011 27th IEEE International Conference on*, 2011, pp. 528-531.
26. J.-L. Letouzey, "The SQALE method for evaluating technical debt," in *Proceedings of the Third International Workshop on Managing Technical Debt*, 2012, pp. 31-36.

Defining Metrics for Continuous Delivery and Deployment Pipeline

Timo Lehtonen¹, Sampo Suonsyrjä², Terhi Kilamo², and Tommi Mikkonen²

¹Solita Ltd, Tampere, Finland

`timo.lehtonen@solita.fi`

²Tampere University of Technology, Tampere, Finland

`sampo.suonsyrja@tut.fi`, `terhi.kilamo@tut.fi`, `tommi.mikkonen@tut.fi`

Abstract. Continuous delivery is a software development practice where new features are made available to end users as soon as they have been implemented and tested. In such a setting, a key technical piece of infrastructure is the development pipeline that consists of various tools and databases, where features flow from development to deployment and then further to use. Metrics, unlike those conventionally used in software development, are needed to help define the performance of the development pipeline. In this paper, we address metrics that are suited for supporting continuous delivery and deployment through a descriptive and exploratory single case study on a project of a mid-sized Finnish software company, Solita Plc. As concrete data, we use data from project "Lupapiste", a web site for managing municipal authorizations and permissions.

Keywords: Agile measurements, continuous deployment, lean software development.

1 Introduction

Software development, as we know it today, is a demanding area of business with its fast-changing customer requirements, pressures of an ever shorter time-to-market, and unpredictability of market [1]. Lean principles, such as "Decide as late as possible", have been seen as an attractive way to answer to these demands by academics [2]. With the shift towards modern continuous deployment pipelines, releasing new software versions early and often has become a concrete option also for an ever growing number of practitioners.

As companies, such as Facebook, Atlassian, IBM, Adobe, Tesla, and Microsoft, are going towards continuous deployment [1], we should also find ways to measure its performance. The importance of measuring the flow in lean software development was identified already in 2010 by [3], but with the emergence of continuous deployment pipelines, the actual implementation of the Lean principles has already changed dramatically [4]. Further on, measuring has been a critical part of Lean manufacturing long before it was applied to software development [5]. However, the digital nature of software development's approach

to Lean (ie. continuous deployment pipelines) is creating an environment, where every step of the process can be traced and thus measured in a way that was not possible before. Therefore, the need for a contemporary analysis of what should be tracked in a continuous deployment pipeline is obvious to us.

In this paper, we address metrics that are suited for supporting continuous delivery and deployment through a descriptive and exploratory single case study on a project of a mid-sized Finnish software company, Solita Plc (<http://www.solita.fi>). As case studies investigate the contemporary phenomena in their authentic context, where the boundaries between the studied phenomenon and its context are not clearly separable [6], we use concrete data from project "Lupapiste", or "Permission desk" (<https://www.lupapiste.fi>), a web site for managing municipal authorizations and permissions. The precise research questions we address are the following:

- RQ1:** Which relevant data for practical metrics are automatically created when using a state-of-the-art deployment pipeline?
- RQ2:** How should the pipeline or associated process be modified to support the metrics that escape the data that is presently available?
- RQ3:** What kind of new metrics based on automatically generated data could produce valuable information to the development team?

The study is based on quantitative data and descriptions of the development processes and the pipeline collected from the developer team. Empirical data of the case project was collected from information systems used in the project, including a distributed version control system (Mercurial VCS) and a monitoring system (Splunk).

The rest of this paper is structured as follows. In Section 2, we address the background of this research. In Section 3, we introduce our case project based on which the research has been conducted. In Section 4, we propose metrics for continuous delivery and deployment pipeline. In Section 5, we discuss the results of case study and provide an extended discussion regarding our observations. In Section 6 we draw some final conclusions.

2 Background and Related Work

Agile methods – such as Scrum, Kanban and XP to name a few examples – have become increasingly popular approaches to software development. With Agile, the traditional ways of measuring software development related issues can be vague. The outcome of traditional measures may become dubious to the extent of becoming irrelevant. Consequently, one of the main principles of Agile Software Development is "working software over measuring the progress" [7].

However, not all measuring can be automatically considered unnecessary. Measuring is definitely an effective tool for example for improving Agile Software Development processes [8], which in turn will eventually lead to better software. A principle of Lean is to cut down waste that processes produce as well as parts of the processes that do not provide added value [9]. To this end, one should first

recognize the current state of a process [3]. This can be assisted with metrics and visualizations, for instance. Therefore, one role for the deployment pipeline is to act as a manifestation of the software development process and to allow the utilization of suitable metrics for the entire flow from writing code to customers using the eventual implementation [10].

Overall, the goal of software metrics is to identify and measure the essential parameters that affect software development [11]. Mishra and Omorodion [11] have listed several reasons for using metrics in software development. These include making business decision, determining success, changing the behavior of teammates, increasing satisfaction, and improving decision making process.

2.1 Continuous Delivery and Deployment

Continuous delivery is a software development practise that supports the lean principles of "deliver as fast as possible" and "empower the team". In it the software is kept deployable to the staging and production environments at any time [12, 13]. Continuous delivery is preceded by continuous integration [14, 15] where the development team integrates its work frequently on a daily basis. This leads to a faster feedback cycle and to benefits such as increased productivity and improved communication [15–17]. Similarly, "the final ladder" — continuous deployment — requires continuous delivery. So, continuous deployment [18, 19] takes one step further from delivery. In it software is automatically deployed as it gets done and tested. Taking continuous deployment to the extreme would mean deployment of new features directly to the end users several times a day [20, 21]. Whether software is deployed all the way to production, or to a staging environment is somewhat matter of opinion [18, 22] but a reasonable way to differentiate between delivery and deployment in continuous software development is the release of software to end users. Delivery maintains a continuously deployable software, deployment makes the new software available in the production environment.

Regardless of actual deployment, continuous software development requires a deployment pipeline (Figure 1) [10], which uses an automated set of tools from code to delivery. The role of these tools is to make sure each stakeholder gets a timely access to the things they need. In addition, the pipeline provides a feedback loop to each of the stakeholders from all stages of the delivery process. An automated system is not about software going into production without any operator supervision. The point of the automated pipeline is that as the software progresses through it, different stages can be triggered for example by operations and test teams by the click of a button.

2.2 Agile Metrics

In [8] the authors categorize agile metrics used in industry into metrics relating to iteration planning and tracking, motivation and improvement, identifying process problems, pre-release and post-release quality, and changes in the processes or tools. The metrics for iteration planning offered help with prioritization

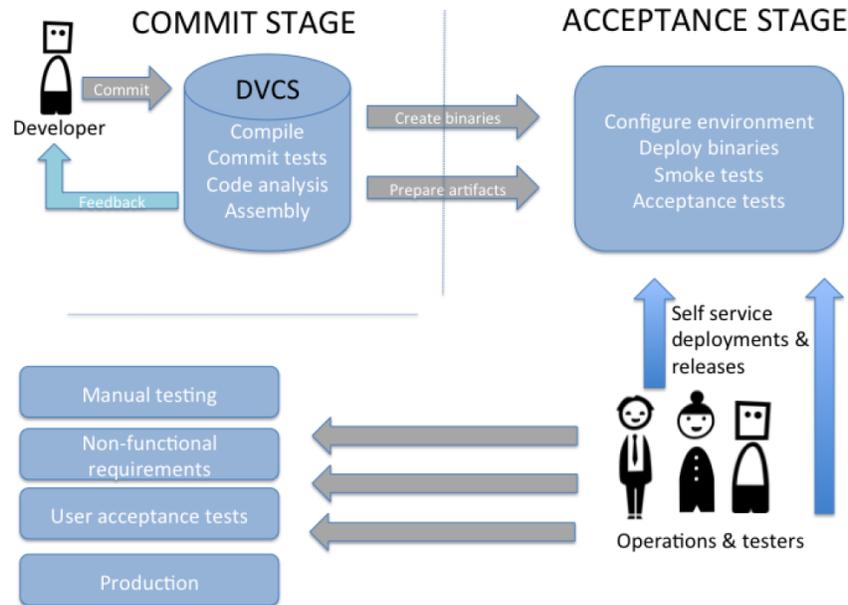


Fig. 1. Anatomy of a Deployment Pipeline according to [10].

of features. These include estimation metrics for measuring the size of features, the revenue a customer is willing to pay for a feature, and velocity of the team in completing a feature development. Iteration tracking include progress metrics such as the number of completed web pages, story completion percentage, and again velocity metrics. In the category of motivation and improvement, approaches such as visualizing the build status and showing the number of defects in monitors were found to lead into faster build and fix times. Using metrics such as lead time and story implementation flow assist in identifying waste and in describing how efficiently a story is completed compared to the estimate. Pre-release quality metrics were found to be used for making sure the product is tested sufficiently and for avoiding integration fails. Post-release quality metrics measure attributes such as customer satisfaction and customer responsiveness. These can be evaluated for example with the number of defects sent by customers, change requests from customers, and customer's willingness to recommend the product to other potential customers. For the final category of metrics for changes in processes or tools, sprint readiness and story flow metrics were found to change company policies to having target values for metrics.

In [11] a more general approach in categorization of agile metrics is used. The authors define the core agile metrics to include product, resource, process, and project metrics. Of these, the product metrics deal with size, architecture,

structure, quality, and complexity metrics. Resource metrics are concerned with personnel, software, hardware, and performance metrics. Process metrics deal with maturity, management, and life cycle metrics, and project metrics with earned business value, cost, time, quality, risk, and so on. Each of these sub-metrics can define a range of additional metrics such as velocity, running tested feature, story points, scope creep, function points, earned business value, return on investment, effort estimates, and downtime. The researchers also point out that teams should invent metrics as they need such, and not use a metric simply because it is commonly used – this might result in data that has no value in the development.

Kunz et al. [23] claim that especially source-code based product metrics increase quality and productivity in agile software development. As examples, the researchers present Number of Name-Parts of a method (NNP), Number of Characters (NC), Number of Comment-Lines (CL), Number of Local Variables (NLV), Number of Created Objects (NCO), and Number of Referring Objects (NRO). All in all, the researchers emphasize the early observation of quality to keep the software stable through the development process.

In their 2009 book [9] the Poppendiecks emphasize the customer-centricity in metrics. They present examples of these including time-to-market for product development, end-to-end response time for customer requests, success of a product in the marketplace, business benefits attributable to a new system, customer time-to-market, and impact of escaped defects.

2.3 Lean Metrics

As lean methods have been developed originally for manufacturing, there are obviously collections of corresponding metrics. For instance, the following has been proposed [3]: Day-by-the-Hour (DbtH) measures the quantity produced over the hours worked. This should correspond to the same rate of customer need. Capacity utilization (CU) is the amount of work in progress (WIP) over the capacity (C) of the process. An ideal rate is 1. On-time delivery (OTD) is presented as the number of late deliveries over the number of deliveries ordered. Moreover, such metrics or signals that help the involved people to see the whole, are mentioned in [24].

Petersen and Wohlin [3] present cost efficiency (CE), value efficiency (VE), and descriptive statistics as measurements for analyzing the flow in software development. A possible way of measuring CE is dividing lines of code (LOC) by person hours (PH). However, they point out how this cost perspective is insufficient as value is assumed to be created always by investment. The increase in LOC is not always value-added as knowledge workers are not machines. On the other hand, $VE = (V(\text{output}) - V(\text{input})) / \text{time window}$. $V(\text{output})$ represents the final product, and $V(\text{input})$ the investment to be made. This type of measuring takes value creation explicitly into account, and therefore it can be a more suitable option.

Overall, according to van Hilst and Fernandez [25] there are two different approaches to evaluating efficiency of a process considering Lean ideals. These

views apply models from queuing theory, in which steps of a process are seen as a series of queues. Work advances from queue to queue as it flows through the process, and process performance is then analyzed in terms of starvation (empty queues) and bottlenecks (queues with backlogs). The first approach is to look at a workstation and examine the flow of work building up or passing through. At the same time, the activities on the workstation are studied to see how they either add value or impede the flow. On the contrary, the second approach follows a unit of work as it passes through the whole process. In that case, the velocity of this unit is studied. Considering these two approaches, van Hilst and Fernandez [25] describe two metrics: Time-in-process and work-in-process. Work-in-process is corresponding with the first approach as it describes the amount of work present in an intermediate state at a given point in time. The second approach is measured with time-in-process describing the time needed for a unit of work to pass through the process. For an optimal flow, both of these need to be minimized.

Finally, Modig [24] takes an even deeper look into measuring flow efficiency. This metric focuses on the unit, which is produced by an organization, (*flow unit*) and its flow through different workstations. Flow efficiency describes how much a flow unit is processed in a specific time frame. Higher flow efficiency is often better from the flow units point of view. For instance, if a resource processes the flow unit for one hour, and then the flow unit is placed to a queue of two hours, and then another resource starts to process it for three hours, the flow efficiency is $4 / 6 = 66\%$. If the length of the queue is shortened to for example half an hour, the flow efficiency is higher ($4 / 4.5 = 89\%$).

3 Case Lupapiste

An industrial single case study was conducted to investigate measuring a state-of-the-art pipeline within a two months time frame of actual development work. The case project, and its deployment pipeline are introduced in the following.

3.1 Description of the Case Project

The application "Lupapiste", freely translated "Permission Desk", is a place for the citizens and companies in Finland to apply for permissions related to the built environment, available at <https://www.lupapiste.fi>. The project was started in 2012, and the supplier of the system is Solita Plc., a mid-sized Finnish ICT company. The end users of the system consist of various stakeholders, with various interests. The Environmental Ministry of Finland owns the project code and acts as a customer in some new functionalities needed to the system.

At the time of research (Fall 2015), the project team consisted of seven developers, a user experience (UX) designer and a project manager that are co-located in a single workspace at the supplier. On the management level there are four more persons in different roles. The team is cross-functional and has also DevOps [26] capabilities. Some team members have an ownership of certain parts of the

system, but the knowledge is actively transferred inside the team by changing the areas continuously and for example by applying agile practices like pair reviewing of code to spread out the knowledge in a continuous manner. The team takes use of a custom Lean Software Development process that includes features from agile Scrum-based processes with lean heritage. The process is ongoing and has no sprints, but milestone deadlines for certain functionalities are set by the product owner team, which consists of project management personnel of the supplier and the formal customer of the project. Furthermore, agile practices, like daily meetings, have been combined with lean practices and tools, like a Kanban board.

3.2 Deployment Pipeline of the Project

The pipeline of the case project has several environments (see. Figure 2) – a personal local development environment (Local), the shared development environment (Dev), a testing environment (Test), a quality assurance environment (QA) and the production environment (Production). Each of these environments serve different needs, and deployments to the different environments are managed through the version control system. Therefore, it automatically provides accurate data and meta data to measure the pipeline, which we have already proposed in an earlier paper [27]. The actual timestamps of deployments are stored in the meta data of the version control system branches.

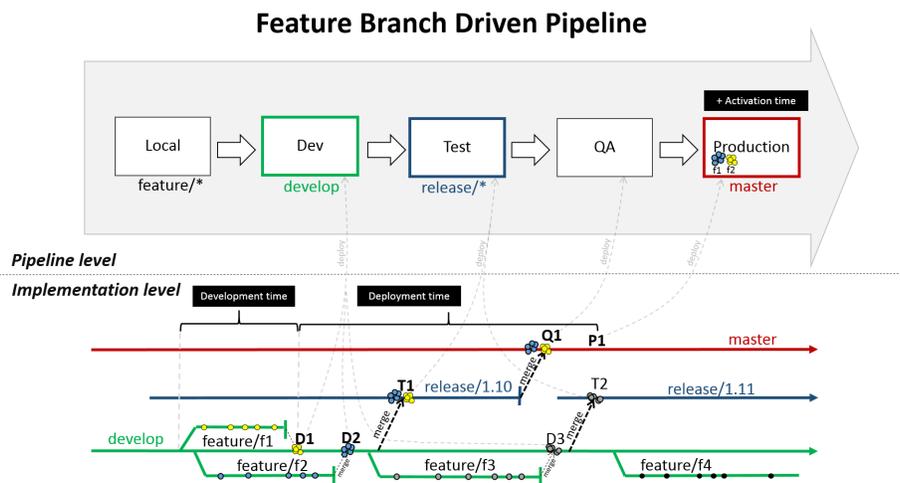


Fig. 2. Deployments to the pipeline environments are triggered by push-events to the distributed VCS. Features f1 and f2 have been merged and pushed to the develop-branch (triggering deployments D1 and D2 to the Dev-environment), then to the Test-environment (deployment T1 to Test-environment) and production environment (P1).

The team uses a VCS-driven solution to manage the deployments to the environments of the pipeline. The team applies the Driessen branching model [28], which utilizes feature branches. Figure 2 presents the connection between the branches and the deployments to the various environments of the pipeline. When the development of a new feature starts, a developer opens a new feature branch and starts developing the feature in the Local environment by committing changes to the new branch. The developer may push the local changes to the version control system from time to time, but no CI jobs are executed in this phase. When the development of the feature is ready, the feature branch is closed and the changes are merged to the develop-branch. When the changes are pushed to the version control system, a stream of CI jobs for deploying a new version to the Dev-environment is triggered automatically (deployments D1, D2 and D3 in Figure 2). The CI jobs build the software, migrate the database, and deploy and test the software.

The deployment to the Test environment is accomplished by merging the develop-branch to a release-branch. Once again, when the changes to a release branch are pushed to the version control system, a stream of CI jobs for building the software, migrating the database, and deploying and testing the software in the Test environment is triggered (deployments T1 and T2). For instance, deployment t1 in Figure 2 was triggered by a push to branch release/1.10, which contained features f1 and f2. Similarly, the production deployment happens by closing the release branch, which is then merged to the master-branch. The new version to be released can then be deployed to the QA (deployment Q1) and production environments (deployment P1) with a single click from the CI server.

In Figure 2, feature f1 flows from the development to production in deployments D1, T2 and P1. Feature f2 flows in deployments D2, T1 and P1. Feature f3 has flown to the test-environment in deployments D3 and T2. In order to deploy feature f3 to the production environment, release branch release/1.11 should be closed and merged to the master branch, which then would be manually released with a single click from the CI system.

Figure 3 presents the correspondence of the branches in the version control system and the CI jobs on the radiator screen in the team workspace. If a CI job fails, the team is immediately knowledgeable of the problems. Moreover, the current status of the functional end-to-end tests running in the Dev-environment is visible to the team.

In case of urgent problems in the production environment, the branching model also allows creation of a hotfix branch. Figure 3 represents a situation where urgent problems occurred after a deployment to the production environment. The automated tests had passed, but the login button was invisible on the front page because of layout problems. In this special case, a hotfix branch was then opened, the layout problems were fixed, the branch was merged to the master branch, and when the changes were pushed and a CI job was triggered manually, the problem was fixed and the users could continue logging in to the system.

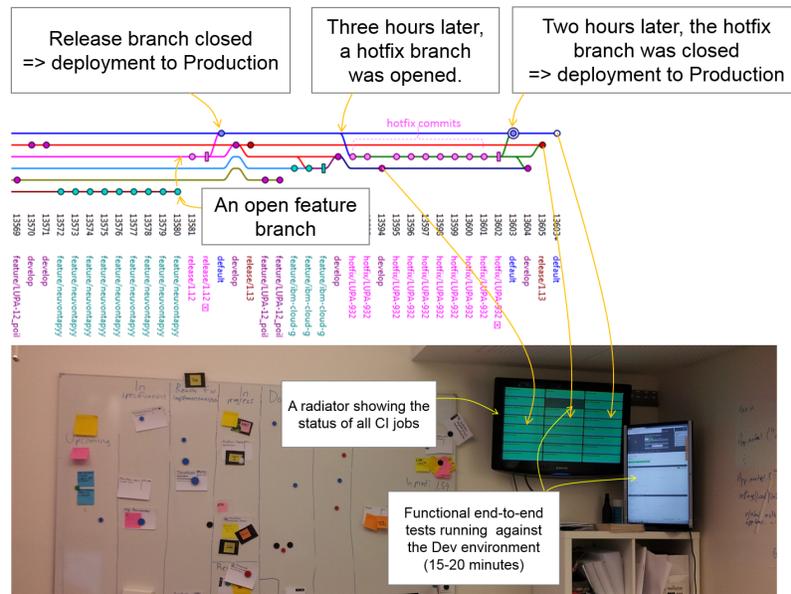


Fig. 3. An actual usage sample of the branching model and its correspondence to the CI jobs.

4 Defining Metrics for Pipeline

In this section, we define several new metrics which describe the properties of the deployment pipeline. The goal of the metrics is to provide valuable information for the team for improving the performance of the pipeline. With them, it is possible to detect bottlenecks, indicate and consequently eliminate, waste, and find process improvements.

We divided the metrics into two categories. First, *Metrics on the Implementation Level* dependent of the toolset and practices used to implement the pipeline. Second, *Metrics on the Pipeline Level* are metrics that are independent of the actual implementation of the pipeline. The metrics in the two categories are discussed in more detail in the following.

4.1 Metrics on the Implementation Level

The availability of data to calculate flow and throughput depends on the implementation of the pipeline and the actual tools and practices used. In essence, development, deployment and activation time must be available for each feature, discussed in more detail in the following.

- *Development time*, or the time it takes for the team to implement a new feature. The development time of a single feature can be measured in our case project, as each new feature is a new branch in the version management

system. The starting time for the new feature is simply the time when the branch is created, and completion time is when the branch is merged with the master branch. See Figure 2 for an example of development time of feature/f1. It is the time from opening the feature branch until D1. In an earlier paper [27] we measured the value of this metric during a three month period. The value was typically one or two days, but for larger features, it was even 12 working days.

- *Deployment time*, or the time it takes to deploy a new feature to production use when its implementation has been completed. There are two dimensions to this metric. One is the execution time of the tools needed for the actual deployment (e.g. seconds or minutes), and the other is the decision process to deploy the features, if additional product management activities, for example acceptance testing, are associated with the deployment (e.g. hours or days). See Figure 2 for an example of deployment time of feature/f1 – the time from D1 to P1. In [27], we measured a mean value of nine working days during a three month period.
- *Activation time*, or the time it takes before the first user activates a new feature after its deployment. Activation time can only be measured for features that are specific enough to be visible in production logs. At times, however, this can be somewhat complicated. For instance when a new layout is introduced, the first time the system is activated can be considered as the first use of the feature. See Figure 2 for an example of activation time of features found in the production log. It is the time from P1 to the first use caught from the production logs. The mean activation time in [27] was three working days while the median was one working day.

Another viewpoint to the time a feature spends in the deployment pipeline, is to count the age of the features that have been done, but are still waiting for production environment deployment. The following metric is based on measuring the current time spent on the pipeline:

- *Oldest done feature (ODF)*, or the time that a single feature has been in development done state, but is still waiting for deployment to the production environment in some of the environments of the deployment pipeline. The metric is dependent on Definition of Done (DoD) [29]. In our case project, this data is available from the meta data of the feature branches: a feature branch closed, but not merged to a closed release branch. At the time of research (Autumn 2015), the value of ODF in the case project is currently six days and the weekly release schedule has kept the value in less than one week constantly.

4.2 Metrics on the Pipeline Level

In the context of continuous delivery and deployment, the throughput of the pipeline used to deliver features to the end user is an important metric. Out of

the existing metrics, flow efficiency, proposed in [24], best captures the spirit of the pipeline. We propose the following new metrics to this category.

- *Features Per Month (FPM)*, or the number of new features that have flown through the pipeline during a month. The metric is based on Day-by-the-Hour (DbtH), which measures quantity produced over hours worked [3]. In the case project, the data for this metric can be collected from the implementation level data (number of feature branches closed and merged to a release branch that has been closed). Apparently, this metric can be measured in many other implementation settings, for example in a project that does not use feature branches. For example, issue management system data or version commit messages following a certain convention, are possible sources for this data. At the time of research, the value of this metric for the case project is 27 FPM during the last three months, which is more than one feature per working day.
- *Releases Per Month (RPM)*, or the number of releases during one month. Long term change of this metric provides information on changes in the release cycle. In the case project, this data is available both in the version control system and the CI server logs. At the time of research, the value of this metric for the case project is 7 RPM, which is one or two releases per week.
- *Fastest Possible Feature Lead Time*, or the actual time the feature spends in the build and test phase on the pipeline. In our case project, there is latency which originates from the use of feature branches and separate build processes for each branch. The code is compiled and packaged multiple times during the different phases of the pipeline. A build promotion approach in e.g. [30], where the code is build only once and the same binary is deployed to all environments, the lead time may be shorter. At the time of research, the value for this metric is two hours (quick integration and unit tests running some minutes in the commit stage and functional end-to-end browser tests running one hour in the pipeline environments). As a shortcut for urgent issues, the team can also use a hotfix branch, which allows making a quick fix in minutes.

5 Results

Working in close cooperation with industry to answer to our research questions has given us important insights over industry tools, processes and needs. Next, we will revisit our original research questions, and give a short discussion regarding our observations.

5.1 Research Questions Revisited

Considering the metrics defined above in the light of data available in version control system has given us high confidence that these metrics can be gathered

in a straightforward fashion, with certain exceptions. However, tools are needed to automate data collection process, and to help visualizing the results [31]. The exact answers to research questions are the following.

RQ1: Which relevant data for practical metrics are automatically created when using a state-of-the-art deployment pipeline?

Data can be collected regarding development, deployment and activation time from the tool chain that is used by the project. For the two former, data is precise, but requires following certain conventions, such as creating feature branches in the version data base for new features – a feature which is not supported by all version control systems. Regarding feature activation, the situation is less clear, since for numerous features it is not obvious when they are truly activated. When referring to a new function in the system, such as a widget in the screen for instance, the activation produces identifiable traces, whereas a change in layout or in libraries used are harder to pinpoint. Therefore, to summarize, with version control and usage monitoring system data, it is also possible to address the numbers of features in development, deployment, and activation, although for the latter with only some limitations and interpretations.

Regarding practicality, we feel that any team performing continuous delivery and deployment should place focus on metrics listed above. Based on discussions with the team developing Lupapiste, visualizing the data regarding features on the pipeline was found very useful, and exposing developers to it actually led to faster deployment and to less uncompleted work in the pipeline.

RQ2: How should the pipeline or associated process be modified to support the metrics that escape the data that is available?

While actions related to actual development are automatically stored in the version control system, end users' actions are not. Therefore, better support for feature activation is needed. This can not be solved with tools only but require project-specific practices. For instance, additional code could be inserted to record activation of newly written code, or aspect-oriented techniques could be used to trace the execution of new functions as proposed in [32].

In the present setup, there is no link to product management activities. In other words, the pipeline only supports developers, not product management. More work and an improved tool chain is therefore needed, which is also related to the above discussion regarding feature activation. However, it can be questioned if this falls within the scope of the pipeline, or should be considered separately as a part of product management activities.

RQ3: What kind of new metrics based on automatically generated data could produce valuable information to the development team

We presented data collection methods for collecting data for new metrics regarding the deployment pipeline. We proposed multiple new metrics for the deployment pipeline. For example, metric *Oldest Done Feature (ODF)*, which the

team of the case project found especially potentially useful, could be applied for measuring the current state of the deployment pipeline. Exposing such a metric to the team for example on the radiator screen in the team workspace, could improve the release cycle of the project.

We measured the values for the new metrics proposed for the case project. The team was producing more than one feature a day and making releases at least once a week. The *Oldest Done Feature (ODF)* at the time of research was only six days old. According to these metrics, the features are flowing fluently from development to the production environment.

5.2 Observations

To begin with, the deployment to production may have extra latency even in a state-of-the-art deployment pipeline. For instance, in the case project, many of the features suffered from a long latency of even weeks or months between the time the feature was done till the time when it was deployed to the production. The team was shortly interviewed about the obstacles why the features were not deployed to the production environment earlier. The obstacles were often related to features that had been merged to the develop-branch, which then accompanied the develop branch to a state where it was not possible to deploy anymore. For instance in one case, a key functionality was broken and the fix needed data from a design process.

The time after a new feature that has been deployed to the production environment and is waiting for users to use the feature, can be regarded as waste. To eliminate such, the users of the system must be informed regarding newly deployed features, and they also have to have the skills to use them. Because the users in the case project are the municipal authority users nationwide, an announcement sent by email as new features are introduced. Moreover, a wizard, which would tell about the new features, for example after login or in the context of the features, could help the users to find the new functionality.

We discussed about the proposed new metrics with the development team of the case project. *Oldest Done Feature* was found as the most useful metric that could possibly help the team to improve the flow of the pipeline. The team even considered that this kind of metric could be shown on the radiator screen – if the oldest feature is for example two weeks old, the radiator could indicate the problem in the pipeline. However, the actual usage of such a metric is not straightforward. There are times, when the develop branch is not deployable because of, for example, a major refactoring. In this kind of circumstances this kind of metric may disturb the team.

6 Conclusions

A metric should be used for a purpose. A modern deployment pipeline paves a highway for the features to flow from the development work to actual usage in the production environment. The tools on the pipeline produce a lot of data

regarding the development and deployment activities of the new features. We analyzed the tools and practices of an industrial single case study in order to identify which data are automatically created by the several tools of the deployment pipeline. The results show that data for many new useful metrics is automatically generated.

Based on this data, we defined several new metrics for describing the properties of the deployment pipeline. For instance, the metrics proposed can be applied to analyze the performance and the present status of the pipeline. The goal of metrics is to provide valuable information to the team to improve processes and the pipeline. Applying the metrics in a continuous delivery project setting can help to achieve this.

Acknowledgements

This work is a part of the Digile Need for Speed project (<http://www.n4s.fi/en/>), which is partly funded by the Finnish Funding Agency for Innovation Tekes (<http://www.tekes.fi/en/tekes/>). Persons in Figure 1 are designed by Paulo S Ferreira from thenounproject.com.

References

1. G. G. Claps, R. B. Svensson, and A. Aurum, "On the journey to continuous deployment: Technical and social challenges along the way," *Information and Software Technology*, vol. 57, pp. 21–31, 2015.
2. M. Poppendieck and T. Poppendieck, *Lean software development: an agile toolkit*. Addison-Wesley Professional, 2003.
3. K. Petersen and C. Wohlin, "Measuring the flow in lean software development," *Software: Practice and experience*, vol. 41, no. 9, pp. 975–996, 2011.
4. M. Fowler, "Agileversuslean," <http://martinfowler.com/bliki/AgileVersusLean.html>, 2008, retrieved: November 2014.
5. R. Shah and P. T. Ward, "Defining and developing measures of lean production," *Journal of operations management*, vol. 25, no. 4, pp. 785–805, 2007.
6. R. K. Yin, *Case study research: Design and methods*. Sage publications, 2014.
7. K. Beck, M. Beedle, A. van Bennekum, A. Cockburn, W. Cunningham, M. Fowler, J. Grenning, J. Highsmith, A. Hunt, R. Jeffries, J. Kern, B. Marick, R. C. Martin, S. Mellor, K. Schwaber, J. Sutherland, and D. Thomas, "The agile manifesto," <http://agilemanifesto.org>, 2001, retrieved: November 2014.
8. E. Kupiainen, M. V. Mäntylä, and J. Itkonen, "Why are industrial agile teams using metrics and how do they use them?" in *Proceedings of the 5th International Workshop on Emerging Trends in Software Metrics*. ACM, 2014, pp. 23–29.
9. M. Poppendieck and T. Poppendieck, *Leading lean software development: Results are not the point*. Pearson Education, 2009.
10. J. Humble and D. Farley, *Continuous delivery: reliable software releases through build, test, and deployment automation*. Pearson Education, 2010.
11. S. Misra and M. Omorodion, "Survey on agile metrics and their inter-relationship with other traditional development metrics," *ACM SIGSOFT Software Engineering Notes*, vol. 36, no. 6, pp. 1–3, 2011.

12. S. Neely and S. Stolt, "Continuous delivery? easy! just change everything (well, maybe it is not that easy)," in *Agile Conference (AGILE)*, Aug 2013, pp. 121–128.
13. M. Fowler, "Continuous delivery," <http://martinfowler.com/bliki/ContinuousDelivery.html>, retrieved: November 2014.
14. D. Ståhl and J. Bosch, "Modeling continuous integration practice differences in industry software development," *Journal of Systems and Software*, vol. 87, pp. 48–59, 2014.
15. M. Fowler, "Continuous integration," <http://martinfowler.com/bliki/ContinuousDelivery.html>, retrieved: November 2014.
16. J. Humble and D. Farley, *Continuous delivery: reliable software releases through build, test, and deployment automation*. Pearson Education, 2010.
17. A. Miller, "A hundred days of continuous integration," in *Agile, 2008. AGILE '08. Conference*, Aug 2008, pp. 289–293.
18. J. Humble, "Continuous delivery vs continuous deployment," <http://continuousdelivery.com/2010/08/continuous-delivery-vs-continuous-deployment/>, retrieved: November 2014.
19. J. Humble, C. Read, and D. North, "The deployment production line," in *Agile Conference*. IEEE, 2006, pp. 6–pp.
20. D. Feitelson, E. Frachtenberg, and K. Beck, "Development and deployment at facebook," *IEEE Internet Computing*, p. 1, 2013.
21. J. Humble, "Continuous delivery vs continuous deployment," <http://continuousdelivery.com/2010/08/continuous-delivery-vs-continuous-deployment/>, 2010, retrieved: November 2014.
22. B. Fitzgerald and K.-J. Stol, "Continuous software engineering and beyond: trends and challenges," in *Proceedings of the 1st International Workshop on Rapid Continuous Software Engineering*. ACM, 2014, pp. 1–9.
23. M. Kunz, R. R. Dumke, and N. Zenker, "Software metrics for agile software development," in *Software Engineering, 2008. ASWEC 2008. 19th Australian Conference on*. IEEE, 2008, pp. 673–678.
24. N. Modig and P. Åhlström, *This is lean: Resolving the efficiency paradox*. Rheologica, 2012.
25. M. Van Hilst and E. B. Fernandez, "A pattern system of underlying theories for process improvement," in *Proceedings of the 17th Conference on Pattern Languages of Programs*. ACM, 2010, p. 8.
26. P. Debois, "Devops: A software revolution in the making," *Cutter IT Journal*, vol. 24, no. 8, 2011.
27. T. Lehtonen, T. Kilamo, S. Suonsyrjä, and T. Mikkonen, "Lean, rapid, and wasteless: Minimizing lead time from development done to production use," in *Submitted to publication*.
28. V. Driessen, "A succesful git branching model." <http://nvie.com/posts/a-successful-git-branching-model/>, retrieved: November 2014.
29. K. Schwaber and M. Beedle, "Agile software development with scrum. 2001," *Upper Saddle River, NJ*, 2003.
30. L. Chen, "Continuous delivery: Huge benefits, but challenges too," *Software, IEEE*, vol. 32, no. 2, pp. 50–54, 2015.
31. A.-L. Mattila, T. Lehtonen, K. Systä, H. Terho, and T. Mikkonen, "Mashing up software management, development, and usage data," in *ICSE Workshop on Rapid and COntinuous Software Engineering*, 2015.
32. S. Suonsyrjä and T. Mikkonen, "Designing an unobtrusive analytics framework for java applications," in *Accepted to IWSM Mensura 2015, to appear*.

Metrics for Gerrit code reviews

Samuel Lehtonen and Timo Poranen

University of Tampere, School of Information Sciences, Tampere, Finland
samuel.lehtonen@gmail.com, timo.t.poranen@uta.fi

Abstract. Code reviews are a widely accepted best practice in modern software development. To enable easier and more agile code reviews, tools like Gerrit have been developed. Gerrit provides a framework for conducting reviews online, with no need for meetings or mailing lists. However, even with the help of tools like Gerrit, following and monitoring the review process becomes increasingly hard, when tens or even hundreds of code changes are uploaded daily. To make monitoring the review process easier, we propose a set of metrics to be used with Gerrit code review. The focus is on providing an insight to velocity and quality of code reviews, by measuring different review activities based on data, automatically extracted from Gerrit. When automated, the measurements enable easy monitoring of code reviews, which help in establishing new best practices and improved review process.

Keywords: Code quality; Code reviews; Gerrit; Metrics;

1 Introduction

Code reviews are a widely used quality assurance practice in software engineering, where developers read and assess each other's code before it is integrated into the codebase or deployed into production. Main motivations for reviews are to detect software defects and to improve code quality while sharing knowledge among developers. Reviews were originally introduced by Fagan [4] already in 1970's. The original, formal type of code inspections are still used in many companies, but has been often replaced with more modern types of reviews, where the review is not tied to place or time. Code reviews discussed in this paper are tool assisted modern code reviews.

Reviews are known to be very efficient in defect detection and prevention especially in large software projects [10, 1]. For the review process to be successful, reviews have to be done carefully and without unnecessary delays. To foster that, review process has to be motivating for developers while also being monitored and controlled to ensure the review quality. In this paper we are proposing a way to increase monitoring possibilities and controls by introducing different metrics to follow activities that take place during the review process. The introduced metrics are used to examine data that Gerrit [5], a web based light-weight code review tool, automatically stores during different events in Gerrit code review. This information can then be processed and prepared to be presented as graphs or key figures.

Code reviews in Gerrit are an interesting research target because of Gerrit's current position as a popular review tool among some of the leading open source projects. It is used in projects like LibreOffice, Wikimedia, Eclipse and Android Open Source Project [5].

The areas of measurement include different phases of the review process and review outcomes. The biggest focus area is on measuring the time that is spent on each activity and monitoring the number of reviews that lead into further code changes. The data provided by the metrics also establishes foundations for possible future research on how to improve the review processes even further, by optimizing individual activities.

After analyzing the metrics results from four different projects, we identified typical time division in code reviews, which was similar in all of the studied projects. Also we identified that review results depend on the person who is making the review.

The rest of the paper is organised as follows. In Section 2 we give a deeper introduction to code reviews and effects of code reviews to code quality. New metrics for measuring Gerrit code reviews are given in Section 3 and in Section 4 these metrics are applied to sample projects and findings are discussed. Last section concludes the work.

2 Code Reviews

Originally the reason for code reviews was mainly to spot defects and prevent any bad quality code from getting into codebase. Bachelli and Bird [1] reveal that while finding defects is still the main reason and motivation for reviews, there are also many other benefits that act as motivation to conduct reviews. These include at least sharing of knowledge, improved communication among developers and better overall software quality.

Successful code reviews enable a product with less defects and improved maintainability [16]. Improvements are not limited to code, but the whole project may benefit from the reviews as programming practices and other ways of work are discussed and refined during the reviews. Future work becomes easier as the development becomes smoother. Reviews are also known to speed up the whole project at later stages. When reviews are done at early stage, for example, when 10 % of the project is complete, rather than later stage when a lot of work has been done already, the lessons learned in reviews let the rest 90 % of the work to be better executed. This also mean better integrity and any fixes are easier to apply and are less likely to mess up the code structure. [6, 18]

Code reviews are demonstrated to be efficient way of detecting defects, but every developer does not spot defects with the same efficiency. Rigby et al. [14] and Porter et al. [13] researched the number of defects found per developer and found that more experienced developers discovered more defects than less experienced developers, which is natural because senior developers have more experience with the program than junior developers. The amount of code reviewed also has a big effect on review results. Statistical analysis of code reviews

by McIntosh et al. [11] shows that there is a significant correlation between code review coverage and the amount of post release defects. However, even with 100 % review coverage it is likely that some defects remain. On the other hand, when reducing the review coverage to around 50 %, it is almost certain that at least one defect per component will get through.

For addition to defect finding, code reviews have become an important knowledge sharing tool especially in projects where people are working in different locations. Reviews can substitute a huge amount of formal documentation what would have otherwise been needed to describe to other developers what changes have been made since the last version. Modern review tools like Gerrit are able to show all the changes made on a side by side comparison with color codes. When changes are seen by several developers, it increases their awareness and understanding of the software being developed. Communication between developers is also enhanced, especially when review tools are used, which make it easy for developers to leave inline comments or questions. As frequent communication is used to discuss problems, there is also better atmosphere to generate new ideas for implementing the code [Bachelli and Bird, 2013; Markus, 2009]. To further emphasize the importance of knowledge sharing during the reviews, the knowledge sharing and frequent communication among developers is known to improve the software quality. [12]

Because of the idea of constant feedback and sharing of written code with others, code reviews are also a powerful learning tool. New developers get feedback from more seasoned programmers which enable them to fortify their skills and to learn from each other's work. To summarize, code reviews can be considered as a very important defect detection tool and even more important when working with large projects, where software evolves and new versions rely on earlier work. Code reviews are considered to be the best tool for detecting evolvability defects while also being effective tool on finding functional defects.

Review processes in Gerrit [5] can vary between projects, but the basic workflow is always the same. There is the author of the code change, one or more reviewers and possibly approver and integrator. The most basic use case of Gerrit only involves the author and reviewer, but if we want to measure the process with better accuracy, at least integrator role is needed to give details about integration times. The approver is usually a senior developer who makes the final acceptance decision for the code change.

While the the metrics proposed in this paper are best used with projects which utilize approver and integration flags, it is also possible to use them without those roles, but the measurement results will be more limited.

Typical code review process proceeds as follows:

1. Author uploads the code change. This step could also include automatic unit testing.
2. When author considers that the change is ready, he/she assigns reviewers for the change and gives the change a code review with a score of +1.
3. Reviewers review the code and give it either +1 or -1. If the reviewer doesn't find anything to improve, a score of +1 is given. Whenever there is something

- to improve, -1 should be given. The author has to address the issues raised by the reviewer before the process can continue.
4. When there are enough reviews, approver makes the final decision to pass the change for integration by giving it +1 or sending it back for rework by giving it -1. If approver is not used, the code change goes straight into integration.
 5. After the change is approved, integrator starts the integration process and gives the change +1 to mark the beginning of integration. If the code cannot be integrated, the integrator gives -1 and sends the change back to author for rework. Any merge conflicts also result as integrator -1, which typically means that the change goes back to author for fixing.
 6. If the change can be merged and passes final testing, it is merged into the repository.

3 Metrics for measuring Gerrit code reviews

The motivation to have quality and efficiency metrics is that without them, process improvement and identification of the best practices would be impossible. [Sommerville, 2011; Kupiainen et al., 2015] While Gerrit gives good tools to organize reviews, it only provides limited amount of metrics about the changes. To improve code reviews in Gerrit, sufficient metrics are essential. Measuring activities in Gerrit allows more control over the code reviews which helps managers to ensure steady flow of code changes. Although Gerrit doesn't provide metrics, it has all the necessary data for building the metrics stored in a MySQL database.

The goal of the metrics is to present the data extracted from the review process to developers and managers, in a way which helps them to monitor and control the review process, in order to pick out problems, but also to motivate for better performance. Based on the knowledge gained from the figures, managers and developers themselves could take action to improve the process or make corrective actions. Based on the knowledge gained from the metrics, it is possible to refine the review process for better overall software quality with fewer defects. Metrics also provide valuable information of the whole process, which supports the improvement of code review process in the company.

The measurements can be divided into two categories: time and quality. The time metrics are focused on measuring the velocity of different activities and the time spent for different tasks. The quality metrics are measuring various items which can hint on process quality and code quality. In the following the metrics are discussed in detail.

3.1 Review time

The review time is one of the most fundamental metrics we have. It describes how long it takes for the change to get the necessary amount of positive code reviews and an approval by the nominated approver. The review time starts when a developer has uploaded a change and has given it code review +1 in

Gerrit, to acknowledge that the change is ready to be reviewed. The +1 given by the author is also a sign that the author has tested the change and considers it ready to be submitted to the baseline. The +1 that the change owner gives is an agreed practice used in the case company. It is not a forced process by Gerrit, but a good way to communicate that the author approves the change ready for review. This is especially useful if continuous integration and unit testing is used alongside Gerrit.

The review time consists of the time it takes for reviewers to start the review after they have been invited combined with the time it actually takes to complete the review. Measuring the actual review time, meaning the time how long it takes for the reviewer to actually review the code, cannot be automatically measured and it has not been seen convenient to add an additional flag that the reviewer has to set before he or she starts to review the code, especially if the review is not completed within a session.

The review time also includes the time that the author uses for fixing the change whenever it receives a -1. There could be multiple fixing rounds until the change can be fully accepted by the reviewers. This loop continues until the change has received required amount of positive reviews without any -1's and is thought to be ready for merge. The final decision is done by the approver whose +1 also ends the review time. If approver is not used, integrator +1 could also mark the end of review time.

Review time is an important measure of how quickly new changes are ready for integration. Long review times slow down the whole process, especially when there are dependencies present. There is a couple of ways reducing the review time. The most obvious way is that the reviewers should review the change soon after being invited for a review. Another way to easily reduce the review times is to promote the communication during the code reviews so that everyone involved is aware when reviews or fixes are needed. In urgent issues it is best to send email directly or have talk face to face to get the things moving. Messages left to single change in Gerrit may remain unseen for a while or are easily forgotten when developers are simultaneously involved in numerous open changes.

The review time metric allows managers to follow review times and react to any longer than usual times. Long review times could be caused by a reviewer forgetting or neglecting the review, or delay may be due to an absence of a developer. If reviews pile up because of absence of some person, change author should nominate another reviewer, so that the process can move on. Long review times can also be a result of poor code quality which generates multiple -1's. Received reviews by a developer are also measured and that can be used to complement review times measurement, to better understand where the problem with long review times lies. Within the company it is useful to compare review times of different projects and discuss why some projects achieve better review times than others. Without these measures it would be inconvenient to try to determine whether reviews take reasonable amount of time or compare them with other projects.

3.2 Pre-review time

When the change is first uploaded to Gerrit the pre-review time starts. The purpose of measuring the pre-review time is to separate the time that is used by the author to fix the change after integration and tests from the time that is used to complete the code reviews. After uploading the change, the source is built and tested for errors by using automated tools. If the patch passes the tests it is possible to set the patch ready for review by giving it code review +1. Before giving +1 the author can still make changes by uploading a new patch-set to address any issues the author may have come across during the tests. Pre-review time ends at the moment the code review +1 is given by the author. This also marks the beginning of review time.

The pre-review time measures how quickly uploaded change is ready for review. Long pre-review time indicates that there has been need for rework after the initial upload and first tests. Typically this time should be less than a day and any longer times need a good reason and managers should take action whenever long pre-review times are noticed. The owner +1 is not a mandatory action in Gerrit, but is applied practice in the case company where the metrics were introduced. Although it is optional in Gerrit for the author to give +1, it is a good indicator that the author wants the change to be reviewed. It is especially useful if change is queued for tests or continuous integration after the upload. The author can then give the +1 when all the tests have fully passed. By separating these test activities and possible fixing to pre-review time, we can achieve more accurate review time.

3.3 Pre-integration time

The pre-integration time is the time after the review is completed, but the actual integration process is yet to start. This time should be minimized as it is time when no one is working with the change, but it is just waiting for integrator to start the integration process. Basically pre-integration time describes how long it takes for the integrator to start the integration process. The integration process should start as soon as the review is completed and approver has given +1. This means that the pre-integration time should not be long as there is no reason for the integrator not to start the integration. Typically the integration should be started the same day as the review is completed or the next day at the latest. The pre-integration time shouldn't therefore be more than 24 hours excluding weekends.

3.4 Integration time

The integration time is a measure of how long it takes for the change to be merged into the trunk after the review process is completed. Integration time includes various phases which take place during the integration. Depending on a project, integration time could include very different tasks. Validation processes and build processes depend on platform and the nature of code being tested.

Some changes may get integrated pretty fast, while in some other projects it is a slow process. Comparing integration times between projects is therefore not very useful. Comparison should rather be done within the same project.

At its best, case scenario integration starts almost immediately when the change has been approved. During the integration time the integrator tries to merge the patch with rest of the code and if there are no conflicts and the change passes tests, it can be integrated.

However, sometimes there are merge conflicts which need to be resolved before the change can be fully merged and tested. When a conflict appears, the integration process is stopped and integrator gives the change -1. If the conflicts are small, the integrator can then fix them and a review of changes by the change author is enough. If there are major conflicts then the change needs rework and it has to go through the review process all over again. Also if change cannot pass integration testing, it is returned to the author for rework. The time spent to rework can be measured in a different metric called integration fix time.

Ideal integration time is less than 24 hours, which means that the integration is done the next day at the latest, after the change has been approved. There should not be any excuse for the integrator not to start the integration as soon as the code change has been approved for integration. For meaningful results, integration times should only be compared within same project or within projects with similar workflows. This is because projects working with for example with application layer or firmware have very different validation processes during the integration.

Whenever the change gets rejected during the integration, the integration reject time metric can be used to calculate time wasted to rejected changes. It is simply a measurement of how long it takes, before the change is rejected after being approved by the approver. There are many reasons why a change could be rejected during the integration. The reason can be some known issue in the change which prevents integration or the integrator -1 could be a result from merge conflicts or failed tests. Any change receiving integrator -1 needs fixing. Often the reason is of merge conflicts. In a case of simple conflicts the integrator can fix them immediately, but most often the problems are more profound and the change is sent back to the author for rework. The time used for reworking with the change is another measure called integration fix time.

The reject time depends on at what point the integrator notices that the change cannot be merged. The most common situation is that the change is based on too old baseline which is no more compatible with the current codebase. This results as errors when merging or building. Typically integrator starts the build at the end of the day and reviews the results the next morning at the latest. Because of this, the reject time is typically around 12-60 hours including weekends. The integration fix time is the additional time that is used to fix the change after it has been rejected during the integration. This time includes the time the developer uses to actually fix the change, but also the time it takes to review and approve this new fix.

After measuring the reject time and fix time, it is clear that any rejected change becomes expensive as it takes many days for the change to be fixed and reviewed again. If the developer used few extra hours to rebase and validate the change to be ready for integration at the first place, it would reduce the overall lifetime by tens of hours at the minimum.

3.5 Quality metrics

The purpose of quality metrics is to accompany time measurements by giving a perspective on how code reviews are made. When review times of a project suddenly fall while reviews constantly result with only +1's, it hints for change in review motivation and should be investigated. To maintain good code quality reviews have to be performed with care. Reviews are successful when defects are found. However if someone's patches constantly gets a lot of minuses there might be a need for discussion with the developer why that is happening. Also if certain reviewer always finds something to improve, it is useful to investigate if all those minus ones are necessary.

Metrics were also established to ensure that the reviews are done properly. When schedules are tight and deadlines are closing in, it is tempting to skip the reviews. However, any time saved here could cause even bigger delays and costs later in the development lifecycle. To recognize if any team is trying to catch its schedule by hurrying or ignoring the reviews, a few different quality measurements were introduced.

The main motivation for quality measurements is monitoring that the reviews are conducted following the good code review practice. The measurements used are the number of reviews given and received. The motivation for not giving negative reviews is often a need to speed up the review process. When a project is in hurry it may be tempting to hurry the code reviews too or skip it altogether so that feature complete milestones can be achieved. Ignoring code reviews or testing is very shortsighted and it is likely that any schedule advantage gained by reducing reviews or testing, will later be lost because the end product is too buggy or hard to maintain. By measuring the review activity, problems like this are tried to be avoided. To complement these, there is also a metric to measure review results. Reviews should have quite steady amount of +1s and -1s. If some project starts to receive only +1's it might be better to investigate what is the reason.

3.6 Average number of patch-sets

One straight forward metric for measuring change quality is the number of patch-sets. In addition to change quality it can be used for measuring process smoothness. Smaller amount of patch-sets means that the code change has needed less fixes until it has been integrated in to the codebase. The number of patch-sets is measured with two different ways. There is the absolute number of patch-sets per change, which makes it possible to easily point out individual changes with larger than normal count of patch-sets. Then there can be a histogram chart

which shows the distribution of number of patch-sets needed for a change. The histogram can quickly tell that the most of the changes go through with only one patch-set, which may be due to very trivial nature of the change or +1's are given too softly in code review. For better understanding these patches need to be investigated individually. By following this distribution over time, a normal value for each frequency can be established and big variances to that could then be a sign to start investigation. A good number of patch-sets is around 2-5, which implies that the initially uploaded patch did not need numerous fixes, but the reviewer has found something to comment or improve. As each patch-set represents an improvement to the earlier patch, this can be used to give an estimate on how effectively code reviews are used to improve the code quality. The interesting thing to follow are situations where change receives very few or large amount of patch-sets. If the change receives only one or no fixes at all, it indicates that reviewers may not have looked the code at all, or have decided not to give feedback on minor issues. It is rarely the case that the first version of the code change is perfect and there is no room for improvement.

On the other hand a change with a lot of patch-sets indicates that the change has needed many fixes and has probably been under development for a long time. This can be because the change is big and requires a lot of work phases or the change could be fundamentally problematic and needs a lot of fixes to get it working with the rest of the code. If the number of changes with large number of patch-sets increases, it should be investigated whether the changes are too complex and if they should be divided into smaller ones. Sometimes changes need to wait for other changes being finished first. For example, in the case of dependencies the change might need numerous rebases while it waits the parent change being finished. Rebasing means applying latest changes from the master branch in Git. This can happen multiple times and that increases the number of patch-sets although there is no quality issues whatsoever. Other explanation is that the change has needed many fixes and has been somehow problematic.

If there is always only one patch-set, it means that reviews may not have been done properly or with enough care. While the main motivation of code reviews is to find defects, it is also important to try to improve the code. The code is not improved if the reviewer doesn't provide any feedback on the proposed change. Therefore, it is positive to see more than just one patch-set. From the code improvement point of view, it may not be the ideal situation to have minimum number of patch-sets. On the other hand, if the amount of patch-sets is very high, it often means that the patch is too complex or that the programmer is not doing good enough job. Comparison of number of patch-sets should be done within a same project on weekly or monthly basis, as the amount of patch-sets vary a lot between projects due to the different nature of development. High patch-set count is a good indicator of change complexity as more complex changes typically require more revisions, which increases the complexity even more. The complexity should be reduced by making smaller changes, which would make reviews easier and dependent changes would also get merged faster.

An important metric for determining the state of the code reviews is the count of currently open code changes. The difference between opened and closed items gives the number of currently open items, which is a useful measure to get an insight of current pace of opening and closing changes. If number of open patches goes up for extended period of time, it immediately tells that there are some problems in the process which need attention. If open items pile up, they are harder to manage and dependency management and the number of rebases needed become an issue. This metric can be accompanied with a count of abandoned changes, which measures how many of the uploaded code changes are abandoned and not integrated.

3.7 Measuring given and received reviews

Measuring review results shows how much negative and positive reviews are given. As part of the motivation for code review is code improvement, it is important that changes receive also negative reviews. Whenever a change receives -1, it means that reviewer has noticed something to comment. Comments are usually a good thing, because that implies that the code is being improved and discussed. Only time when comments may be useless are false positives, where reviewer thinks something as a defect when it is not. When this was discussed with developers, the consensus was that false positives are very rare. Measuring the amount of positive and negative reviews given by developer is also an important measure from the quality control perspective, because if some reviewer gives constantly only +1's he/she may not review thoroughly or is too soft in giving reviews. It is unlikely that the changes are always perfect.

4 Findings

To understand how time is allocated on average between the four major activities in Gerrit code review process, data was collected from four different Gerrit projects with over 200 changes per month each. The project sizes varied from 30 to 100 developers per project. Each researched project had teams based in several locations around the world. More detailed illustrations of project data can be found from Lehtonen's [8] thesis.

The results gained from the four projects are shown on Table 1. The integration reject time and integration fix time are eliminated because the event of integrator intervention is rare and hardly ever happens in most of the projects. Project number one is suffering from a couple of changes where integration time is extraordinary long, which partly explains the share of integration time being 36 % from the change life time.

From the analysis of four projects with different number of people and different tasks it can be seen that the division of time spent is still quite similar in all of the projects. The times in each project are also fairly long at some stages, so there is clearly some room for improvement. The change stays in Gerrit for around one day before the author declares it ready for review. In the investigated

Table 1. Distribution of time in Gerrit code review. Times are hours with percentage of total change life cycle.

	Pre-review time	Review time	Pre-integration time	Integration time
Project 1	28 (10 %)	99 (36 %)	49 (18 %)	101 (36 %)
Project 2	30 (11 %)	166 (62 %)	28 (10 %)	44 (17 %)
Project 3	14 (7 %)	103 (54 %)	16 (14 %)	49 (25 %)
Project 4	14 (12 %)	119 (59 %)	20 (10 %)	38 (19 %)
Average	24 (10 %)	122 (53 %)	31 (13 %)	58 (24 %)

projects the actual review time took 36 – 62 % from the total time, including possibly multiple rounds of reviews, which makes it the most time consuming activity. This is no surprise as that is when most of the work during review process is done. However, the hours spent is currently quite a lot. As a percentage, the share of reviews could increase, as now around 10 % or 30 hours is spent during the pre-integration time, which is the time the change waits for integration to begin. This is long time, which partly is explained with people working in different time zones, but still there is clearly room for improvement. In integration times there is quite a bit of variance, which is because of the different nature of the projects and amount of required integration testing.

The integration fix time was excluded from the individual projects due to too small sample size, but when a group of projects was investigated where the total number of changes goes beyond 1000, the typical time needed to fix a change after rejection in integration, requires 100 – 150 hours, before it is again ready for integration. From that finding, it is safe to say that it is worth investing time to get the change ready for integration at the first place. The fix time quickly accumulates because developer needs time to do the fixing, but the change also needs a new round of code reviews and an approval.

Another place where time can be saved in researched projects, is in the code reviews. Average time for the code change to complete the code review stage is 122 hours, which equals to roughly 5 days. The time a reviewer actually uses to review the change is typically less than an hour, which means that the change waits several days for reviews to be completed. Even in the best of the researched projects, the average review time was 99 hours. When examining the frequency of minus ones in code reviews, the variance between projects is noticeable. The share of changes going through without any negative reviews is shown in Table 2.

Table 2. Percentage of changes which receive at least one code review -1.

	Share of rejects
Project 1	73 %
Project 2	62 %
Project 3	35 %
Project 4	27 %

The project which has received the most rejects in code reviews has almost three times the number of rejects than the project with least rejects. However, this does not always correlate with the number of patch-sets. For example, in a project where only 27 % of changes received a negative review, every change had two or more patch-sets, meaning that the fixes were self-motivated or due to comments given together with positive review, which is against the agreed practice. In the project where 35 % of the changes went through with at least one negative review, share of changes with only one patch-set was around 20 %.

The number of patch-sets is another quality metric and is good in pointing out complex code changes. In an optimal case the change receives two or three patch-sets which indicates, that there has been some improvements to the initial code change. If changes get continuously integrated with only one patch-set, it should be brought to discussion, because the idea of code reviews is to improve the code and when the first version of the code is always accepted, there will be no improvement. On the other hand if a change has much more patch-sets than two or three, it tells that the change has some serious problems, because so many fixes has been needed.

Beller et al. [2] reported a case where the figure of undocumented fixes by the author accounted only 10 – 22 % which is significantly less than in most projects that were researched. That might be due to different development processes, because in the projects the unit tests and continuous integration is done after the change has been uploaded into Gerrit.

A reason which may explain the numbers, is that the sizes of changes in the projects with high rate of zero negative reviews was often very small, 5-10 lines. This means that there is very little space for defects. To support this observation, the project that scored the highest number of rejections during the code review, also has considerably larger change sizes on average. Contrary to the findings by Beller et al. [2], who examined small software projects, it was found that reviewer do have significant effect on review results. Some reviewers truly are stricter than others. There are also other factors which influence the number of negative reviews, like cultural background. Beller's study is based on two small projects while our sample size is hundreds of employees working within one company, but in very different projects with their own ways of work.

5 Conclusions and discussion

Code reviews have been around for almost 40 years, but the practice is still evolving. Modern code reviews are not much studied and without many concrete best practices being established. In an effort to recognize new best practices for code reviews in Gerrit code review, this paper has studied code reviews in a case company and introduced different metrics for measuring the reviews in a number of projects.

The main measurement areas are the time spent in review process and the frequencies of various actions in order to establish foundations for measuring code review velocity and quality.

In this paper we have demonstrated that measuring code review process with various metrics, can help controlling the review process on whole, while enabling improvement of the review process. The metrics give valuable information to management and developers including how quickly new code changes are reviewed and how long it takes for them to become integrated. It was also examined what are the stages in the process that need the most improvement. The metrics are most useful in large projects where keeping track of single changes would be too laborious.

When places of delays are identified and other metrics like number of negative and positive reviews given are measured, it is possible for the management to improve the efficiency of the review process. In a case of long change life times the key is to identify the factors which slow the process down and apply corrective actions. By measuring the number of positive and negative reviews per developer, it is possible to monitor the review activity and quality. If changes constantly receive low amount of negative reviews management should encourage stricter review policies so that more negative reviews are given and the code gets therefore improved more. In addition, the number of given feedback can be monitored and if very little feedback is given, the reviewers are encouraged to give more comments on the code changes.

When reviews and testing are applied properly, the speed of development becomes the most important thing to measure, as in the end the ability to create software fast is the thing that allows to gain the competitive advantage in the industry. From these time measurements, the most interesting one measures how quickly new code changes can be integrated after being uploaded to Gerrit.

Measuring code quality is also very important as code with lot of defects, functional or evolvability will cause additional costs later in the lifecycle. These metrics do not directly measure the code quality. The focus is more on measuring areas which do have secondary effect on quality. These include the number of rejects during review and number of patch-sets. From these two measurements it is possible to draw conclusions of the initial quality of the changes. When there are a lot of patch-sets, it typically means problems. When compared with data of received reviews it is possible to tell whether the change was well crafted. A lot of negative reviews combined with long change lifetime is a signal of bad code quality.

5.1 Limitations

The usage of the metrics introduced here is partly limited to Gerrit and some of the metrics require certain configurations. However, especially the velocity metrics are quite generic, and can be also used with other code review systems. While the interface could be different, the basic principles are typically similar, with change author and reviewers. By using the time stamps from reviews it is possible to calculate at least pre-review and review times. The proposed quality metrics are more Gerrit specific, but the concept is not tied to any unique functionality and can be copied to be used with other systems as well.

The sample size in the findings is four projects, all from the same organization, which sets some limitations to the credibility of the study. On the other hand, the studied projects are largely self organized, working with different software components, which makes them heterogeneous. Therefore, the sample data can be considered diverse enough to validate the observations.

5.2 Further research

In addition to the measurement areas covered in this paper, there are number of other possibilities for measurements. The metrics that were presented do not go deep into the factors behind the results. There are, for example, many variables that affect the code review process and could be used in later studies to complement the proposed metrics.

Tomas et al. [17] list and discuss few of the variables in detail including code complexity, code dependencies, number and size of comments, amount of code smells and code design. These additional metrics can be used to give more detail on why reviews take certain time or how effective they are in improving the code quality. To get deeper understanding of the factors affecting the metrics results, the future work with the metrics could, for example, take the variables studied by Tomas et al. and combine them with the metrics introduced in this paper.

The metrics still do not measure how long it takes for the reviewer to start the review after the change has been made available for review. Neither do the metrics measure how long it actually takes to complete the review. This would require adding a new functionality to Gerrit where reviewer can set the review started. However this is a manual process and might be hard to measure accurately as it relies to the fact that reviewer always remembers to set review started. Also, if review is not completed at a one session, the measurement would be skewed. However, this is very interesting information and the possibility to measure this will be investigated. Another additional measure could be review start time. It would be a measure of how long it takes for the change to receive its first review. This would give some insight on how quickly developers are reacting to review requests as currently measured review time only measures the total time spent on reviews.

References

1. Bacchelli, A., Bird, C.: Expectations, outcomes, and challenges of modern code review. In Proceedings of the 2013 International Conference on Software Engineering (2013), IEEE Press, 712-721.
2. Beller, M., Bacchelli, A., Zaidman, A., Juergens, E.: Modern code reviews in open-source projects: which problems do they fix? In: Proceedings of the 11th Working Conference on Mining Software Repositories (2014), ACM, 202-211.
3. Cheng, T., Jansen, S., Remmers, M.: Controlling and monitoring agile software development in three Dutch product software companies. In: Proceedings of the 2009 ICSE Workshop on Software Development Governance, (2009), IEEE Computer Society, 29-35.

4. Fagan, M.E.: Design and Code Inspections to Reduce Errors in Program Development. *IBM Systems Journal*. 15(3) (1976), 182-211.
5. Gerrit - web-based team code collaboration tool, <https://code.google.com/p/gerrit/>
6. Jones, C.: 2008. Measuring Defect Potentials and Defect Removal Efficiency, *The Journal of Defense Software Engineering*. June 2008. 11-13.
7. Kupiainen, E., Mäntylä, M.V., Itkonen, J.: Using metrics in Agile and Lean Software Development - A systematic literature review of industrial studies. *Information and Software Technology*. 62 (2015), 143-163.
8. Lehtonen, S.: Metrics for Gerrit code reviews, University of Tampere, School of Information Sciences, master's thesis, 62 pages, 2015.
9. Markus, A.: Code reviews. *SIGPLAN Fortran Forum* 28, 2, (2009) 4-14.
10. McConnell, S.: *Code Complete*, 2d Ed.. Microsoft Press, 2004.
11. McIntosh, S., Kamei, Y., Adams, B., Hassan, A.E.: The impact of code review coverage and code review participation on software quality: a case study of the qt, VTK, and ITK projects. In: *Proceedings of the 11th Working Conference on Mining Software Repositories*, (2014), ACM, 192-201.
12. Perry, D.E., Staudenmayer, N.A., and Votta, L.G.: 1994. People, organizations, and process improvement. *Software*, IEEE. 11(4) (1994), 36-45.
13. Porter, A., Siy, H., Mockus, A., Votta, L.: 1998. Understanding the sources of variation in software inspections. *ACM Transactions on Software Engineering Methodology*. 7(1) (1998), 41-79.
14. Rigby, P.C., German, D.M., Cowen, L., Storey, M.: Peer Review on Open-Source Software Projects: Parameters, Statistical Models, and Theory. *ACM Transactions on Software Engineering Methodology*. 23(4) (2014), Article no. 35.
15. Sommerville, I.: *Software engineering*, Ninth edition. Addison-Wesley, 2011.
16. Thongtanunam, P., Kula, R.G., Cruz, A.E.C., Yoshida, N., Iida, H.: Improving code review effectiveness through reviewer recommendations. In: *Proceedings of the 7th International Workshop on Cooperative and Human Aspects of Software Engineering*, (2014), ACM, 119-122.
17. Tomas, P., Escalona, M.J., Mejias, M.: Open source tools for measuring the Internal Quality of Java software products. A survey, *Computer Standards & Interfaces*. 36(1) (2013), 244-255.
18. Wieggers, K.E.: 2002. Seven Truths About Peer Reviews, *Cutter IT Journal*, July 2002. Retrieved from <http://www.processimpact.com>.

Test Suite Evaluation using Code Coverage Based Metrics

Ferenc Horváth¹, Béla Vancsics¹, László Vidács², Árpád Beszédes¹,
Dávid Tengeri¹, Tamás Gergely¹, and Tibor Gyimóthy¹

¹ Department of Software Engineering
University of Szeged
Szeged, Hungary

{`hferenc,vancsics,beszedes,dtengeri,gertom,gyimothy`}@inf.u-szeged.hu

² MTA-SZTE Research Group on Artificial Intelligence
University of Szeged
Szeged, Hungary
`lac@inf.u-szeged.hu`

Abstract. Regression test suites of evolving software systems are often crucial to maintaining software quality in the long term. They have to be effective in terms of detecting faults and helping their localization. However, to gain knowledge of such capabilities of test suites is usually difficult. We propose a method for deeper understanding of a test suite and its relation to the program code it is intended to test. The basic idea is to decompose the test suite and the program code into coherent logical groups which are easier to analyze and understand. Coverage and partition metrics are then extracted directly from code coverage information to characterize a test suite and its constituents. We also use heat-map tables for test suite assessment both at the system level and at the level of logical groups. We employ these metrics to analyze and evaluate the regression test suite of the WebKit system, an industrial size browser engine with an extensive set of 27,000 tests.

Keywords: code coverage, regression testing, test suite evaluation, test metrics

1 Introduction

Regression testing is a very important technique for maintaining the overall quality of incrementally developed and maintained software systems [5, 21, 14, 4]. The basic constituent of regression testing, the *regression test suite*, however, may become as large and complex as the software itself. To keep its value, the test suite needs continuous maintenance, *e.g.* by the addition of new test cases and update or removal of outdated ones [12].

Test suite maintenance is not easy and imposes high risks if not done correctly. In general, the lack of systematic quality control of test suites will reduce their usefulness and increase associated regression risks. Difficulties include their ever growing size and complexity [13] and the resulting incomprehensibility. But

unlike many advanced methods for efficient source code maintenance and evolution available today (such as refactoring tools, code quality assessment tools, static defect checkers, etc.), developers and testers have hardly any means that may help them in test suite maintenance activities, apart from perhaps test prioritization/selection and test suite reduction techniques [21], and some more recent approaches for the assessment of test code quality [2].

Hence, a more disciplined quality control of test suites requires that one is able to understand the internal structure of the test suite, its elements and their relation to the program code. Without this information it is usually very hard to decide about what parts of the test suite should be improved, extended or perhaps removed. Today, the typical information available to software engineers is limited to knowing the purpose of the test cases (the associated functional behavior to be tested), possibly some information about the defect detection history of the test cases and – mostly in the case of unit tests – their overall code coverage.

Furthermore, most of previous work related to assessing test suites and test cases are defect-oriented, *i.e.* the actual amount of defects detected and corrected are central [6]. Unfortunately, past defect data is not always available or cannot be reliably extracted. Approximations about defect detection capability may be used instead which, if reliable, could be a more flexible and general approach to assess test suites. In this work, we employ *code coverage*-based approximations that are based on analyzing coverage structures related to individual code elements and test cases in detail (code coverage is essentially a signature of dynamic program behavior reflecting which program parts are executed during testing, often associated with the fault detection capability of the tests [21, 22]).

In our previous work [16], we developed a method for a systematic assessment and improvement of test suites (named *Test Suite Assessment and Improvement Method – TAIME*). One of its use cases is the assessment of a test suite which supports its comprehension. In TAIME, we decompose the test suite and program code into logical groups called *functional units*, and compute associated code coverage and other related metrics for these groups. This will enable a more elaborate evaluation of the interrelationships among such units, thus not limiting the analysis to overall coverage on the system level. Such an in-depth analysis of test suites will help in understanding and evaluating the relationships of the test suite to the program code and identify possible improvement points that require further attention. We also introduce “heat-map tables” for more intuitive visualization of the interrelationships and the metrics.

In this paper, we adapted the TAIME approach to use it for assessment purposes and verified the usefulness of the method for the comprehension of the test suite of the open source system WebKit [20], a web-browser layout engine containing about 2.2 million lines of code and a large test suite of about 27 thousand test cases. We started from major functional units in this system and decomposed test cases and procedures³ into corresponding pairs of groups. We

³ We use the term procedure as a common name for functions and methods

were able to characterize the test suite both as a whole and its constituents and eventually provide suggestions for improvement.

In summary, our contributions are the following:

1. We adapted the TAIME method to provide an initial assessment of a large regression test suite and present a graphical representation of the metrics computed from code coverage.
2. We demonstrate the method on a large open source system and its regression test suite, where we were able to identify potential improvement points.

The paper continues with a general introduction of the analysis method (Section 2) and an overview of the metrics used for the evaluation (Section 3). Then, we demonstrate the process by evaluating the WebKit system: in Section 4, we report how functional units were determined, while the assessment itself is presented in Section 5, where we also introduce the heat-map visualization. Relevant related work is presented in Section 6, before we conclude and outline future research in the last section.

2 Overview of the Method

Our long-term research goal is to elaborate an assessment method for test suites, which could be a natural extension to existing software quality assessment approaches. The key problem is how to gain knowledge of overall system properties of thousands of tests and at the same time understand lower level structure of the test suite to draw conclusions about enhancement possibilities. In our previous work [16], we proposed a method to balance between the two extreme cases: providing system level metrics is not satisfactory for in depth analysis; while coping with individual tests may miss higher level aims in case of large size test suites.

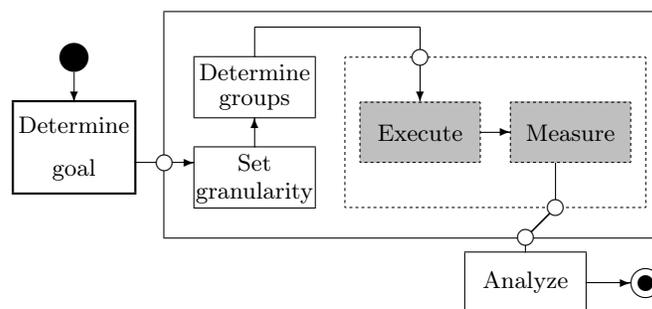


Fig. 1. Overview of the adapted TAIME approach

We adapted the TAIME approach to use it for assessment purposes. The main steps of the proposed test suite analysis method are presented in Figure 1. The basic idea is to decompose the program into features that will divide the

test suite and the program code into different groups. This can be done in several ways, e.g. by asking the developers or, as we did, by investigating the tests and the functionality what they are testing. After we have a set of features we can decompose the test suite and the code into groups. Those tests, which are intended to test the same functionality, are grouped together (*test groups*). The features are implemented by different (possibly overlapping) program parts (such as statements or procedures), which we call *code groups*. In the following, we will use the term *functional unit* to refer to the decomposed functionality, which we consider as a pair of associated *test group* and *code group* (see Figure 2). The whole analysis process is centered around functional units, because by this division the complexity of large test suites can be handled more easily. The decomposition process (the number of functional units, the granularity of code groups and whether the groups may overlap) depend on the system under investigation. It may follow some existing structuring of the system or may require additional manual or automatic analysis. More on how we used the concept of functional units in our subject system will be provided later in the paper.

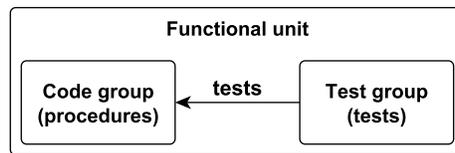


Fig. 2. Functional units encapsulate procedures and corresponding tests

According to the GQM approach [3], the aim of test suite understanding and analysis can be addressed by various questions about functional units. The proposed method lets us answer questions about the relation of code and test groups to investigate how tests intended for a functionality cover the associated procedures; and also about the relation of an individual functional unit to other units or to the test suite.

3 Metrics for Test Suite Evaluation

In this section we give an overview of metrics, coverage and partition that will be used for test suite evaluation. All metrics will be defined for a pair of a test group T and a code group P . Let T be the set of tests of a functional unit and P be the set of procedures which usually belong to the same functional unit.

3.1 Coverage Metric (COV)

We define the *Coverage metric* (COV) as the ratio of the number of procedures in the code group P that are covered by test group T . We consider a procedure as covered if one of its statement was executed by a test case. This is the traditional

notion of code coverage, and is more formally given as:

$$\text{COV}(T, P) = \frac{|\{p \in P \mid p \text{ covered by } T\}|}{|P|}.$$

Code coverage measures are widely used in white-box test design techniques, and they are useful, among others, to enhance fault detection rate, drive test selection and test suite reduction. This metric is easy to calculate based on the coverage matrix produced by test executions. Possible values of COV fall into $[0, 1]$ (clearly, bigger values are better).

3.2 Partition Metric (PART)

We define the *Partition metric* (PART) to express the average ratio of procedures that can be distinguished from any other procedures in terms of coverage. The primary application of this metric is to support fault localization [19]. The basis for computing this metric are the coverage vectors in a coverage matrix corresponding to the different procedures. The equality relation on coverage vectors of different procedures will determine a *partitioning* on them: procedures covered by the same test cases (*i.e.* having identical columns in the matrix) belong to the same partition. For a given test group T and code group P we denote such a partitioning with $\Pi \subseteq \mathcal{P}(P)$. We define $\pi_p \in \Pi$ for every $p \in P$, where

$$\pi_p = \{p' \in P \mid p' \text{ is covered by and only by the same test cases from } T \text{ as } p\}.$$

Notice that $p \in \pi_p$ according to the definition. Having fault localization application in mind, $|\pi_p| - 1$ will be the number of procedures “similar” to p in the program, hence to localize p in π_p we would need at most $|\pi_p| - 1$ examinations. Based on this observation, the PART metric is formalized as follows, taking a value from $[0, 1]$:

$$\text{PART}(T, P) = 1 - \frac{\sum_{p \in P} (|\pi_p| - 1)}{|P| \cdot (|P| - 1)}.$$

The numerator can be also interpreted as the sum of $|\pi| \cdot (|\pi| - 1)$ values for all different partitions. It will be best (1) if test cases partition procedures so that each procedure belongs to its own partition, while it will be the worst (0) if there is only one big partition with all the procedures.

$$\mathbf{C} = \begin{matrix} & \begin{matrix} p_1 & p_2 & p_3 & p_4 & p_5 & p_6 \end{matrix} \\ \begin{matrix} t_1 \\ t_2 \\ t_3 \\ t_4 \\ t_5 \\ t_6 \\ t_7 \\ t_8 \end{matrix} & \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 1 \\ 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 \end{pmatrix} \end{matrix}$$

Fig. 3. An example coverage matrix

The COV and PART metric values for our example coverage matrix (see Figure 3) can be seen in Table 1. The metrics were calculated for four test groups and for the whole matrix, the code group always consisted of all procedures. This example exhibits several cases where either COV is higher than PART, or the other way around. Although in theory there is no direct relationship between these two metrics, with realistic coverage matrices we observed similarities for our subject system as described later.

Table 1. Metrics of the example coverage matrix in Figure 3

Test group	COV	PART	Partitions
$\{t_1, t_2\}$	0.33	0.60	$\{p_1\}, \{p_2\}, \{p_3, p_4, p_5, p_6\}$
$\{t_3, t_4\}$	1.00	0.33	$\{p_1, p_2, p_3, p_4, p_6\}, \{p_5\}$
$\{t_5, t_6\}$	0.50	0.73	$\{p_1, p_2\}, \{p_3\}, \{p_4, p_5, p_6\}$
$\{t_7, t_8\}$	1.00	0.80	$\{p_1, p_2\}, \{p_3, p_4\}, \{p_5, p_6\}$
$\{t_1 \dots t_8\}$	1.00	1.00	$\{p_1\}, \{p_2\}, \{p_3\}, \{p_4\}, \{p_5\}, \{p_6\}$

4 Data Extraction from the WebKit System

The first two steps in the adapted TAIME method is to set the granularity and determine the test and code groups. In this section we present the extraction process of these groups from the WebKit system [20], a layout engine that renders web pages in some major browsers such as Safari. WebKit contains about 2 million lines of C++ code (this amounts to about 86% of the whole source code) and it has a relatively big collection of regression tests, which helps developers keep code quality at a high level.

We chose to work on the granularity of procedures (C++ functions and methods) due to the size and complexity of the system, but our approach is generalizable to finer levels as well. (In [16], we applied the TAIME method on both statement and procedure level.)

The next step was to determine test and code groups. WebKit tests are maintained in the `LayoutTests` directory. It contains test inputs and expected results to test whether the content of a web page is displayed correctly (they could be seen as a form of system level functional tests). In addition to strictly testing the layout, it contains, for example, http protocol tests and tests for JavaScript code as well. Tests are divided into thematic subdirectories to group tests of particular topics. These topics are the different features of the WebKit system and this separation made by the WebKit developers gave the base of the decomposition of the program into functional units. Based on this separation, the decomposition of the test suite into test groups was a natural choice.

At the separation of the program code, it is important to note that the associated code groups were not selected based on code coverage information, rather on expert opinion about the logical connections between test and code. Code groups could be determined automatically based on the test group coverage, but in that case the coverage of code groups would be always 100%. Our choice

to involve experts helps to highlight the differences between the plans of the developers and the actual state that the test suite provides.

Table 2. Functional units in WebKit with associated test and code group sizes

Functional unit	Number of Tests	Number of Procedures
WebKit	27013	84142
canvas	1073	400
css	2956	1899
dom	3749	3761
editing	1358	1690
html5lib	2118	4176
http	1778	454
js	8313	8113
svg	1955	6336
tables	1340	2035

The resulting functional units can be observed in Table 2. We identified a total number of 84142 procedures in WebKit, and the full test suite contains 27013 tests, as shown in the first row of the table.⁴ The rest of the table lists selected functional units and the statistics of the associated test and code groups, which are detailed in the following subsections. We asked 3 experienced WebKit developers – who have been developing WebKit at our department for about 7 years – to determine functional units of the WebKit system.

4.1 Test Groups

As the experts suggested, the test groups were determined based on how WebKit developers categorize tests. The `LayoutTests` directory contains a separate directory for each functionality in the code that is intended to be tested. These directories (and their sub-directories) contain the test files. A test case in WebKit is a test file processed by the WebKit engine as a main test file, which can include other files as resources. The distribution of the number of tests in these main test directories can be seen in Figure 4. There are several small groups starting from the 12th largest directory in the middle of the diagram. We could either put them into individual groups, which would make the analysis process inefficient (with too many groups but less gain), or we could treat them as one functional unit breaking the logical structure of the groups. Finally we decided to exclude these tests, so about 91% of tests were included in our experiments.

⁴ Actually, there are more tests than this, but we included only those tests that are executed by the test scripts. We used revision `r91555` of the Qt (version 4.7.4) port of WebKit called `QtWebKit` on the `x86_64` Linux platform. Also, only C++ code was measured.

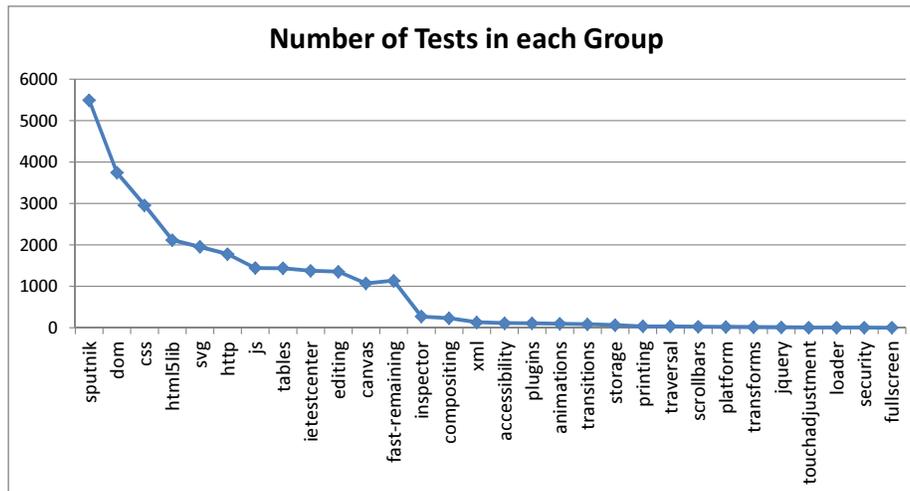


Fig. 4. Number of tests in various test directories in WebKit

However, we found some inconsistencies in the test directory structure, so made further adjustments to arrive at the final set of groups. First, the **fast** directory contains a selection of test cases for various topics to provide quick testing capability for WebKit. This directory is constantly growing, and takes about 27% of all the tests in WebKit. Unfortunately, it cannot be treated as a separate functional unit as it contains tests that logically belong to different functions. Hence we associated each **fast/*** subdirectory to some of the other groups and a small, heterogeneous part is left uncategorized, marked as **fast-remaining** (starting with this point we cut the remaining test groups, see in the middle of Figure 4). Second, there exist three separate test groups for testing JavaScript code. This language plays a key role in rendering web pages, and the project adopted two external JavaScript test suites called **sputnik** and **ietestcenter**. In addition, the WebKit project has its own test cases for JavaScript in the **fast/js** directory. We used the union of these three sources to create a common JavaScript category called **js** in our tables.

4.2 Code Groups

Once the features were formed, we asked the experts to assign source code directories/files to functionalities based on their expertise, but without letting them access the coverage data of the tests. This task required two person-day effort from the experts and the result of this step was a list of directories and files for each functional unit. The path of each procedure in the code was matched against this list, and the matching functional unit(s) were assigned to the procedures. We chose this approach to avoid investigating more than 84000 procedures one by one and so we determined path prefixes and labeled all procedures automatically. However, this method resulted in a file level granularity in our dataset, and increased the overlap between functional units. Eventually, out of 84142 proce-

dures we associated about 28800 with one of the functional units, during code browsing only procedures strictly implementing the functionality of the units were selected. The omitted procedures usually included general functionality that could not be associated with any of the functional units unambiguously. At the end of this process, all experts accepted the proposed test and code groups.

We used several tools during this process. For the identification of all procedures in WebKit (including non-covered ones) the static C/C++ analyzer of the Columbus framework [8] was used. Furthermore, we used the SoDA library [15, 17] for trace analysis and metrics calculation after running the tests on the instrumented version of WebKit.

At the end of this process we determined the groups, executed the test suite and measured the coverage using the given metrics.

5 The Internals of a Test Suite as seen through Metrics

In this section, we present the last, analyze step of the proposed TAIME approach by first presenting our measurement results using the introduced metrics on WebKit, then at the end of the section we evaluate the system based on the result and determine possible enhancement opportunities.

5.1 Coverage metrics

In the first set of experiments, we investigated the code coverage metrics for the different functional units to find out if a test group produces higher overall coverage on the associated code group than the others. In particular, for each *test group* - *code group* pair we calculated the percentage of code that the corresponding test cases cover, which we summarize in Table 3. Rows in the table represent test groups while code groups are in the columns, and each cell contains the associated coverage metric COV. For example, we can observe that tests from the `canvas` functional unit cover 26% of the procedures of unit `css`.

The first row and column of this table show data computed for the whole WebKit system without separating the code and tests of the functional units. The coverage ratio of the whole system is 53%, which means that about 47% of the procedures is never executed by any of the test cases (note that for computing system level coverage all tests are used including the 9% tests omitted from individual groups). This is clearly a possible area for improvement globally, but let us not forget that in realistic situations complete coverage is never aimed due to a number of difficulties such as unreachable code or exception handling routines.

We can interpret the results of Table 3 in two ways: by comparing results in a row or in a column. In the former case, we can observe which code groups are best covered by a specific test group, and in the latter the best matching test group can be associated with a code group. Using the latter it can be observed which tests are in fact useful for testing a particular piece of code. We used a graphical notation in the matrix of two parallel horizontal lines for rows and two

Table 3. Coverage metric values and heat-map for test groups in functional units

<i>Test</i> \ <i>Code</i>	WebKit	canvas	css	dom	editing	html5lib	http	js	svg	tables
WebKit	.53	.56	.61	.59	.67	.67	.65	.47	.50	.72
canvas	.16	.46	.26	.24	.07	.19	.00	.30	.03	.45
css	.24	.13	.51	.33	.25	.36	.00	.32	.11	.62
dom	.33	.17	.38	.52	.34	.51	.12	.35	.08	.57
editing	.23	.02	.31	.38	.66	.35	.01	.31	.06	.59
html5lib	.29	.12	.37	.43	.46	.52	.13	.34	.20	.63
http	.33	.23	.41	.42	.25	.41	.65	.39	.14	.57
js	.33	.16	.37	.47	.51	.44	.15	.44	.11	.63
svg	.26	.01	.38	.35	.17	.21	.01	.31	.50	.56
tables	.18	.00	.29	.30	.16	.31	.00	.26	.02	.62

vertical lines for columns, respectively, to indicate the maximal values. In order to have a more global picture of the results, we also present them in a graphical way in the form of a ‘heat-map’: the intensity of the red background of a cell is proportional to the ratio of the cell value and the column maximum, *i.e.* the column maxima are the brightest.

The most important to observe is that, except for **tables**, each code group is best covered by its own test group. This is an indicator of a good separation of the tests associated with the functional units. In the other dimension, there are more cases when a particular test group achieves higher coverage on a foreign code group, but the value in the diagonal is also very high in all cases. The dominance of the diagonal is clearly visible, however there are some notable cases where further investigation was necessary. For example code group **http**, which is very specific to its test group, and **tables**, which does not have a clear distinction between the test groups. We are going to discuss these cases in more detail at the end of this section. Another observation is that the coverage values in the main diagonal are close to the overall coverage of 53%.

5.2 Partition metrics

The partition metrics for the functional unit pairs showed surprising results. As can be seen in Table 4 – that also shows the corresponding heat-map information –, the PART metric values basically indicate the same relationship between the test and procedure groups as the coverage metrics. In fact, the Pearson correlation of the two matrices of metrics (represented as vectors) is 0.98.

Table 4. Partition metric values and heat-map for test groups in functional units

<i>Test</i> \ <i>Code</i>	<i>WebKit</i>	<i>canvas</i>	<i>css</i>	<i>dom</i>	<i>editing</i>	<i>html5lib</i>	<i>http</i>	<i>js</i>	<i>svg</i>	<i>tables</i>
WebKit	.77	.80	.84	.83	.88	.88	.80	.71	.74	.92
canvas	.29	.69	.45	.41	.13	.33	.00	.50	.06	.65
css	.39	.23	.72	.53	.37	.57	.00	.51	.15	.81
dom	.55	.30	.62	.76	.56	.76	.22	.57	.16	.79
editing	.39	.03	.51	.61	.87	.56	.00	.51	.11	.81
html5lib	.48	.22	.60	.67	.70	.76	.23	.55	.30	.84
http	.54	.40	.64	.66	.44	.64	.79	.62	.26	.80
js	.53	.28	.59	.71	.73	.64	.26	.68	.18	.84
svg	.43	.00	.55	.56	.31	.33	.00	.50	.74	.79
tables	.31	.00	.46	.50	.21	.51	.00	.45	.04	.83

In general, the coverage and the partition metric values do not necessarily need to be correlated, which is illustrated also in our example from Figure 3 and Table 1. However, certain general observations can be made as follows. When the number of tests and procedures are over a certain limit, it is more likely that a unit with high coverage will include different tests and produce high partition metric as well, but it can happen that a unit with high coverage consists of test cases with high but very similar coverage values, in which case the partition metric will be worse. On the other hand, if the coverage is low, it is unlikely that the partition metric will be high because non-covered items do not get separated into distinct partitions. In earlier work [19], we used partition metrics in the context of test reduction, and there the difference between partitioning and coverage was distinguishable. We explain this also by the fact that the reduced test sets consisted of much less test cases compared to the test groups of functional units from the present work.

5.3 Distribution of procedures

In a theoretical case of an ideal separation of functional units (code and test groups) the above test suite metric tables would contain high values only in their diagonal cells. Our experiments show that this does not always hold for WebKit test groups. We identified two possible reasons for this phenomenon: either test cases are not concentrated to their own functional unit, or procedures are highly overlapping between functional units. We calculated the number of common procedures in all pairings of functional units and it turned out that in general the overlap is small: *css* is slightly overlapped with four other groups;

and the `html5lib` group contains most of the `canvas` and is overlapped with `tables`. From the small amount of the total 567 common procedures we reason that test cases cause the phenomenon. Although the tests are aiming well defined features of WebKit, technically they are high level (system level) test cases executed through a special, but fully functional browser implementation. WebKit developers acknowledged that tests go through functional units, for example a single test case for testing an `svg` animation property will cover also `css` for style information, `js` for controlling timing and querying attributes, which also implies the coverage of some `dom` code as well.

5.4 Characteristics of the WebKit test suite

We summarize our observations on functional units that we found during the analysis of metrics in the previous sections.

Regarding the special cases, we identified two extremes in the system. On the one hand, there are functional units, where code groups are not really exercised by other test groups, while their test groups cover other code groups. One of these groups is `http`. By investigating this group, we found that most functionalities – *i.e.* assembling requests, sending data, etc. – are covered by the `http` test group, while other test groups usually use basic communication and small number of requests. The number of test cases in these groups could probably be reduced without losing coverage, but only with taking care of tests which cover the related code of the group.

On the other hand, there are groups like the `tables` group which is some kind of an outlier in the sense that this code group is heavily covered by all of the test groups. The reason for this is that it is hard to separate this group from the code implementing the so-called box model in WebKit, which is an integral part of the rendering engine. Thus, almost anything that tests web pages will use the implementation of the box model, which is mostly included in the `table` code group. Hence, the coverage data is highly overlapping. Although its coverage is the highest one `tables` maintains good PART metrics. Highly covered by other test groups, `tables` should be the last one to be optimized among the test groups. The number of test cases in this group could probably be reduced due to the high coverage by other modules, however, more specific tests could be used to improve coverage.

According to our analysis there is room for improving the coverage of all code groups as they are around the overall coverage rate of 53%. Another general comment is that component level testing (unit testing) is usually more effective than system level testing (as is the case with WebKit) when higher code coverage is aimed. For example, error handling code is very hard to be exercised during system level testing, while at component level such code can be more easily tested. Thus, in a long term, introducing a unit test framework and adding real component tests would be beneficial to attain higher coverage.

In summary, the experience that we gathered from analyzing the data of the adapted TAIME method appoints two main ways to improve testing. The proposed metrics can either be used to provide guidance for the developers during

the maintenance of the tests, or to focus test reduction on enhancing specific capabilities of the test suite, e.g. fault detection and localization [19].

6 Related Work

Although there exist many different criteria for test assessment [22], the main approach to assess the adequacy of testing has long been the fault detection capability of test processes in general and test suites in particular. Code coverage measurement is often used as a predictor to the fault detection capability of test suites, but other approaches have been proposed as well, such as the output uniqueness criteria defined by Alshahwan and Harman [1]. Code coverage is also a traditional base for white-box test design techniques due to the presumed relationship to defect detection capability, but some studies showed that this correlation is not always present, inversely correlated to reliability [18], or at least not evident [11, 10].

There have been sporadic attempts to define metrics that can be used to assess the quality of test suites, but this area is much less developed than that of general software quality. Athanasiou *et. al.* [2] gave an overview on the state of the art. They concluded that although some aspects of test quality had been addressed, basically it remained an open challenge. They provided a model for test quality based on the software code maintainability model, however, their approach can only be applied on tests implemented on some programming language.

Researchers started to move towards test oriented metrics only recently, which strengthens our motives to work towards a more systematic evaluation method for testing. Gomez *et. al.* provide a comprehensive survey of measures used in software engineering [9]. They found that a large proportion of metrics are related to source code, and only a small fraction is directed towards testing. Chernak [6] also stresses the importance of test suite evaluation as a basis for improving the test process. The main message of the paper is that objective measures should be defined and built into the testing process to improve the overall quality of testing, but the employed measures in this work are also defect-based.

Code coverage based test selection and prioritization techniques are also related to our work as we share the same or similar notions. An overview of regression test selection techniques has been presented by Rothermel and Harrold, who introduced a framework to evaluate the different techniques [14]. Elbaum *et al.* [7] conducted a set of empirical studies and found that fine-grained (statement level) prioritization techniques outperform coarse-grained (function level) ones, but the latter produce only marginally worse results in most cases, and a small decrease in effectiveness can be more than offset by their substantially lower cost. A survey for further reading in this area has been presented by Yoo *et al.* [21].

7 Conclusions

Systematic evaluation of test suites to support various evolution activities is largely an unexplored area. This paper provided a step towards establishing a systematic and objective evaluation method and measurement model for (regression) test suites of evolving software systems, which is based on analyzing the code coverage structures among test and code elements. One direction for future work will be to continue working towards a more complete test assessment model by incorporating other metrics, possibly from other sources like past defect data.

We believe that our method for test evaluation is general enough and is not limited to the application we used it in. Nevertheless, we plan to verify it by involving more systems having different properties, and in different test suite evolution scenarios such as metrics-driven white-box test case design.

Regarding our subject WebKit, our observations are based mostly on the introduced metrics and we did not take into account the feasibility of the proposed optimization possibilities of the tests. We plan to involve more test suite metrics and investigate our actual suggestions in the future.

References

1. Alshahwan, N., Harman, M.: Coverage and fault detection of the output-uniqueness test selection criteria. In: Proceedings of the 2014 International Symposium on Software Testing and Analysis. pp. 181–192. ACM (2014)
2. Athanasiou, D., Nugroho, A., Visser, J., Zaidman, A.: Test code quality and its relation to issue handling performance. *Software Engineering, IEEE Transactions on* 40(11), 1100–1125 (Nov 2014)
3. Basili, V., Caldiera, G., Rombach, H.D.: Goal question metric approach. In: *Encyclopedia of Software Engineering*, pp. 528–532. John Wiley & Sons, Inc. (1994)
4. Beck, K. (ed.): *Test Driven Development: By Example*. Addison-Wesley Professional (2002)
5. Bertolino, A.: Software testing research: Achievements, challenges, dreams. In: *2007 Future of Software Engineering*. pp. 85–103. IEEE Computer Society (2007)
6. Chernak, Y.: Validating and improving test-case effectiveness. *IEEE Softw.* 18(1), 81–86 (Jan 2001)
7. Elbaum, S., Malishevsky, A.G., Rothermel, G.: Test case prioritization: A family of empirical studies. *IEEE Trans. Softw. Eng.* 28(2), 159–182 (Feb 2002)
8. Ferenc, R., Beszédes, Á., Tarkiaenen, M., Gyimóthy, T.: Columbus - reverse engineering tool and schema for C++. In: *Proceedings of the 6th International Conference on Software Maintenance (ICSM 2002)*. pp. 172–181. IEEE Computer Society, Montreal, Canada (Oct 2002)
9. Gómez, O., Oktaba, H., Piattini, M., García, F.: A systematic review measurement in software engineering: State-of-the-art in measures. In: *Software and Data Technologies, Communications in Computer and Information Science*, vol. 10, pp. 165–176. Springer (2008)
10. Inozemtseva, L., Holmes, R.: Coverage is not strongly correlated with test suite effectiveness. In: *Proceedings of the 36th International Conference on Software Engineering (ICSE 2014)*. pp. 435–445. ACM (2014)

11. Namin, A.S., Andrews, J.H.: The influence of size and coverage on test suite effectiveness. In: Proceedings of the Eighteenth International Symposium on Software Testing and Analysis. pp. 57–68. ACM (2009)
12. Pinto, L.S., Sinha, S., Orso, A.: Understanding myths and realities of test-suite evolution. In: Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering. pp. 33:1–33:11. ACM (2012)
13. Pinto, L.S., Sinha, S., Orso, A.: Understanding myths and realities of test-suite evolution. In: Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering. pp. 33:1–33:11. FSE '12, ACM, New York, NY, USA (2012)
14. Rothermel, G., Harrold, M.J.: Analyzing regression test selection techniques. *IEEE Trans. Softw. Eng.* 22(8), 529–551 (1996)
15. SoDA library. <http://soda.sed.hu>, last visited: 2015-08-20
16. Tengeri, D., Beszédes, Á., Gergely, T., Vidács, L., Havas, D., Gyimóthy, T.: Beyond code coverage - an approach for test suite assessment and improvement. In: Proceedings of the 8th IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW'15); 10th Testing: Academic and Industrial Conference - Practice and Research Techniques (TAIC PART'15). pp. 1–7 (Apr 2015)
17. Tengeri, D., Beszédes, Á., Havas, D., Gyimóthy, T.: Toolset and program repository for code coverage-based test suite analysis and manipulation. In: Proceedings of the 14th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM'14). pp. 47–52 (Sep 2014)
18. Veevers, A., Marshall, A.C.: A relationship between software coverage metrics and reliability. *Software Testing, Verification and Reliability* 4(1), 3–8 (1994), <http://dx.doi.org/10.1002/stvr.4370040103>
19. Vidács, L., Beszédes, Á., Tengeri, D., Siket, I., Gyimóthy, T.: Test suite reduction for fault detection and localization: A combined approach. In: Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), 2014 Software Evolution Week - IEEE Conference on. pp. 204–213 (Feb 2014)
20. The WebKit open source project. <http://www.webkit.org/>, last visited: 2015-08-20
21. Yoo, S., Harman, M.: Regression testing minimization, selection and prioritization: a survey. *Software Testing, Verification and Reliability* 22(2), 67–120 (2012)
22. Zhu, H., Hall, P.A.V., May, J.H.R.: Software unit test coverage and adequacy. *ACM Comput. Surv.* 29(4), 366–427 (Dec 1997)

Accounting Testing in Software Cost Estimation: A Case Study of the Current Practice and Impacts

Jurka Rahikkala^{1,2}, Sami Hyrynsalmi², Ville Leppänen²

¹ Vaadin Ltd, Turku, Finland

jurka.rahikkala@vaadin.com

² Department of Information Technology, University of Turku, Finland

{sthyry, ville.leppanen}@utu.fi

Abstract. Testing has become an integral part of most software projects. It accounts for even as high a share as 40% of the overall work effort. At the same time software projects are systematically exceeding their effort and schedule forecasts. Regardless of the overruns and big share of testing, there is very little advice for estimating testing activities. This case study research assesses the current practice of estimating testing activities, and the impact of these practices on estimation and project success. Based on the interviews with 11 stakeholders involved in two case projects and examination of project documentation, this study shows that companies easily deviate from their standard procedures, when estimating testing. This may even lead to severe estimation errors. The deviations can be explained by negative attitudes towards testing. Furthermore, this study shows that the extant literature has sparsely addressed estimation of software testing.

Keywords: Software Cost Estimation, Software testing, Project Management

1 Introduction

Software cost estimation (SCE), or effort estimation, is an art which is not well handled by the software industry (see e.g. Rahikkala et al., 2014). The famous *Chaos Reports* by Standish Group International have for decades claimed that nearly two thirds of software products either failed to meet their budget and timetable goals or they were never delivered (The CHAOS Manifesto, 2013). While these numbers have been heavily criticized (see Laurenz Eveleens and Verhoef, 2010), high failure percentages have been shown also by other research and consultancy institutes (c.f. Moløkken and Jørgensen, 2003; Mieritz, 2012). Nevertheless, these numbers clearly show the dilemma of software cost and effort estimations: they often fail.

In this study, our topic is an increasingly important aspect of SCE: estimating the effort of software testing activities. The software testing activities have been found to take from 20%–25% (Haberl et al., 2012; Lee et al., 2013; Buenen and Teje, 2014) to even 40% or more (Ng et al., 2004) of the total cost of the project. However, there is

still a lack of knowledge of SCE methods as well as proper tools and practices for software testing estimation. For example, from the studies included into Jørgensen and Shepperd's (2007) systematic literature review on software cost estimation, only a handful seems to discuss estimation of software testing activities.

Thus, there is a gap in extant literature on the effect of software testing effort estimation. The objective of this paper is to 1) gain in-depth knowledge of the current practice for accounting testing in SCE, and 2) understand the impact of the used practices on SCE and project success.

To answer the research questions, a qualitative research approach is used. We study software cost estimation practices in two projects in two case companies. Based on the study of 11 interviews and 17 project related documents, this paper contributes to the scientific literature by reporting on the current practice of estimating testing effort, and the impact of use of these practices on estimation and project success. Understanding the role of testing effort estimation in software projects better may help project managers, other software professionals and researchers to pay more attention on testing and related estimation.

The rest of this paper is structured as follows. Section 2 briefly presents the background of software cost estimation studies. It is followed by a description of research design and the case study subjects. The fourth section presents the results and Section 5 discusses the practical and scientific implications, and addresses the limitations of the study as well as the future research directions. The final section concludes the study.

2 Background

Software cost and effort estimation has a long tradition in the field of computing disciplines. For example, Farr and Nanus (1964) and Nelson (1966) addressed the cost estimation of programming five decades ago. Since then a multitude of software cost estimation methods and models have been presented (c.f. Boehm et al., 2000; Briand and Wiczorek, 2002; Jørgensen and Shepperd, 2007). However, as discussed in the introduction, the software industry has an infamous history with the success of software cost and effort estimations. Despite the decades of research, projects often run over their budgets and timetables (Moløkken and Jørgensen, 2003).

Software testing is showed to be hard to estimate and predict. In the survey of Australian companies by Ng et al. (2004), only 11 companies out of the 65 studied were able to meet their testing budget estimates. Most of the companies spent more than 1.5 times the resources they had estimated. Furthermore, three of the studied companies estimated the cost of testing twice as high as was the actual cost. At the same time, most of the companies reported using approximately 10-30% of their initial budget to the testing activities.

The extant literature has classified presented estimation methods in several ways. For example, Boehm, Abts, and Chulani (2000) divided the existing methods into six different categories: Model-based, Expertise-based, Learning-oriented, Dynamics - based, Regression-based and Composite techniques. However, for the sake of simplicity, we consider only two major types of software cost estimation methods: Model-based and Non-model-based (Briand and Wieczorek, 2002). The first category, in general, consists of methods that have a model, a modelling method, and an application method. A classical example of these kinds of software cost estimation methods is COCOMO by Boehm (1981). The methods of the second category consist of one or more estimation techniques. These methods do not involve any models, only estimations. For example, bottom-up expert estimations belong to this category.

Some methods for software test effort estimation have been presented. For example, Calzolari, Tonella and Antoniol (1998) presented a dynamic model for forecasting the effort of testing and maintenance. The model is based on a view that testing is a similar activity to predating a prey (i.e. software bug) in nature. Engel and Last (2007) have also presented a model for estimating testing effort based on fuzzy logic. In contrast to modelling-focused techniques, Benton (2009) recently presented the IARM-estimation method for software cost and effort. The model is remarkably simpler than presented formal models and it emphasizes expert evaluation by team members; however, there is a lack of validation and verification of the proposed model.

To summarize, software testing activities seem to be rather hard to estimate. While there is a lack of studies on software testing estimations, Ng et al. (2004) showed that most of surveyed Australian companies do not hit their estimates. Furthermore, there is a rather wide understanding on the size of software testing of the whole software development project. While some studies suggest the use of only one-fourth of the budget (e.g., Haberl et al., 2012; Lee et al., 2013; Buenen and Teje, 2014), there have been claims of even half of the budget (see e.g. Ammann and Offutt, 2001). Nevertheless, in our literature review, we sparsely found any studies focusing on developing or validating software cost estimation techniques for testing. This is a noteworthy disparity between the cost caused by testing activities and the academic interest towards the topic.

3 Research methodology

In the following, we will first present the research approach and design of this study. It is followed by a description of case study subjects.

3.1 Research design

This study is based on a qualitative research approach (Cresswell, 2003). We use a case study research strategy and interviews as the main tools of inquiry. The qualita-

tive research approach was selected to allow us to get an in-depth understanding about the phenomenon under the study lens. The case study research strategy was used as the researchers have no control over the study subject (Yin, 2003). As Patton (2001) states, the case studies are able to shed light on phenomena occurring in a real-life context. This study is *exploratory* of type, finding out what is happening, seeking new ideas and generating hypotheses and ideas for new research (Robson, 2002). The research uses a multiple case study design following a replication logic (Yin, 2003). The case study organisations were selected based on the impression of the SCE maturity of the organisation gained during the initial discussions with the organisation representatives. The unit of analysis is a single software cost estimate. The study is focused on the experiences gained during the preparation of the cost estimate.

This study about estimation of software testing consists of two case companies that we call as *Small Global* and *Large Multinational*. The companies wished to remain anonymous in this study. From both companies, we selected one software implementation project that has either ended or is near its ending. The first project was developed for a customer, thus the estimate was used for pricing the project, in addition to other planning purposes. The other is a software tool development project, which is aimed for a mass market. In this project, the estimates were especially used for estimating the release date of the product.

For the selected case study projects, we interviewed different stakeholders involved in the projects. The interviewees' roles varied from developers and testers to project managers and senior executives. In addition to the interviews, we collected various documents related to the project. For example, we reviewed project plans, design documents and minutes of weekly meetings related to the projects.

We created an interview protocol consisting of several questions related to software cost estimation and testing. The protocol was created following the guidelines by Runeson et al. (2012). The interviews were conducted as semi-structured (Robson, 2002). In the interviews, while following the protocol, new ideas were allowed to be brought up and discussed with the interviewee. The interviews were conducted by two researchers where one acted as the main interviewer. An interview session was approximately one hour in length and the discussion was recorded. The recordings were transcribed and sent to the interviewees for review. All case subjects participated in the study voluntarily and anonymously, and the collected data was treated as confidential.

For the analysis of data, we used nVivo 10. All transcribed interviews, notes done during the interviews, in addition to the auxiliary materials, were imported into the software. The analysis was conducted in a series of steps (Robson, 2002). First the texts were coded by the researchers, whereafter iteration followed, until the conclusions were reached.

3.2 Case subjects

The Small Global is a software producing firm of about 100 persons, headquartered in Finland. The company's line of business consists of selling consultancy and support services in addition to software products to businesses. The company is global; it has customers and offices in several countries. The selected project, referred to as *Developer Tool*, is a typical software product development project in the company. In the project, the aim was to produce a visual design tool for developing applications. The project followed a Waterfall-style software engineering method: it was strictly planned up-front but the actual development work was divided into sprints. The interviews conducted in the company are shown in Table 1.

The Developer Tool project started with a prototype where technical challenges and possible development stacks were studied. After the prototype project, a project aiming at the release of version 1.0 was planned. The management named a project manager and a product owner for the product. The product owner crafted a design document for the product, and based on that document, the project management created a project plan with time estimates. Initially, the project was estimated to take three months with a team of four people.

The project missed its first deadline, a beta version release, approximately after the half-point of the estimated duration of the project. First, the project team narrowed the content plan of the product in order to hit the planned release date. Later, the initial content was returned to the plan and deadlines were postponed in the future. When the problems were noted, some developers were changed and new ones were introduced. Currently, the project is expected to finish after approximately nine months of development with the team of approximately four persons.

Table 1. Interviews done in this research

Interview code	The role of interviewee
<i>Small Global</i>	
Interview V ₁	Key informant interview (Product owner)
Interview V ₂	Senior manager of <i>Developer Tool</i>
Interview V ₃	Product owner of <i>Developer Tool</i>
Interview V ₄	Senior manager of <i>Developer Tool</i>
Interview V ₅	Project manager of <i>Developer Tool</i>
<i>Large Multinational</i>	
Interview T ₁	Key informant interview (Project manager)
Interview T ₂	Project manager of <i>Operational Control System</i>
Interview T ₃	Senior manager of <i>Operational Control System</i>
Interview T ₄	Software test manager of <i>Operational Control System</i>
Interview T ₅	Requirements expert of <i>Operational Control System</i>
Interview T ₆	Senior developer of <i>Operational Control System</i>

The other case study subject comes from The Large Multinational, which is a large multinational company also headquartered in Finland. The Large Multinational produces software and consultancy for a wide area of business sectors. The selected project, referred to as *Operational Control System*, is a typical software development project for the company. The resulted software is a business intelligence reporting system for following certain control activities. The software was ordered by a long-term customer of the company. The interviews conducted in the company are shown in Table 1.

Also this project followed a Waterfall-like software development process: the requirements were elicited before the features of the product were agreed upon with the customer. The product was estimated only after the specification documentation was ready and accepted by the customer. The estimation was done by the developers and testers themselves. The Large Multinational uses a structured sheet for estimations that the workers filled individually by themselves. The project manager prepared the final estimates by using the expert estimates as the base.

The Operational Control System project was planned according to certain preconditions: the customer had a fixed budget and limited time for development. Only after both the customer and the software vendor had agreed upon the content with a limited set of unknown features, the development started. The development work continued rather straightforwardly from design and implementation to testing and delivery. The project lasted 10 months; the size of the project was approximately 30 man-months, of which 25% was used for testing and quality assurance. While there were major changes asked by the customer in the midway of the project, it hit the targets by keeping the budget and the timetable. In certain areas, there were estimation mismatches: in one software testing area, there was a significant overrun (+76% of initial estimate); however, in the implementation of a certain feature, an expert level developer was able to underbid the estimate (-77%) that was created with the presumption of a general level developer.

4 Results

This section presents the findings identified during the analysis of data, as described in the research methodology chapter. The findings are grouped into the following five categories:

1. General
2. Estimation practices
3. Testing plan
4. Attitudes
5. Estimation challenges

4.1 General

In both projects testing was strongly related to effort and schedule overruns. In the Operational Control System project the overrun of the data import testing was nearly 80%, although the overall project managed to keep the original budgets. In the Developer Tool project, omitted testing tasks resulted in significant effort and schedule overruns, being roughly 100% in July 2015. Regardless of the significant testing related overruns, the management in both projects report that the actual amount of work would probably not have been affected by more accurate estimates. In other words, the actual testing work was necessary, although bigger than anticipated.

The overrun of the testing effort in the Operational Control System lead to discussions with the customer, and likely to decreased customer satisfaction. In the Developer Tool, the project has received a late project status because of the overruns. This has lead to some negative late project symptoms, like decreased development team motivation, increased number of status meetings, frequent re-estimation and more discussions about the project scope (McConnell, 2006).

Both organisations have a long experience of software projects and related testing. The unit responsible for the Operational Control System has dedicated testing engineers and teams, while in the Developer Tool project the testing was conducted partially by dedicated testing engineers, partially by the development team. In both cases customer representatives or pilot users participated in user testing. Following the Testing Maturity Model (TMM) presented by Burstein et al. (1998), the testing maturity was on level 3 and level 2 in Operational Control System and Developer Tool, respectively, in the scale from 1 (low maturity) to 5 (high maturity).

4.2 Estimation practices

There were standard procedures for the estimation of the testing effort in both of the companies. The procedures required that the project plan, functional specification and testing plan must be ready before the effort estimation. Expert estimation was used for estimating testing tasks in both projects, i.e. experts estimated the effort of the testing tasks based on their professional competence. In the Operational Control System project a spreadsheet based work breakdown structure was used for preparing the final estimate, and the effort was also compared with the historical project data. There was also a guideline based on the historical data that the testing effort varies between 50% and 200% of implementation effort, depending on the application area to be tested. The expert estimation of testing tasks was conducted by the actual testing engineer in the Operational Control System project and by the product owner in the Developer Tool, while feature implementation tasks were estimated by the actual software engineers in both projects. In the Operational Control System, the actual testing engineers

were known by the time of estimation, which was not the case in the Developer Tool project. Finally the estimates were reviewed in the project steering groups.

The fact that the actual testing engineer estimated the testing tasks was seen to have a positive impact on the estimation results in the Operational Control System project, as well as the experience of the estimator. The Operational Control System team members report also that knowing the actual persons who will be doing the testing helps in preparing estimates. That is, the amount of work hours needed depends on the persons doing the work.

4.3 Testing Plan

Regardless of the requirement by the standard estimation procedures, neither of the projects had a testing plan ready, when the implementation phase of the project was started. In other words, the exact scope of the testing was not defined in detail before testing was estimated. In the Operational Control System project the testing effort was accounted in the project plan, and the testing manager reports that there was a shared vision of the testing based on the previous projects. In the Developer Tool project only automated regression testing was accounted in the project plan, but user testing and manual testing were omitted. In both projects the testing plan was created in a later phase of the project. The reason for not finalizing the testing plan before starting the implementation was a hurry to proceed with the implementation, not to save money.

Interviewees in the Developer Tool report that there was a significant difference in the expectations of the project results between the development team and the management. The management expected a finalized, user tested product, while the development team's expectation was more like a regression tested proof-of-concept, where the user testing would have been postponed until the next version of the product. The overruns in the Developer Tool project are specifically related to this difference in expectations. Although the testing activities are seen as necessary, a senior manager in the Developer Tool states that a testing plan would have helped to discover testing related problems earlier and to plan testing activities better. A proof-of-concept project was done prior to the actual Developer Tool project to test all key technological assumptions, but that did not cover testing. Retrospectively a senior manager speculates that testing should have been part of the proof-of-concept to avoid testing related surprises.

In the Operational Control System the project manager reported that the first estimate for testing did not fit in the project schedule, and it was discovered that the estimate was prepared in five minutes. The project manager had also challenged the high effort for testing activities, being not able to understand the basis for the estimate. Furthermore, the project and testing managers report that the difficulties in testing

were related to e.g. unavailability of test data and higher complexity of the data import logic than expected.

4.4 Attitudes

In the Operational Control System, the testing manager describes that testing is generally considered as of low importance, and that the competence and historical data is not valued as it is valued in the implementation. This feeling contradicts with the general comments where professional competence and use of historical data are strongly connected to estimation success. The Operational Control System project owner emphasizes that the testing effort must be on a reasonable level considering the application area and that it must add value to the project. The project owner continues that Large Multinational has invested significant amount of time and money for improving testing capabilities within the past two years. This communicates about the experienced importance of testing. However, according to the product owner, changing established practices will take time. Based on the interviews, changing attitudes seem to take time, like changing any other capabilities. In both projects the technical staff considered especially automated regression testing to have a high importance. In the Developer Tool project, a senior manager describes that developers' attitudes towards manual testing and user experience testing are negative.

Other findings in this research support the feeling that testing is not considered equally important as implementation. For example, the testing plans were not ready when the project was estimated or testing started, and in one project the actual testers did not estimate the testing work or the actual testers were not known at the time of estimation. The Developer Tool projects' product owner reports that testing related tasks are first omitted if the schedule is tight and something needs to be dropped out from the scope. This lower level of importance may prevent testing as a function from evolving, including estimation of testing. The testing manager in the Operational Control System pointed out that the attitudes towards testing are not encouraging even in universities: *"There was only one course of software testing out of two hundred offered"*. Additionally, the testing manager commented that testing is seen as a task for those who are not good enough for ordinary programming.

4.5 Estimation challenges

In the Operational Control System project, the testing manager considered estimation of testing as more difficult than estimating implementation. The primary reason for this is the high number of dependencies: problems with data or specifications, or high number of bugs will reflect to testing. For example, if the test data is delayed by one week, the testing team is still there, but not doing what they were planned to do. The number of found bugs was relatively high in the Operational Control System, which

cumulated extra work because of manual testing practices. Also the Developer Tool project manager emphasised the role of external parties in estimation. For example, the unavailability of external testing engineers or users for user testing may delay the project. Other interviewees could not make a difference in difficulty, but a senior manager in the Developer Tool pointed out that there is generally less experience of estimating testing.

Automated testing was widely seen to make regression testing more predictable, because then the regression would not cause additional testing work. The project manager in the Developer Tool emphasises the importance of the right degree and phase of testing, otherwise the maintenance of tests can generate extra work. A senior executive of the project continues that indecisiveness in processing user feedback may cause overruns, and underlines the importance of decisiveness and time boxing in user testing.

5 Discussion

This section discusses the main case study findings, and presents the related practical and theoretical implications. This section also addresses the limitations of this study, and gives pointers for further research.

This study has captured in-depth experiences from two *typical* project organisations, who have established software development practices for one project, based on their internal guidelines. As is typical for these kinds of organisations, the maturity and optimisations of the practices are not on an exemplary level, because the projects exist for only a certain period of time, and the contents of the projects vary. The primary need of these organisations is to get the basic things right in every project, not to optimise the last details.

5.1 Implications for practice

This study clearly shows that the basic principles for estimating testing related tasks are the same as for estimating implementation tasks, but there is a strong tendency to take shortcuts and postpone tasks beyond their planned deadlines. The most significant finding is that the testing plan was not delivered in either project by the time of estimation. This can be compared to estimation of implementation with incomplete or no specifications, which is connected to overruns (Standish group, 1994; Lederer and Prasad, 1995). Especially in the Developer Tool project this was the reason for overruns, and the evidence suggests also that some surprises could have been avoided also in the Operational Control System project with a finalised testing plan.

Also, the practical estimation tasks were subject to taking short-cuts in the Developer Tool. While the implementation tasks were estimated by a developer and the initial project team was known, the testing was estimated by the product owner for an

unknown team. This is problematic from the estimation point of view, because the productivity of developers is known to vary significantly (Kemayel, Mili and Ouederni, 1991).

The attitudes towards testing and its estimation seem to be contradictory: on one hand testing cannot be omitted, but on the other hand it seems not to be a full-fledged part of the software project. This research has found that the testing personnel perceives that testing is considered as unimportant, and the management emphasises that the cost needs to be reasonable. The tendency to take shortcuts supports this perceived unimportance. The extant literature clearly shows that the attitudes of the senior management influences strongly the change or improvement of a certain area (Stelzer and Mellis, 1998; Wiegers, 1998), which means that the negative attitudes are likely to influence also the estimation of testing tasks negatively.

The technical staff in both projects has very positive attitudes towards automated testing. This is claimed to reduce manual work and improve predictability. This is a fair assumption, and confirmed, for example, by Ramler and Wolfmeier (2006) and Hoffman (1999), but only if implemented properly. They emphasise that the project characteristics need to be accounted when planning testing, in order to implement effective and reasonably priced testing. This was supported by the comments from a senior manager in the Operational Control System (*“the extent of testing must consider the project at hand”*) and (*“testing must add value to the project”*), the project manager from the Developer Tool (*“degree and phase of testing must be carefully planned”*) and a senior manager in the Developer Tool (*“decisiveness and time boxing is important in user testing”*).

Changing attitudes, as well as other capabilities, seem to take time. The Large Multinational has invested resources in building capabilities significantly within the past two years, but the know-how or attitudes are still not on the same level as in implementation. This corresponds to results from other change situations (Piderit, 2000).

As a summary, similar projects as the Developer Tool and Operational Control System are recommended to follow the same estimation practices for testing related tasks as for implementation. Neither of the projects reported that the reason for postponing or omitting tasks was cutting cost. In this sense, it seems counter intuitive and productive not to follow the same rigour in testing as with implementation, especially when the short-cuts seem to cause even severe estimation errors.

Finally, it was noted in the interviews that the personnel was rarely properly educated to use the estimation tools. While the other of our case study companies offered support and education in the estimation to the project management, the development teams were not aware of these services. Furthermore, only a few mentioned university education as a source of cost estimation knowledge, even though cost estimation based on an expert's opinion is nowadays a part of the standard workflow in many companies. While effort estimation is a part of the curriculum guidelines suggested by

ACM¹, there might be a need for education organisations to re-evaluate the content of teaching.

5.2 Implications for theory

The current SCE literature provides only a few case studies providing in-depth knowledge of real-life situations (Jørgensen and Shepperd, 2007), and, to our best knowledge, this is among the first studies to report on experiences related to estimating software testing. This paper contributes to the body of knowledge by showing that organisations seem to deviate from their standard practices when estimating testing related tasks, causing estimation errors. The reason for deviations resides in negative attitudes towards testing. Cost savings are rejected as a reason for deviations, as well as poorly performed testing as a reason for overruns in testing.

Furthermore, our literature review shows that there is a lack of empirical and theoretical studies on addressing software testing effort estimation. While further work such as a systematic literature review is needed to confirm this observation, this hints that one of the reasons for software cost estimation related problems is the lack of proper focus on testing and its effort estimation. However, new estimation tools or methods for software testing are not silver bullets. A holistic view is needed for improving the accuracy of software estimates.

5.3 Limitations and further research

This study has certain limitations. First, the findings of this study are subject to constraints of the research methodology. This research studied two very similar software projects, which limits the validity of the findings to similar contexts. It is recommended that further research would be conducted in a different context to see if e.g. larger projects or continuous product development projects suffer from similar testing estimation related problems as reported in this paper. Second, a qualitative study is always, to some degree, subjective and might be influenced by the expectations and opinions of the researchers and interviewees. However, we did our best to treat the data objectively and took countermeasures to reduce bias. For example, all transcripts and quotes, as well as the manuscript, were checked and approved by the interviewees before the publication.

Considering the big share of testing work in the overall software project and the problems reported in this study, further research of the topic is justified. First, our unstructured literature review did not reveal many studies in this topic. Thus, a more thorough literature review should focus to shed more light on possible research gaps

¹ Software Engineering 2014. Curriculum Guidelines for Undergraduate Degree in Software Engineering. https://www.acm.org/education/SE2014-20150223_draft.pdf

in software testing estimation. Second, there is a lack of studies addressing the methods and tools used in the industry to estimate software testing. Further work should be devoted to reveal the best practices already used in the industry.

6 Conclusions

This research provides evidence that software testing related tasks in software projects are estimated by using similar practices as for implementation related tasks, but deviations from the standard practice occur often. These deviations, such as estimating without a testing plan, estimating work that will be carried out by others or estimating without knowing the actual testers, have been a source for even severe overruns. The research rejects poorly performed testing as a source for overruns. Overruns themselves are a source of decreased team motivation and customer satisfaction, and additional work, among other things.

The results of this research suggest that the reason for process deviations resides in the negative attitudes towards testing. Widespread deviations from established practices among both project management and technical staff, together with the direct comments from the interviews, indicate that testing is not considered as important as implementation, and therefore deviations are allowed. The results reject cost savings as the reason for deviations.

The main implications from the results for software managers, experts, project managers and academia are the following:

- A deviating estimation process for software testing may lead to severe estimation errors.
- Software projects should use the same rigor in estimating testing as for estimating implementation. Deviations should not be allowed.
- Managers responsible for software and project processes must recognise the importance of testing and promote the importance of it to change the attitudes towards testing. This is necessary in order to establish the use of good practices in organisations.

Finally, the aforementioned serve also as a good starting point for further research.

Acknowledgements.

The authors gratefully acknowledge Tekes – the Finnish Funding Agency for Innovation, DIGILE Oy and Need for Speed research program for their support.

References

1. Ammann, P., Offutt, J.: *Introduction to Software Testing*. Cambridge University Press, UK (2008)
2. Benton, B.: *Model-Based Time and Cost Estimation for Software Testing Environment*. In: *Sixth International Conference on Information Technology: New Generations*, pp. 801-806. IEEE (2009)
3. Boehm, B., Abts, C., Chulani, S.: *Software development cost estimation approaches — A survey*. *Annals of Software Engineering*, 10(1-4), 177-205 (2000)
4. Boehm, B.: *Software Engineering Economics*. Prentice-Hall, Englewood Cliffs, New Jersey (1981)
5. Briand, L.C., Wieczorek, I.: *Resource Estimation in Software Engineering*. *Encyclopedia of Software Engineering*. John Wiley & Sons, Inc., New Jersey (2002)
6. Buenen, M., Teje, M.: *World Quality Report 2014-15: Sixth Edition*. Capgemini, Sogeti and HP (2014)
7. Calzolari, F., Tonella, P., Antoniol, G.: *Dynamic model for maintenance and testing effort*. In: *Proceedings of the International Conference on Software Maintenance 1998*, pp. 104-112. IEEE (1998)
8. Creswell, J.: *Research Design: Qualitative and Quantitative and Mixed Methods Approaches*. Second edition. SAGE Publications, Inc., Thousand Oaks, California (2003)
9. Engel, A. Last, M.: *Modelling software testing costs and risks using fuzzy logic paradigm*. *Journal of Systems and Software*, 80(6), 817-835 (2007)
10. Farr, L., Nanus, B.: *Factors that Affect the Cost of Computer Programming*. Volume I. Technical Documentary Report No. ESD-TDR-64-448. United States Air Force, Bedford, Massachusetts (1964)
11. Haberl, P., Spillner, A., Vosseberg, K., Winter, M.: *Survey 2011: Software Testing in Practice*. Auflage, dpunkt.verlag, (2012) http://www.istqb.org/documents/Survey_GTB.pdf
12. Hoffman, D.: *Cost Benefits Analysis of Test Automation*, *Software Testing Analysis & Review Conference (STAR East)*. Orlando, FL (1999)
13. Jørgensen, M., Shepperd, M.: *A Systematic Review of Software Development Cost Estimation Studies*. *IEEE Transaction on Software Engineering*, 33(1), 33–53 (2007)
14. Kemayel, L., Mili, A., Ouederni, I.: *Controllable factors for programmer productivity: A statistical study*. *Journal of Systems and Software* 16(2), 151-163 (1991)
15. Laurenz Eveleens, J. and Verhoef, C.: *The rise and fall of the Chaos Report figures*. *IEEE Software*, 27(1), 30–36 (2010)
16. Lederer, A.L., Prasad, J.: *Causes of Inaccurate Software Development Cost Estimates*. *Journal of Systems and Software*, 31(2), 125-134 (1995)
17. Lee, J., Kang, S., Lee, D.: *Survey on software testing practices*. *IET Software*, 6(3), 275-282 (2012)
18. Mieritz, L.: *Surveys Shows Why Projects Fail*. G00231952. Gartner, Inc. (2012)
19. Moløkken, K., Jørgensen, M.: *A Review of Software Surveys on Software Effort Estimation*. In: *Proceedings of International Symposium on Empirical Software Engineering*, pp. 220–230. IEEE (2003)
20. Nelson, E. A.: *Management Handbook for the Estimation of Computer Programming Costs*. Technical Documentary Report No. ESD-TR-67-66. United States Air Force, Bedford, Massachusetts (1966)
21. Ng, S.P., Murnane, T., Reed, K., Grant, D., Chen, T.Y.: *A preliminary survey on software testing practices in Australia*, In: *Proceedings of the 2004 Australian Software Engineering Conference*, pp.116-125. IEEE (2004)

22. Patton, M.: *Qualitative Research and Evaluation Method*, Third edition. SAGE Publications, Inc., Thousand Oaks, California (2001)
23. Piderit, S.K: *Rethinking Resistance and Recognizing Ambivalence: A Multidimensional View of Attitudes toward an Organizational Change*. *Academy of Management Review*, 25(4), 753-794 (2000)
24. Rahikkala, J., Leppänen V., Ruohonen, J., Holvitie, J.: *Top management support in software cost estimation: A study of attitudes and practice in Finland*. *International Journal of Management Projects in Business*, 8(3), 513-532 (2015)
25. Ramler, R., Wolfmaier, K.: *Economic perspectives in test automation: balancing automated and manual testing with opportunity cost*. *Proceedings of the 2006 international workshop on Automation of software test*, pp. 85-91. ACM (2006)
26. Runeson, P., Höst, M., Rainer, A., Regnell, B.: *Case Study Research in Software Engineering: Guidelines and Examples*. John Wiley & Sons, Inc., New Jersey (2012)
27. Standish Group: *The Chaos Report*. The Standish Group (1994)
28. Stelzer, D., Mellis, W.: *Success factors of organizational change in software process improvement*. *Software Process: Improvement and Practice* 4(4), 227-250 (1998)
29. *The CHAOS Manifesto: Think Big, Act Small*. The Standish Group International (2013)
30. Wieggers, K. E.: *Software process improvement: Eight traps to avoid*. *CrossTalk, The Journal of Defense Software Engineering* (1998)
31. Yin, R. K.: *Case Study Research: Design and Methods*. Third edition. SAGE Publications, Inc., Thousands Oaks, California (2003)

ICDO: Integrated Cloud-based Development Tool for DevOps

Farshad Ahmadighohandizi and Kari Systä

Department of Pervasive Computing, Tampere University of Technology
 Korkeakoulunkatu 10, FI-33720 Tampere, Finland
 Email: `firstname.lastname@tut.fi`

Abstract. This research is based on three drivers. Firstly, software development and deployment cycles are getting shorter and require automatic building and deployment processes. Secondly, elastic clouds are available for both hosting and development of applications. Thirdly, the increasingly popular DevOps introduces new organizational and business culture. This paper presents a research prototype and demonstrator of an integrated development tool. The tool is cloud based and thus accessible from any Web-enabled terminal. Automation is maximized so that deployment cycles can be as fast as possible. Since the aim is to use cloud resources as a utility in a flexible manner, cloud brokering – i.e. finding the most suitable provider – is included in the system. The contributions of the paper include: an idea of a new kind of DevOps tool, description on how it can be implemented on top of standard components and implications to software development processes.

Keywords: Continuous deployment, Cloud brokerage, Cloud federation, DevOps, Software development, EASI-CLOUDS project

1 Introduction

Continuous Integration (CI) [2] is used in many organizations and has shown its power in improving the efficiency of software development. Continuous Delivery and Deployment [3] expand CI with automatic delivery and deployment and are often attached with mechanisms to immediate collection of data of user behavior and fast reaction to that feedback [12]. DevOps [4] builds on CI and CD but also includes organizational and cultural aspects – especially it removes the wall between developers and operation personnel. There are several tools for continuous integration with automatic building and testing – Jenkins¹ is perhaps the most well-known. For automatic deployment and management of operations separate tools like Puppet² and Rundeck³ can be used. However, the assumption of these tools, i.e., the deployment is separated from the development, maintains the wall that DevOps wants to remove.

¹ <https://jenkins-ci.org/>

² <https://puppetlabs.com>

³ <https://rundeck.com>

Cloud technologies provide new opportunities for both development and operation. With cloud-based tools, developers get scalable and pervasive aid for all actions like coding, building and testing. Cloud-based development tools like CoRED [7] and Cloud9⁴ allow developers to access the tool if they just have access to network and browser. Therefore, developers do not need to install any software on their local machines, and the computing capacity for the tools can be elastically allocated according to current needs.

When applications and services are deployed into the cloud, several factors need to be considered. Enough resources need to be allocated for the hosting of the application to ensure the required service level under assumed usage load. Various legal and trust-related concerns can also affect the decision of the physical and virtual location of the service. Finally, the cost of running the service needs to be minimized.

ITEA2 project EASI-CLOUDS⁵ has developed technologies for allocation and management of cloud resources based on SLA (Service Level Agreement) and enables a market place for cloud resources through brokering and federation. In other words, EASI-CLOUDS targets both cloud providers and consumers. It helps cloud providers utilize each other's resources through federation and brokerage. It also assists cloud consumers to adopt to multi-cloud architecture and select cloud services by simply requesting them by using manifests.

In this paper, we show how development tools like IDE and continuous integration systems can be combined with brokering-based deployment and service operation. The same integrated tool can be used for both development and management of operations. In other words, use of the cloud-based tool starts with code editor for programming, continues with finding the suitable cloud resources for application hosting, goes on with automatic building and testing, and finally ends with the application deployment. In addition, the deployed application can be managed via the same tool.

The first step of thus work was done as a demonstrator for EASI-CLOUDS project for communicating the results to the project consortium and to external stakeholders. In addition, we conducted research on how cloud-based development tools and brokered cloud resources complement each other and how the ideas of DevOps integrate with the ideas of EASI-CLOUDS.

The remainder of this paper is organized as follows. Section 2 explains the fundamentals our research is based on. Section 3 describes the integrated tool, its usage and gives some of the implementation details of our prototype. Section 4 analyses the prototype and its implications. Section 5 presents comparison of our research to related work. Finally, Section 6 concludes and states future work.

⁴ <https://c9.io/>

⁵ <http://easi-clouds.eu/>

2 Technology background

2.1 Cloud-based IDE

Department of Pervasive Computing at Tampere University of Technology has developed a cloud-based IDE. Its different versions include CoRED which mostly focuses on collaborative aspects of the development and MIDEaaS which adds a visual UI editor to CoRED [7]. In our research, we chose CoRED as there was no need for a visual UI editor. Important features of CoRED include:

- *Cloud-based.* Running in the cloud, CoRED frees developers from problems like installation, upgrades and maintenance. To access CoRED and to start development only an up-to-date browser and a network connection are needed.
- *Real-time collaboration on common programming tasks.* This is similar to collaboration on document writing with Google Docs⁶. CoRED provides developers with various means to collaborate. Collaboration and communication of developers let them know what they are doing as a team and can prevent redundancy or even conflicting work. The different collaboration features of CoRED are explained in more detail in [6] and [10].

More information about CoRED and MIDEaaS is available through <http://cored.cs.tut.fi>. An open source version is also available.

2.2 Cloud Brokerage and EASI-CLOUDS

As more companies adopt cloud computing as a key enabler for their business, more cloud service providers emerge. This variety of cloud providers has generated new challenges for both providers and their end users. From the users' perspective, finding the appropriate cloud provider that fills the requirements on issues like security, billing, and supported technologies from number of possible providers is a challenging task.

EASI-CLOUDS project proposes brokerage and federation solutions for the above challenges. A broker is an entity that manages the use, performance and delivery of cloud services. It also negotiates relationships between cloud providers and cloud consumers [8]. Cloud federation provides cloud provider with a mean to send a cloud request to multiple cloud providers as if they were a single cloud provider [5]. EASI-CLOUDS project developed both technologies. The aim of the project was to build a system that assists in usage of offerings from several cloud providers and acts as an intermediary between providers and their users. It helps consumers to find and select the most suitable cloud resource among several providers, and to provision it in a unified mechanism by making providers interoperable.

EASI-CLOUDS project reuses components from an earlier project CompatibleOne⁷. CompatibleOne introduces a common description model for Cloud

⁶ <https://www.google.com/docs/about/>

⁷ <http://www.compatibleone.org>

resources and a platform which executes provisioning tasks for these resources [14]. The common description model is called CORDS (CompatibleOne Resource Description Model) and the execution platform is called ACCORDS (Advanced Capabilities for CORDS).

2.3 Provisioning Heterogeneous PaaS Resources

Different cloud providers may base their offerings to different PaaS technologies (e.g., Cloud Foundry⁸, OpenShift⁹, Jelastic¹⁰, etc.) for provisioning their resources. Switching from a provider using a PaaS solution to another one with a different solution takes much effort and learning. So, a PaaS-independent solution to enable use of heterogeneous PaaS resources is required.

CompatibleOne Application Platform Service (COAPS) [11] offers this PaaS-independent solution. It provides a unified model to describe PaaS resources regardless of their hosting providers. It also proposes a generic API which is an abstraction layer for heterogeneous PaaS providers. COAPS API is shown in Figure 1.

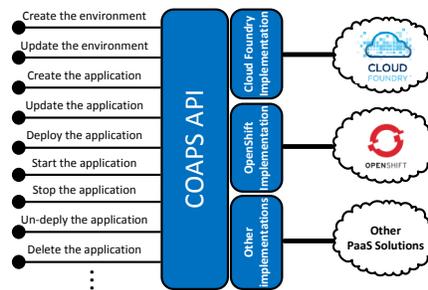


Fig. 1. COAPS API is an intermediary for management of heterogeneous PaaS resources [1]

The unified description model proposed by COAPS divides PaaS resources into two parts: *Application* resource and *Environment* resource which prepares the required environment to host the application. The description language for resources in COAPS is similar to CORDS description model in ACCORDS (see Subsection 2.2). This similarity facilitates their communication as they are designed to be used together.

For a cloud application to be deployed to a PaaS through COAPS two steps are needed as shown in Figure 2. Firstly, environment and application resources should be provisioned on the target PaaS by calling related COAPS APIs (*create the environment* and *create the application* APIs respectively). In this step,

⁸ <https://www.cloudfoundry.org/>

⁹ <https://www.openshift.com/>

¹⁰ <https://jelastic.com/>

COAPS also associates application and environment resources with unique IDs and sends these IDs back to the caller for further references. In the second step, actual deployment and starting of the application is done with *deploy the application* and *start the application* methods in COAPS API.

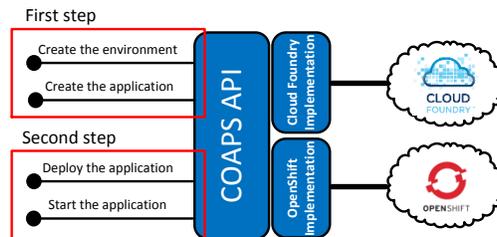
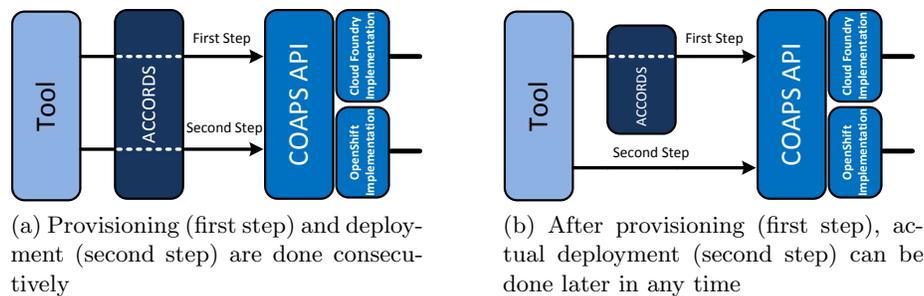


Fig. 2. Steps to host and run an application through COAPS

2.4 Application Deployment methods in EASI-CLOUDS

To deploy an application to a multi-provider PaaS architecture using EASI-CLOUDS components, two methods are proposed as follows:



(a) Provisioning (first step) and deployment (second step) are done consecutively

(b) After provisioning (first step), actual deployment (second step) can be done later in any time

Fig. 3. Immediate (a) and deferred (b) deployment methods

Immediate Deployment. In this method, all requests for brokering, provisioning, and deployment are sent to ACCORDS. ACCORDS first finds the most appropriate PaaS provider through the brokerage. Then for the provisioning cloud resources (step 1 in Figure 2) and the deployment and running the application (step 2 in Figure 2), it interacts with COAPS. In other words, ACCORDS performs two steps shown in Figure 2 one after another as an atomic operation. Figure 3(a) shows how in immediate deployment method a tool should interact with ACCORDS to host and run an application in the selected PaaS.

Deferred Deployment. In his method, the only request for brokering is sent to ACCORDS. After brokerage, ACCORDS interacts with COAPS of the selected

PaaS to create environment and application resources (step 1 in Figure 2). Actual deployment of the cloud application (step 2 in Figure 2) can be done later in any time by direct interaction with COAPS. Figure 3(b) illustrates how in deferred deployment method a tool should communicate with ACCORDS and COAPS to host and run an application.

3 The integrated development tool

3.1 Same tool for development, deployment and operation

CoRED is a cloud-based IDE and its original implementation already included a simple deployment that could be done after building. In this research, we added a programmable CI system that can control more complex build and test procedures. In addition, we integrated deployment to manageable PaaS platform and cloud brokering to the system. Thus, the developer can use the same tool to develop, build, allocate cloud resources and to deploy the application.

Our tool demonstrates DevOps by bringing development and operations into one integrated tool. Users of our tool are provided with an *operation and management* console. A snapshot of this console is shown at the bottom part of Figure 4. Users can start, stop, and delete the deployed applications via this console. There is also a hyperlink to each application so that the user can easily run the deployed applications. Figure 4 shows that one application named Addressbook has been deployed to two PaaS providers (F-Secure¹¹ and BULL SAS¹²)¹³. It also shows that current status of the application deployed to BULL SAS is *stopped*, whereas the state of the application hosted at F-secure is *started*.

3.2 Integrated User Interface

Several existing tools are used as components of our prototype. Each of them has initially designed to perform specific tasks such as editing code, brokering, building, and deployment. The goal of our research is to provide a uniform interface to the user. Thus, the user does not need to switch between several tools.

Figure 4 shows a snapshot of our prototype. The left column is for management of the project files, the top right part is for editing code, and the bottom part has two tabs. Tab named *deploy* is for brokerage, initiation of the build and deployment, and management of operations. *Console* tab displays logs and possible errors appeared in build and deployment phase as illustrated in Figure 5. The left panel of console tab shows a high-level view (i.e., pushing code to the repository, build, and deployment), the middle panel shows the build log, and the right panel shows the deployment log.

¹¹ <https://www.f-secure.com>

¹² <http://www.bull.com/>

¹³ F-secure and Bull were two partners of EASI-CLOUDS project

The screenshot displays the 'Operation and Management Console' for an application named 'addressbook'. The interface is divided into several sections:

- Project Contents:** A tree view on the left showing 'NAME', 'Views', 'Main', 'Other Files', and 'App.java'.
- Console:** A central area displaying 16 lines of Java code, including package declarations and imports for Vaadin classes like `Theme`, `Title`, `Container.Filter`, `Item`, `Property`, `ValueChangeEvent`, `FieldGroup`, `FieldGroup.CommitException`, `IndexedContainer`, `FieldEvents`, `TextChangeEvent`, `TextChangeListener`, `AbstractTextField`, and `TextChangeEventMode`.
- Server Management:** A table titled 'CURRENT SERVERS' with columns for 'APP ID', 'STATUS', and actions. It lists two servers: 'BULL SAS' (APP ID 17, STATUS STOPPED) and 'F SECURE' (APP ID 18, STATUS STARTED). Each server has buttons for 'start', 'stop', and 'delete'. There are also 'Link To App' and 'remove PaaS' buttons.
- Actions:** Buttons for 'Find a new host', 'Deploy to selected hosts', 'Collect Data', and 'Show usage data' are located at the bottom right.
- Build and Test:** On the left, there are buttons for 'Add New', 'Delete', 'Build', 'Build Log', and 'Test App'.

Fig. 4. A snapshot of our demonstrator

Deploy	Console
Commit and Push Log	Build Log
Changes are pushed.	Started by an SCM change
Waiting for build to start ...	Building in workspace
Build is started ...	/usr/share/tomcat7/jenkins/workspace/addressbookdemo
build is successfully done.	> git rev-parse --is-inside-work-tree # timeout=10
Deployment is started...	Fetching changes from the remote Git repository
Deployment finished.	> git config remote.origin.url /home/ubuntu/git/addressbookdemo.git # timeout=10
	Fetching upstream changes from /home/ubuntu/git/addressbookdemo.git
	> git --version # timeout=10
	> git -c core.askpass=true fetch --tags --progress
	/home/ubuntu/git/addressbookdemo.git +refs/heads/*:refs/remotes/origin/*
	> git rev-parse refs/remotes/origin/master^{commit} # timeout=10
	> git rev-parse refs/remotes/origin/origin/master^{commit} # timeout=10
	Checking out Revision f535b2ef0ece2b50f96a16052b86c179ede94b
	(refs/remotes/origin/master)
	> git config core.sparsecheckout # timeout=10
	> git checkout -f f535b2ef0ece2b50f96a16052b86c179ede94b
	> git rev-list --topo-order --reverse --format=%H %s
	timeout=10
	addressbookdemo1 \$ /usr/share/maven/bin/mvn vaadin:update-widets
	Deployment Log
	Application Name: addressbookdemo
	Environment ID: 2
	Application ID: 6
	Deployment may take several minutes ...
	deploy Application Response: addressbookdemo_f-secure
	description.addressbookdemo_f-secure.130.230.142.107.xip.io
	URL of Application: addressbookdemo_f-secure.130.230.142.107.xip.io
	start Application Response: addressbookdemo_f-secure
	description.addressbookdemo_f-secure.130.230.142.107.xip.io

Fig. 5. Build and deployment steps together with their log is reported to the user

3.3 Detailed workflow and implementation

This section walks through the operation of our cloud-based development tool and describes the implementation details of the proposed solution. Figure 6 shows the components of our tool. These components are referenced and their roles are explained in the rest of this section.

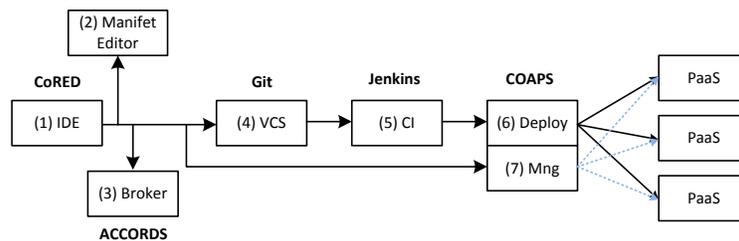


Fig. 6. Cloud-based development tool and our demonstrator

As the editor, (1) in Figure 6, we used CoRED (Subsection 2.1). The initial version of CoRED only supported Vaadin¹⁴ applications. Although support for a range of new languages has later been added to CoRED, our research has been done on developing Vaadin applications in Java programming language.

To deploy an application into a multi-cloud architecture we integrated CoRED with cloud brokering solution, (3) in Figure 6, of EASI-CLOUDS (see Subsection 2.2). We chose the deferred deployment method (see Subsection 2.4) to enable

¹⁴ <https://vaadin.com/>

users to manage the operations of the deployed applications. More details about this choice are given in Subsection 4.2.

In the deferred method, brokering is done before deployment. In the user interface shown in Figure 4, brokering is initiated by pressing 'Find a new host' button. Then users will then be provided with a graphical tool, (2) in Figure 6, to define requirements for the needed cloud resources and to adjust quantities like the amount of memory and disk space, required database solution, the number of instances, and desired geographical location of PaaS provider. These settings will be sent to the brokering system that will choose the most suitable PaaS provider. Although we tested usage of ACCORDS for brokering, it was replaced by a mock-up in the final implementation since the current ACCORDS do not have proper support for deferred brokering.

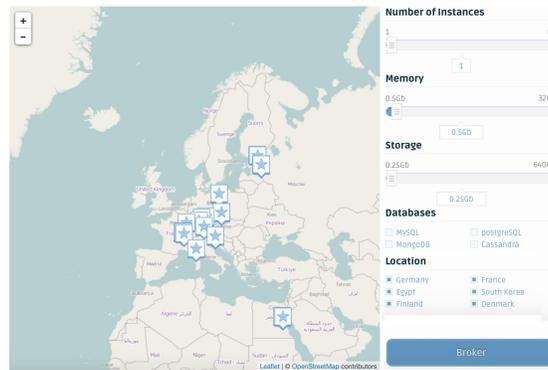


Fig. 7. Graphical tool with which users can specify required cloud resources

The automated building and deployment pipeline is started by pressing *Deploy to the selected hosts* button (see Figure 4) from the IDE. In the first step of automated pipeline the edited source files of the project are committed and pushed to source code repository, (4) in 6, that is based on Git¹⁵ revision management system. Git repository informs related Jenkins task about changes. Consequently, the Jenkins task fetches the updated project and invokes build environment, (5) in 6, using Maven¹⁶ technology. Maven manages the building process and creates the final WAR file. Each of these steps and examples of their corresponding logs are shown to the user as shown in Figure 5.

One important reason behind our choice of Jenkins is its plugin support. A Jenkins task can be extended by implementing extension points defined in different stages of its life cycle. For example, to perform deployment, post build extension point was extended to use COAPS for deployment – see (6) in Figure 6.

¹⁵ <https://git-scm.com>

¹⁶ <https://maven.apache.org>

Users are also provided with an *operation and management* console, (7) in Figure 6, through which operations of the deployed applications can be managed (see Subsection 3.1).

Our tool is running on TUT infrastructure which is based on OpenStack¹⁷. One virtual machine hosts CoRED and COAPS, whereas Jenkins CI server and Git server runs on another virtual machine. In addition, one PaaS for hosting applications is running Cloud Foundry on a separate virtual machine.

4 Analysis

The resulting system has been tested with a few applications during the project. Different development phases of the tool have been demonstrated to various stakeholders. Based on experiments with those examples we can say that the proposed system is possible and the demonstrator communicates the idea. This section summarizes our key findings regarding development practices and use of DevOps together with cloud brokering. In addition, we report the implementation related issues that need to be solved before commercial exploitation of the ideas.

4.1 Development process and practices

The CoRED code editor has been designed for collaborative coding where several developers can participate in changing the same file simultaneously. However, many organizations use revision control systems (RCS) to manage the collaboration since developers check the code in when they think that others should see it. In our demo the check-in always triggers the building process, but since we use Git we could have implemented traditional RCS-controlled collaboration by using two separate branches, one for continuous integration and the other for sharing code between developers. The different nature of collaboration with cloud-based IDE and with RCS have been discussed in [9]. In that paper, the authors note that with cloud-based editors three new paradigms are possible. Here we discuss these paradigms in the light of this research.

1. **Real-time collaboration between developers instead of revision control based collaboration.** This has also been the default assumption in our case, i.e., tools help collaboration instead of preventing simultaneous work on the same file. Our domain and assumption of fast release cycles also emphasize the need for collaborative coding since applications are developed as a series of feature increments and each developer develops a specific feature instead of a module or file as in traditional software development.
2. **Automatically created versions.** Revision control has actually several purposes: control of the collaboration, provide ways to backtrack to old version and to trigger automated building and deployment pipeline. The research in [9] notes that modern IDEs have a compiler-like functionality as

¹⁷ <http://www.openstack.org>

a background system for syntax checking and that system could recognize coherent systems to be inserted in the revision database automatically. Although we do not see a need for automatically created deployable version, the amount and nature of automatic quality check integrated into IDE could be more extensive and elastic cloud resources could be used for implementation of such systems.

3. **Co-operation for a jointly created version or solving a conflict.** This would not be needed in our current demo set-up but would be very important use case if components are imported from other projects – especially when the project is been created.

4.2 DevOps

DevOps and organizations that are both software developers and service providers were among core assumptions of our research. Thus, the target was to develop a demonstrator of a tool that supports both development and operation through an integrated user interface.

As discussed in Section 3 we selected deferred deployment but struggled with some implementation issues when we wanted to integrate the ACCORDS brokering platform. ACCORDS would have been easier to integrate if we had selected immediate deployment. However, our work started to follow a simple demo plan that assumed the deferred deployment. After implementing and working with the prototype, we have learned that the selection between those approaches is not obvious.

- Immediate deployment assumes that the user has the built and tested binary before brokering. On the other hand, the user should be given feedback on the results of the brokering – and even have a veto – before allocation of cloud resources and deployment. This means that the whole chain from the version management to deployment cannot be automated since the user needs to be consulted about the results of brokering after building but before brokering. This means that deployment would actually be a separate step.
- In the immediate deployment ACCORDS assumes that it will manage the cloud resources after deployment. However, we wanted to demonstrate a DevOps tool with an integrated user interface to control the whole chain from development to deployment. In the immediate deployment method, interactions between ACCORDS and COAPS are hidden and not accessible to our integration core (all components were integrated to CoRED). For example, provisioning and deployment logs could not be shown in the integrated UI. Also offering the management view would not be possible since access to COAPS through ACCORDS was incomplete.
- Most of the time application deployments are updates to an existing application. In the case of update, deployed application could simply replace the old version, in some other cases it is better to allocate new PaaS with new cloud resources for the new version and to make the switch over as a separate and controlled action. Use of immediate deployment, at least with the current implementation of ACCORDS, would not give enough control for this.

4.3 Implementation issues

Although we showed that the presented integration is possible, we did not use the different components in a way that was intended by their developers. This led to some integration problems. In the following, we list some of the issues we had in our implementation.

Selection between immediate and deferred deployment method in brokering was a difficult choice. At first, the immediate method seemed the natural choice. However, in the immediate deployment method, interactions between ACCORDS and COAPS are hidden from CoRED's point of view. For example, provisioning and deployment logs cannot be shown in CoRED. In addition, there is no way to call COAPS APIs through ACCORDS which would make implementation of the management view in the integrated tool impossible. In addition to reasons discussed in Subsection 4.2, these limitations of ACCORDS led us to the deferred deployment method. In the deferred method, the application deployment can be carried out via a direct interaction with COAPS at any time after resource provisioning. This direct interaction with COAPS allows access to the wide range of generic APIs of COAPS. For example, each button in management view should call the related COAPS API; Stop and delete buttons call *stop the application* and *delete the application* APIs respectively.

During the EASI-CLOUDS project, we installed an instance of ACCORDS and tested it together with our integrated tool. However, our current demonstrator does not use ACCORDS, since the implementation of its deferred deployment method was not as mature as immediate deployment. Consequently, integration to our tool was beyond the time and resources we had to complete the project. Instead, we use the graphical tool, shown in Figure 7, to collect resource requirements for brokering and we simulated the brokering behavior of ACCORDS. This interface was developed in parallel and by another organization, and the user experience is not that well integrated as other parts of the tool.

Although we deployed applications to different PaaS providers (e.g., University of Helsinki, Public Cloud Foundry, etc.) through our tool, deployment to all providers offered in the graphical tool is not implemented. Thanks to COAPS, new PaaS providers could be easily added to our tool.

Software development tools and conventions deal with various information about the application. Examples of such information include dependencies, external components, and required operating systems and services. That information is an important input to brokering and should not be manually queried from the user. All that information can be included in the data that is stored in the revision control database. Actually the data includes all input to brokering, SLA agreements and output of brokering. This information is not necessarily logical part of the revision since a single revision may be deployed to several locations. In our demo and proof of concept, we stored the result of the brokerage among the other files of the project and thus it was available for CI and CD pipeline. In a real implementation, the system should separate revisioning of the application and deployment information. Both of them should be stored in RCS as separate but interdependent entities.

5 Related work

Individual cloud-based development tools have been implemented for different purposes regarding software development and deployment. These tools range from simple code editors to tool sets that complete several parts of development and deployment pipelines. However, we believe that our tool is unique in the sense that it offers some features that have never been implemented before in one integrated tool.

There are several cloud-based IDEs and many of them could be used as a component of the integrated tool. One of such IDEs is Cloud9. Its backend has been implemented using Node.js and its frontend is based on a JavaScript-based editor called ACE. The editor used in our research, CoRED, is also based on ACE editor, whereas our backend has been implemented based on Vaadin framework. Cloud9 supports deployment to various services (e.g., Heroku, Google App Engine, Cloud foundry, etc.). But deployment to each of them needs a user to install related command line tool and learn how to use it.

One of the tools similar to our work is IBM Bluemix¹⁸. It can be used to build, run, and manage different kinds of apps targeted to the web, mobile, and smart devices. Bluemix offers a wide variety of runtimes (e.g. Java, Go, PHP, etc.) and a web-based editor for online coding. While Bluemix editor gives the same experience as the best desktop tools for certain programming languages like JavaScript, syntax-highlighting is the only significant feature when it comes to Java. As a contrast, CoRED supports more advanced features like error checking, code completion, and collaborative features. Moreover, Bluemix is built on Cloud Foundry. But we leverage COAPS to add support for heterogeneous PaaS technologies. Finally, Bluemix lacks cloud brokerage which is the key feature of our work.

Using DevOps tools such as Chef¹⁹ and Juju²⁰, one can automate deployment processes and configure the underline infrastructure. However, these tools support certain artifacts. So, implementation of a comprehensive automated deployment pipeline requires several tools. [13] describes how to orchestrate these artifacts by transforming them into a standards-based TOSCA model²¹. In our research, we orchestrate various DevOps artifacts supported by different PaaS technologies (e.g., Cloud Foundry, Openshift, etc.) through COAPS. COAPS defined a unified model for representation of DevOps artifacts and proposes generic APIs. Moreover, in our research, the DevOps experience is provided through a single and integrated tool.

6 Conclusion

In this research, we have combined automation and brokered cloud to establish an integrated development tool. By using our tool, one can develop applications

¹⁸ <http://www.ibm.com/cloud-computing/bluemix/>

¹⁹ <https://www.chef.io/chef/>

²⁰ <https://jujucharms.com/>

²¹ <https://www.oasis-open.org/committees/tosca/>

in a cloud-based IDE, find the most suitable cloud provider, and deploy the application in the target platform. To enable the fast development and delivery cycle, we automated all steps. Firstly, cloud brokerage automatically finds the deployment target out of several cloud platforms. Secondly, automatic provisioning and deployment are done through a unified interface for heterogeneous PaaS. Finally, continuous deployment pipeline automates code integrations, builds, and deployments.

Our proof-of-concept also demonstrates DevOps by enabling users to manage the deployed applications from a single tool. Users are provided with an operation and management console inside the tool with which they can stop, start, and delete the deployed applications.

For the future work, we first plan to connect our graphical tool to an instance of ACCORDS instead of simulating its behavior. Moreover, we believe that there are many automation opportunities besides build, testing and deployment steps while developing software. For instance, applications can be automatically instrumented so that they collect usage data. By the help of such system, the developers receive fast feedback of which features are important to users, thereby focusing on them first.

Acknowledgments

Major part of the work has been done in ITEA2 project EASI-CLOUDS (2012-2014) and the authors would like to thank all members of the consortium, and especially Mario Lopez-Ramos, Janne Lautamäki, and Jamie Marshall. Part of this research has also been done in Tekes-funded Digile program Need for Speed²² (2014-17).

References

1. Computer Science department. Telecom SudParis.: The compatible one application and platform service (coaps) api specification. <http://www.compatibleone.com/community/wp-content/uploads/2014/05/COAPS-Spec.v1.5.3.pdf>.
2. Fowler, M.: Continuous integration (May 2006) <http://www.martinfowler.com/articles/continuousIntegration.html>.
3. Humble, J., Farley, D.: Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation. 1st edn. Addison-Wesley Professional (2010)
4. Humble, J., Molesky, J.: Why enterprises must adopt devops to enable continuous delivery. *IT Journal* **24**(8) (2011) 6–12
5. Jekal, M., Krebs, A., Nurmela, M., Peltonen, J., Röhr, F., Plogmeier, J.F., Altmann, J., Gagnaire, M., Lopez-Ramos, M.: Deliverable 1.5 final business models for easi-clouds. easi-clouds project report. (2014) http://easi-clouds.eu/wp-content/uploads/2014/12/Deliverable_1_5_Final_business_models.docx.

²² <http://www.n4s.fi/en/>

6. Kilamo, T., Nieminen, A., Lautamäki, J., Aho, T., Koskinen, J., Palviainen, J., Mikkonen, T.: Knowledge transfer in collaborative teams: Experiences from a two-week code camp. In: Companion Proceedings of the 36th International Conference on Software Engineering. ICSE Companion 2014, New York, NY, USA, ACM (2014) 264–271
7. Lautamäki, J., Nieminen, A., Koskinen, J., Aho, T., Mikkonen, T., Englund, M.: Cored: Browser-based collaborative real-time editor for java web applications. In: Proceedings of the ACM 2012 Conference on Computer Supported Cooperative Work. CSCW '12, New York, NY, USA, ACM (2012) 1307–1316
8. Liu, F., Tong, J., Mao, J., Bohn, R., Messina, J., Badger, L., Leaf, D.: NIST Cloud Computing Reference Architecture: Recommendations of the National Institute of Standards and Technology (Special Publication 500-292). CreateSpace Independent Publishing Platform, USA (2012)
9. Mikkonen, T., Nieminen, A.: Elements for a cloud-based development environment: Online collaboration, revision control, and continuous integration. In: Proceedings of the WICSA/ECSA 2012 Companion Volume. WICSA/ECSA '12, New York, NY, USA, ACM (2012) 14–20
10. Nieminen, A., Lautamäki, J., Kilamo, T., Palviainen, J., Koskinen, J., Mikkonen, T.: Collaborative coding environment on the web: A user study. *Developing Cloud Software Algorithms, Applications, and Tools*,(60) (2013) 275–300
11. Sellami, M., Yangui, S., Mohamed, M., Tata, S.: Paas-independent provisioning and management of applications in the cloud. In: *Cloud Computing (CLOUD), 2013 IEEE Sixth International Conference on*. (June 2013) 693–700
12. Suonsyrjä, S., Mikkonen, T.: Designing an unobtrusive analytics framework for java applications. In: *Accepted to IWSM Mensura 2015, to appear*
13. Wettinger, J., Breitenbücher, U., Leymann, F.: Standards-based devops automation and integration using toasca. In: *Proceedings of the 2014 IEEE/ACM 7th International Conference on Utility and Cloud Computing. UCC '14*, Washington, DC, USA, IEEE Computer Society (2014) 59–68
14. Yangui, S., Marshall, I.J., Laisne, J.P., Tata, S.: Compatibleone: The open source cloud broker. *Journal of Grid Computing* **12**(1) (2014) 93–109

A State Space Tool for Concurrent System Models Expressed In C++

Antti Valmari

Department of Mathematics
Tampere University of Technology
P.O. Box 553, FI-33101 Tampere, FINLAND
Antti.Valmari@tut.fi

Abstract. This publication introduces a state space exploration tool that is based on representing the model under verification as a piece of C++ code that obeys certain conventions. This approach facilitates experimenting with many kinds of modelling ideas. On the other hand, the use of stubborn sets and symmetries requires that either the modeller or a preprocessor tool analyses the model at a syntactic level and expresses stubborn set obligation rules and the symmetry mapping as suitable C++ functions. The tool supports the detection of illegal deadlocks, safety errors, and may progress errors. It also partially supports the detection of must progress errors.

Keywords: model checking; stubborn sets; symmetries; safety; progress

1 Introduction

This publication discusses A State Space Exploration Tool called ASSET. It started in autumn 2014 as a simple quickly written tool that could be used to illustrate deadlocks, livelocks, the effect of the level of atomicity, and other concurrency-related issues to advanced programming students. The students had programmed in C++. An important goal was that starting to use the tool would not require complicated installation or learning a new language. Indeed, to install ASSET, it suffices to download one C++ file called `asset.cc`. (However, the need to learn concurrency-oriented *concepts* such as local state, global state and transition cannot, of course, be avoided.)

At the same time, the need arose to test a new method [10] of alleviating the state explosion problem. ASSET proved very suitable for that purpose.

ASSET is written in C++. From the point of view of its users, the key difference to other state space tools is that ASSET does not have an input language of its own, such as the Promela language of the SPIN tool [4] or the textual CSP language of the FDR tool [7]. Instead, the user represents the model under verification as a piece of C++ code that uses the pre-defined type `state_var` provided by ASSET and implements certain functions such as `fire_transition`, `print_state`, and `check_state`. The model is checked by copying it to the file

`asset.model` and then compiling and executing the file `asset.cc`. The latter `#includes` the former.

The advantage is that the user can use most C++ features and his/her earlier programming knowledge very flexibly when writing models of concurrent systems and verification questions. The examples in this publication give an idea of what can be achieved. The disadvantage is that although the modelling of individual transitions is simple and natural, the part of code that directs execution to the right transition is often an ugly collection of `if`- and `switch`-statements.

This approach leads to very fast execution of the transitions of the model, because the model is compiled into machine code instead of being simulated by ASSET. This idea is not new. For instance, SPIN uses it.

Another drawback is that ASSET cannot perform any syntactic analysis on the model, because it is not read by ASSET but by the C++ compiler. This is not a big problem for plain state space exploration. On the other hand, some advanced methods such as the symmetry method [2, 3, 5] require such analysis.

The symmetry method relies on a function that, in a certain sense, represents the symmetry that is exploited. The typical approach is that the input language supports modelling the system in a way that makes the symmetry obvious, so that the tool can easily pick it and construct the necessary function. This means that also the modeller knows the symmetry. Designing the function only requires basic understanding of the symmetry method, and representing it in C++ is just a programming problem. ASSET is used in this way in Section 6. Although this is of course less convenient than the typical approach, it is realistic, especially when ASSET is used for teaching or for scientific experiments.

The advanced method known as stubborn sets [8–10] requires so complicated syntactic analysis that most users cannot be expected to perform it. However, it is reasonable to make experiments where an expert performs the analysis and writes the result in a form that is suitable for ASSET. The results of such experiments would tell whether it would be worth the effort to implement a preprocessor tool that would perform the analysis, or to implement the method in other state space exploration tools than ASSET. The experiment in [10] was of this kind, and another is reported in Section 5.

ASSET and many models are available at <http://www.cs.tut.fi/~ava/ASSET/>.

Throughout this publication, a *demand-driven token ring* is used as an example. It is introduced in Section 2 and modelled for ASSET in Section 3. Section 4 discusses the features that ASSET offers for specifying correctness properties. Sections 5 and 6 focus on the use of the stubborn set and symmetry methods in ASSET. Results of the experiments with the example system are collected in Table 1 towards the end of the publication. They clearly show the good time and memory efficiency of ASSET and the benefits of stubborn sets and symmetries.

2 The Demand-Driven Token Ring

The demand-driven token ring system is a mutual exclusion system. Its overall structure is shown in Figure 1. The system consists of n clients C_0, \dots, C_{n-1}

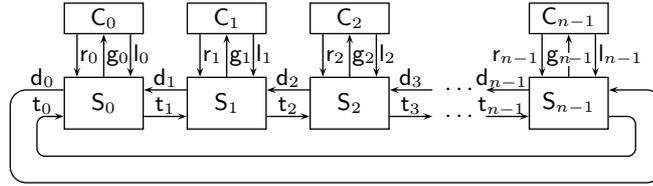


Fig. 1. Overall structure of the demand-driven token ring.

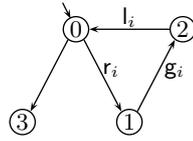
and n servers S_0, \dots, S_{n-1} . Each client has a specific piece of code called *critical section*. The purpose of the system is to ensure that two or more clients are never simultaneously in their critical sections. This property is called *mutual exclusion*. The system must also guarantee *eventual access*, that is, always when a client has requested for access to its critical section, it eventually gets the permission to go there. The servers receive requests from the clients, grant permissions to enter the critical sections, and receive notices that the clients have left the critical sections. In Figure 1, these are denoted with arrows labelled with r_i , g_i , and l_i .

To guarantee mutual exclusion, precisely one *token* circulates in the ring. Only the server that has the token may grant permission to its client. To avoid unnecessary activity, the token is circulated only when there is a pending request. When a server S_i that does not have the token gets a request from its client, it demands the token from the previous server via d_i . The demand propagates in the ring until it reaches the server S_j that has the token. If C_j has made a request or is in its critical section, then S_j grants permission if it has not yet done that, and waits until C_j has left its critical section, after which S_j gives the token to $S_{j \oplus 1}$. By $x \oplus y$ we mean $(x + y)$ modulo n . Otherwise S_j just gives the token to $S_{j \oplus 1}$. The server $S_{j \oplus 1}$ first serves its own client if it has made a request, and then gives the token to $S_{j \oplus 2}$. In this way the token eventually propagates to S_i which can then grant permission to its server.

The clients are shown in state machine form in Figure 2 left. The critical section consists of state 2. In addition to the r_i - g_i - l_i cycle that models requesting for access and then entering and leaving the critical section, there is a transition from state 0 to state 3. It models the *unforced request* property, that is, that the client may choose to not request for access. This is important for catching certain kinds of errors.

To see this, assume that the servers are modified such that after getting the token, each server always waits until its client makes a request and then serves its client before giving the token to the next server. This is incorrect, because if the client never makes the request, then the token is blocked at that server, and the requests by other clients remain unsatisfied forever.

However, in the absence of the transition from state 0 to state 3, this error would not be caught. This is because most if not all formalisms for concurrency make implicitly or explicitly the assumption that if something can happen, then something will happen. (Without this assumption, systems could deadlock at any time without a reason, which of course would be an inappropriate model



```

0: wait until  $r_i$  or  $d_{i\oplus 1}$  has occurred
   if  $t_i$  has not occurred then  $d_i$ 
   goto 1
1: wait until  $t_i$  has occurred
   if  $r_i$  has occurred then  $g_i$ ; goto 2
   else  $t_{i\oplus 1}$ ; goto 0
2: wait until  $l_i$  has occurred
    $t_{i\oplus 1}$ ; goto 0

```

Fig. 2. The clients and servers.

of reality.) With the assumption, with the modified servers, and without the transition from state 0 to state 3, consider the situation where every client has made the request except the one whose server has the token, and all demands have propagated to that server. Then the only thing that can happen is that this client makes a request. The assumption forces it to happen. So the client is forced to make the request even if it does not want to. The client behaviour that would reveal that the servers are incorrect is thus left out from the analysis.

Another way to look at this is that there is a fundamental difference between the transitions from state 2 to state 0 and from state 0 to state 1. If the client chooses to stay in its critical section — that is, state 2 — forever, then requests by other clients cannot be satisfied without violating mutual exclusion. On the other hand, not satisfying them violates eventual access. Because of this, every reasonable analysis of eventual access assumes that if a client is in its critical section, it eventually leaves it. On the other hand, we just argued that we must not assume that if a client is in its initial state, it eventually makes a request. This is a fundamental difference.

Often this difference is represented with so-called idling transitions and fairness assumptions, as explained in [6]. Another mainstream method is to exploit some variant of so-called failures [1, 7]. We do so by adding the transition from state 0 to state 3. The client is not forced to take the transition from state 0 to state 1, because it can take the transition from state 0 to state 3 instead.

That this model is appropriate has been argued in [10, 11], based on the theory of (stable) failures. We do not present full details here, but do briefly discuss one issue that has caused confusion. Because the client cannot come back from state 3 to state 0, this model might seem inappropriate in the situation where the client does not want to make a request now but wants to make it some later time. However, this situation is represented by the possibility of the client staying at state 0 until it wants to make a request again. Moving to state 3 only represents the possibility of making a request *never* again.

The behaviour of the servers is too tricky to be shown here as a state machine. Instead, it is represented in pseudocode in Figure 2 right. A precise ASSET model will be shown in Figure 4.

The server stays in state 0 as long as it has no request or demand to serve. While in this state, it may have the token. A request or demand makes it to move to state 1. When going there, it demands the token, if it does not yet have it. It

```

#ifdef size_par          // n is the number of clients and servers
const unsigned n = size_par; // n comes from the compilation command
#else
const unsigned n = 6;      // default value of n
#endif

state_var
  C[n] = 2, // state of client i:
          // 0 = idle, 1 = requested, 2 = critical, 3 = terminated
  S[n] = 2, // state of server i:
          // 0 = idle, 1 = waiting for token, 2 = waiting for client
  T[n] = 1; // true <=> server i has token

#ifdef symm_must // a trick used with the symmetry method
state_var c0now; // the current index of the original client 0
#else
unsigned const c0now = 0;
#endif

const char Cchr[] = { '-', 'R', 'C', ' ' }, Schr[] = { 'i', 'w', 't' };
void print_state(){
  for( unsigned i = 0; i < n; ++i ){
    std::cout << Cchr[C[i]] << Schr[S[i]];
    if( T[i] ){ std::cout << '*'; }else{ std::cout << ' '; }
  }
  std::cout << '\n';
}

```

Fig. 3. Model of the demand-driven token ring, part 1.

stays in state 1 until it has the token. If it does not have a pending request by its client, it gives the token to the next server and returns to the initial state 0. Otherwise it grants the permission to its client and goes to state 2. It waits there until its client leaves the critical section, after which it gives the token to the next server and returns to the initial state.

When going from state 2 to state 0, the server gives the token to the next server even in the absence of a demand. The purpose of this is to ensure that if its own client makes immediately a new request, it is not served before the other clients have had the chance to go to the critical section.

3 ASSET Model of the System

Figures 3 and 4 show the model of the example system.

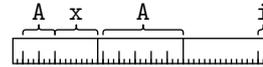
The `#ifdef size_par` structure makes it possible to specify the number of clients and servers via an option that is given to the C++ compiler. Also many features of ASSET can be controlled in a similar way. For instance, with the Gnu

C++ compiler, the options `-Dstubborn -Dsize_par=13 -Dstop_cnt=10000000` command ASSET to use stubborn sets, set $n = 13$, and stop the construction of the state space when 10^8 states are exceeded.

The initialization `C[n] = 2` does not specify that the initial local state of each client is 2 (the critical section) but that two bits are used for representing the local state of each client. The value of each state variable is an unsigned integer in the range $0, \dots, 2^b - 1$, where b is the number of bits. The default value of b is 8. The initial value of each state variable is 0.

Internally, ASSET represents the state of the model as a sequence of unsigned integers. To save memory, ASSET packs many state variables into the same unsigned integer when possible. A `state_var` object does not store the value of the state variable, but information on in which unsigned integer and in which of its bits the value is stored. If the most recently employed unsigned integer has at least as many unused bits as the next state variable needs, then ASSET puts the state variable there. This implies that the order in which the state variables are declared may affect the amount of memory that ASSET uses per state. For instance, assuming 16-bit unsigned integers,

```
state_var x, A[7]=3, i(2);
```



consumes three unsigned integers, while

```
state_var x, i(2), A[7]=3;
```



consumes only two.

The purpose of `c0now` will be explained in Section 6. Until then, please assume that `c0now` is 0.

When ASSET has detected an error, it prints a counterexample in the form of a sequence of states. For this purpose, it needs a `print_state` function. To improve the readability of the counterexamples, the function in Figure 3 uses character encodings for local states. The use of C++ facilitates this and other methods for making the printout look natural for the model. This has been widely exploited in the models on the web page of ASSET.

The function `nr_transitions` in Figure 4 tells ASSET how many *transitions* the model contains. The most common case is that a transition models one or more atomic operations of the system. Transitions in ASSET are deterministic. This implies that nondeterministic operations such as tossing a coin must be modelled by more than one transition. Other than that, ASSET does not restrict the grouping of atomic operations to transitions.

In its initial state, a client of the example system makes a nondeterministic choice between requesting access and terminating for good. For this reason, two transitions are used to model each client. Each server has one transition.

The function `nr_transitions` may also be used for implementing whatever operations are necessary before starting the construction of the state space. In Figure 4, it is used for giving the token to client 1. Client 1 was chosen here, because eventual access will be tested for client 0, so we wanted it to lack the token initially.

```

unsigned nr_transitions(){ T[1] = true; return 3*n; }

inline unsigned next( unsigned i ){ return (i+1) % n; }
inline unsigned prev( unsigned i ){ return (i+n-1) % n; }

bool fire_transition( unsigned i ){

    /* Servers */
    if( i >= 2*n ){
        i -= 2*n;
        #define goto(x){ S[i] = x; return true; }
        switch( S[i] ){
            case 0:
                if( C[i] == 1 || ( S[next(i)] == 1 && !T[next(i)] ) ){ goto(1) }
                return false;
            case 1:
                if( !T[i] ){ return false; }
                if( C[i] == 1 ){ C[i] = 2; goto(2) }
                if( S[next(i)] == 1 ){ T[i] = false; T[next(i)] = true; goto(0) }
                return false;
            case 2:
                if( C[i] == 2 ){ return false; }
                T[i] = false; T[next(i)] = true; goto(0)
            default: err_msg = "Illegal local state"; return false;
        }
        #undef goto
    }

    /* Clients */
    #define goto(x){ C[i] = x; return true; }
    if( i >= n ){ // termination transition
        i -= n;
        if( C[i] == 0 ){ goto(3) }else{ return false; }
    }
    if( C[i] == 0 ){ goto(1) } // request access
    if( C[i] == 2 ){ goto(0) } // leave critical
    return false;
}

```

Fig. 4. Model of the demand-driven token ring, part 2.

The transitions of the model are numbered starting from 0. The function `fire_transition(t)` returns `true` or `false` to indicate whether transition number *t* is enabled. If *t* is enabled, then `fire_transition` modifies the state according to the occurrence of *t*. If *t* is disabled, then `fire_transition` must not modify the state. This rule makes it possible for ASSET to try the next transition without having to upload the state again.

To improve readability, Figure 4 introduces two versions of a `goto(x)` macro. They model the server and the client going to local state *x* and indicate that the transition was enabled.

If $0 \leq t < n$, transition *t* models client *t* going either from local state 0 to local state 1 (that is, requesting access) or from local state 2 to local state 0 (leaving the critical section). If $n \leq t < 2n$, transition *t* models client *t* - *n* going from local state 0 to local state 3 (that is, terminating).

Finally, the transitions $2n \leq t < 3n$ model server *t* - *2n*. It waits in local state 0 until its client requests access or the next server needs the token. For the reason discussed in Section 5, it tests that the next server does not already have the token.

Then it waits in local state 1 until it has the token. If its client has requested access, it moves the client to the critical section and goes to local state 2. Otherwise, if the next server needs the token, server *t* - *2n* gives it to it. Otherwise, server *t* - *2n* continues waiting.

In local state 2, server *t* - *2n* waits until its client has left the critical section. Then it gives the token to the next server and returns to the idle state. As a consequence, its client cannot get access again before the token has circulated through the ring and the other clients have had the chance to get access.

The `default` branch is explained in Section 4.

4 The Checking Features

Figure 5 shows the checking functions used in the experiments of this publication. Each of them can be switched off by commenting out the corresponding `#define`, without having to comment out the function as a whole. This is handy for experimenting. (It would have been nice to use the same word in the `#define` and as the name of the function, but C++ does not allow that.) More flexibility comes from the fact that if `xxx` has not been switched on in the model with `#define xxx`, then it can be switched on at compile time with a compiler option.

ASSET operates in stages. In the first stage, it checks for safety errors and illegal deadlocks (if the checking of them has been switched on). It constructs the state space in breadth-first order, to minimize the length of counterexamples. ASSET calls `check_state` each time when it has constructed a new state, and `check_deadlock` when it has tried to fire transitions in a state but none was enabled. If the state is not good, the function returns a character string. ASSET prints an error message containing it and terminates. That is, ASSET implements on-the-fly detection of safety and illegal deadlock errors. That the state is good is indicated by returning the null pointer 0.

```

/* Check that at most one client is in critical section at any time. */
#define chk_state
const char *check_state(){
    unsigned cnt = 0;
    for( unsigned i = 0; i < n; ++i ){ if( C[i] == 2 ){ ++cnt; } }
    if( cnt >= 2 ){ return "Mutual exclusion violated"; }
    return 0;
}

/* Check that every client has stopped. */
#define chk_deadlock
const char *check_deadlock(){
    for( unsigned i = 0; i < n; ++i ){
        if( C[i] != 3 ){ return "Client not terminated"; }
    }
    return 0;
}

/* Check that the original client 0 eventually gets access
   if it wants to. */
// #define chk_must_progress
bool is_must_progress(){ return C[c0now] != 1; }

```

Fig. 5. The check functions of the model of the demand-driven token ring.

By using a global or static C++ variable, `check_deadlock` (or `check_state`) can be made to count the number of deadlocks that it has detected, and only give the error message for, say, the tenth. Thus, although ASSET does not provide a mechanism for investigating each deadlock in turn, C++ makes it possible.

If ASSET did not detect any errors and if it has further checks to perform, it constructs a data structure that contains the edges of the state space in the reverse direction. To do that, it goes through all states that it has found and fires the transitions again in them. (In the case of stubborn sets, it fires the same subsets of transitions.) In this way, only one unsigned integer per edge is needed. Storing the edges during the first stage and sorting them afterwards would have used two unsigned integers per edge.

Then, if `chk_may_progress` is on, ASSET checks the state space for *may progress errors* by performing a linear-time search along the reversed edges. A may progress error is a reachable state from which no terminal state and no state accepted by `is_may_progress` is reachable. May progress can be thought of as a less stringent alternative to linear-time liveness that does not need fairness assumptions. This feature has been discussed extensively in [10] and is not used in the experiments of the present publication, so it will not be discussed further here.

Next ASSET checks the state space for *must progress errors*, if it has been commanded to do so and it has not yet terminated because of another error. A

must progress error is a cycle in the state space that does not contain any state accepted by `is_must_progress`. This is a restricted form of checking linear-time liveness. For reasons discussed in the next two sections, the simultaneous use of this feature with stubborn sets and symmetries is limited. Therefore, it has been switched off in Table 1.

Outside Table 1, the `is_must_progress` function in Figure 5 was switched on in some experiments. ASSET found no errors in the model in Figures 3 and 4. Also a modified model was used where, when leaving local state 2, instead of giving the token to the next server and going to local state 0, the server goes to local state 1. ASSET reported that this model has a cycle where a requesting client does not get access. In it, another client leaves the critical section, requests for access again, and gets access again. The output of ASSET is shown below with explanatory comments. (ASSET does not minimize the length of counterexamples that contain a cycle.)

```

-i -i*      initial state
=====    after this line, must progress fails for a pending request
Ri -i*     client 0 requests
Rw -i*     server 0 waits for token
-----    after the last line, the counterexample jumps back here
Rw -w*     the demand reaches server 1
Rw Rw*    client 1 requests
Rw Ct*    client 1 gets permission
Rw -t*    client 1 leaves the critical section
!!! Must-type non-progress error
46 states, 66 edges

```

Finally, if the stubborn set method is used, safety or progress was checked, and ASSET has not yet found any error, it checks that the state space is always may-terminating in the sense discussed in the next section.

The model may at any time assign a character string to `err_msg`. It causes ASSET to terminate and print an error message containing the string. This feature is not intended for specifying correctness properties, but for catching inconsistent situations within the model. In Figure 4 it is used in the `default` branch of the `switch` statement, to indicate that the modeller believes that the branch is never entered.

In addition to the memory needed for the state itself, ASSET uses two or five unsigned integers per state, depending on whether the verification task involves graph search operations in the state space. To quickly detect whether a state is new or has been constructed also earlier, there is a hash table whose base table uses 2^h unsigned integers, where h is 23 by default but can be changed. The stubborn set method consumes an additional $5t$ unsigned integers, where t is the number returned by `nr_transitions`.

Therefore, in theory, ASSET uses approximately $((2 + s)n + 2^h)u$ bytes for the easier and $((5 + s)n + m + 2^h)u$ or $((5 + s)n + m + 2^h + 5t)u$ bytes for the more difficult verification tasks, where n is the number of constructed states,

```

void next_stubborn( unsigned i ){

    if( i >= 2*n ){
        i -= 2*n;
        switch( S[i] ){
            case 0: if(
                C[i] == 1 || ( S[next(i)] == 1 && !T[next(i)] )
            ){ return; }
                stb(i, next(i)+2*n); return;
            case 1: if( !T[i] ){ stb(prev(i)+2*n); return; }
                if( C[i] == 1 ){ return; }
                if( S[next(i)] == 1 ){ stb(i); return; }
                stb(i, next(i)+2*n); return;
            case 2: if( C[i] == 2 ){ stb(i); }
                return;
            default: return;
        }
    }

    if( i >= n ){ stb(i-n); return; }
    switch( C[i] ){
        case 0: stb(i+n, i+2*n); return;
        case 1: stb(i+2*n); return;
        case 2: stb_all(); return;
        default: return;
    }
}

```

Fig. 6. The stubborn set obligation rules of the demand-driven token ring.

m is the number of constructed edges, s is the number of unsigned integers used for representing a state, and u is the number of bytes in an unsigned integer (usually $u = 4$ or $u = 8$). These formulae correctly predict the numeric and “_” entries in Table 1. However, because of how the dynamically growing arrays of C++ work, the real memory consumption may add almost $(2 + s)nu$ or $(5 + s)nu$ bytes to this. With 13 servers and clients, the stubborn set method yields $13 \cdot 3\,777\,949 = 49\,113\,337$ states. This would otherwise fit the memory, but the doubling of the arrays at $2^{25} = 33\,554\,432$ states causes a memory overflow, explaining the “..” entries in Table 1.

5 The Stubborn Set Method in ASSET

The implementation of the stubborn set method in ASSET is discussed extensively in [10]. Therefore, it is discussed here only briefly.

Only the basic strong stubborn set method that preserves the deadlocks is implemented. However, theorems in [10] tell that if the model is always may-

terminating — that is, if from every reachable state, a terminal state is reachable, then the basic strong stubborn set method preserves also safety and certain progress properties. Furthermore, whether the model is always may-terminating can be checked from the reduced state space.

To use the method, the function `next_stubborn` must be provided. It represents state-dependent rules of the form “if this transition is in the stubborn set, then also these transitions must be”. Figure 6 shows the rules used in the experiments reported in Table 1.

The present author did not at first realize the necessity of the part `&&!T[next(i)]` in case 0 in Figure 4. When it was lacking, the plain and symmetry methods did not give any error messages. Indeed, mutual exclusion and eventual access are not violated. However, thanks to the check that the model is always may-terminating, the stubborn set method gave the following when $n = 2$.

```
-i -i*
-i i*
=====
Ri i*
Rw i*
Rw w*
Rw* i
Rw* w
Ct* w
-t* w
-i w*
-----
i w*
w w*
w* i
w* w
!!! State was reached from which termination is unreachable
67 states, 93 edges
```

In it, client 0 visits the critical section and both clients terminate. Because waiting information may be propagated from server $i \oplus 1$ to server i even if the former has the token, unnecessary waiting information enters the ring. Eventually the model runs in a cycle where unnecessary waiting information circulates in one direction and, driven by it, the token circulates in the opposite direction. The cycle consists of the states below the “-----” mark. Reaching a terminal state is impossible after the “=====” mark. So the first erroneous state is `Ri i*`.

The stubborn set implementation in ASSET does not guarantee that must progress errors are found. If must progress is used with stubborn sets and ASSET finds no errors, then it gives a warning that the pass verdict is unreliable. With the modified model discussed in Section 4, ASSET did find the error with stubborn sets switched on. With $n = 8$, there were 2 472 336 states and 17 539 200 edges without and 163 264 states and 293 984 edges with stubborn sets.

```

void symmetry_representative(){
    unsigned i = 0;
    while( !T[i] ){ ++i; } // find the server with the token
    i = prev(i);
    if( !i ){ return; } // terminate, if the state maps to itself
    unsigned A[n], j;
    for( j = 0; j < n; ++j ){ A[j] = C[(i+j) % n]; }
    for( j = 0; j < n; ++j ){ C[j] = A[j]; }
    for( j = 0; j < n; ++j ){ A[j] = S[(i+j) % n]; }
    for( j = 0; j < n; ++j ){ S[j] = A[j]; }
    for( j = 0; j < n; ++j ){ A[j] = T[(i+j) % n]; }
    for( j = 0; j < n; ++j ){ T[j] = A[j]; }
    #ifdef symm_must
    cOnow = (cOnow + n-i) % n;
    #endif
}

```

Fig. 7. The symmetry representative function of the demand-driven token ring.

6 The Symmetry Method in ASSET

Many systems contain similar components organized in a symmetric fashion. Several authors have suggested exploiting the symmetry for reducing the size of the state space, including [2, 3, 5].

The implementation of the symmetry method in ASSET is very simple. Unfortunately, as we will see, it leaves a lot of responsibility to the modeller or preprocessor tool.

Most importantly, the modeller or preprocessor must provide a function `symmetry_representative` that maps each state to a symmetric state. The more states are mapped to the same state, the better are the reduction results. Ideally, all states that are symmetric to each other are mapped to the same state. However, the method remains correct even if the function is not ideal in this respect.

If the symmetry method has been switched on, ASSET calls the function `symmetry_representative` on the initial state and on each result of a successful firing of a transition. As a consequence, paths in the reduced state space may contain *symmetry hops*, that is, the head state of an edge is not necessarily the real result of firing the transition in question in the tail state of the edge. Instead, it may be another state that is symmetric to the real result.

Figure 7 shows the symmetry mapping used in the experiments of this publication. It first finds the server that has the token, and then rotates the ring so that the found server becomes server 1.

The modeller or preprocessor must take the symmetry mapping into account when formulating the checked properties. Because each of `check_state` and `check_deadlock` analyses a single state, it is not difficult to make them give the same reply on symmetric states. The versions in Figure 5 do so.

It is more difficult with progress properties. For instance, consider the eventual access property “if client 0 wants to go to the critical section, it eventually gets there”. When `symm_must` is off, the `is_must_progress` function in Figure 5 formulates the property in a manner that is appropriate only when the symmetry method is not used. Because the symmetry mapping in Figure 7 always rotates the system such that server 1 has the token, only client 1 ever gets to the critical section in the symmetry-reduced state space. However, the rotation does not prevent client 0 from trying to go to the critical section. What happens is that just when client 0 is about to get to the critical section, it becomes client 1. So ASSET incorrectly reports that client 0 tried to go to the critical section but never got there.

To solve this problem, the state variable `cNow` was added to the model. It keeps track of the current number of the original client 0. The verification results became correct, but the reduction in the size of the state space was lost entirely. This is a problem of not just ASSET, but the symmetry method in general.

The counterexamples printed by ASSET may contain symmetry hops. However, in the experience of the present author, they have not harmed the interpretation of counterexamples. They may even be helpful. When a counterexample contains many symmetric copies of the same theme, symmetry hops may reduce them to a single copy.

Acknowledgement. I thank the anonymous reviewers for helpful comments.

References

1. Brookes, S. D. & Hoare, C. A. R. & Roscoe, A. W.: A Theory of Communicating Sequential Processes. *Journal of the ACM* 31, 3 (1984) 560–599
2. Clarke, E. M. & Filkorn, T. & Jha, S.: Exploiting Symmetry in Temporal Logic Model Checking. In: Courcoubetis, C. (ed.) *Computer-Aided Verification '93*, Lecture Notes in Computer Science 697 (1993) 450–462
3. Emerson, E. A. & Sistla, A. P.: Symmetry and Model Checking. In: Courcoubetis, C. (ed.) *Computer-Aided Verification '93*, Lecture Notes in Computer Science 697 (1993) 463–477
4. Holzmann, G. J.: *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley (2003) 596 p
5. Jensen, K.: *Coloured Petri Nets, Volume 2, Analysis Methods*. Monographs in Theoretical Computer Science, Springer (1995) 174 p
6. Manna, Z. & Pnueli, A.: *The Temporal Logic of Reactive and Concurrent Systems – Specification*. Springer-Verlag (1992) 427 p
7. Roscoe, A. W.: *Understanding Concurrent Systems*. Springer, Heidelberg, Germany (2010) 533 p
8. Valmari, A.: *Error Detection by Reduced Reachability Graph Generation*. In: Proceedings of the 9th European Workshop on Application and Theory of Petri Nets (1988) 95–122
9. Valmari, A.: *The State Explosion Problem*. In: Reisig, W. & Rozenberg, G. (eds.) *Lectures on Petri Nets I: Basic Models*, Lecture Notes in Computer Science 1491 (1998) 429–528

10. Valmari, A.: *Stop It, and Be Stubborn!* In: Haar, S. & Meyer, R. (eds.) 15th International Conference on Application of Concurrency to System Design, IEEE Computer Society (2015) 10–19, DOI 10.1109/ACSD.2015.14
11. Valmari, A. & Setälä, M.: Visual Verification of Safety and Liveness. In: Gaudel, M.-C. & Woodcock, J. (eds.) *Formal Methods Europe '96: Industrial Benefit and Advances in Formal Methods*, Lecture Notes in Computer Science 1051 (1996) 228–247

n		states	edges	time		states	edges	time
2	P	68	140	0.0	S	44	60	0.0
	Y	34	70	0.0	B	22	30	0.0
3	P	468	1 350	0.0	S	219	327	0.0
	Y	156	450	0.0	B	73	109	0.0
4	P	2 928	10 880	0.0	S	920	1 432	0.0
	Y	732	2 720	0.0	B	230	358	0.0
5	P	17 280	78 600	0.1	S	3 505	5 625	0.0
	Y	3 456	15 720	0.0	B	701	1 125	0.0
6	P	98 064	527 760	0.2	S	12 540	20 772	0.1
	Y	16 344	87 960	0.1	B	2 090	3 462	0.0
7	P	541 296	3 364 200	0.8	S	43 015	73 899	0.2
	Y	77 328	480 600	0.4	B	6 145	10 557	0.1
8	P	2 927 232	20 632 320	4.5	S	143 408	256 880	0.4
	Y	365 904	2 579 040	1.6	B	17 926	32 110	0.2
9	P	15 583 104	122 821 920	30.0	S	469 053	879 885	1.4
	Y	1 731 456	13 646 880	10.0	B	52 117	97 765	0.3
10	P	81 933 120	714 052 800	262	S	1 514 900	2 984 860	4.6
	Y	8 193 312	71 405 280	59.5	B	151 490	298 486	0.9
11	P	–	–	341	S	4 852 771	10 057 839	16.3
	Y	38 771 136	370 202 400	339	B	441 161	914 349	2.6
12	P				S	15 464 040	33 719 400	60.1
	Y	–	–	1039	B	1 288 670	2 809 950	9.1
13	P				S	65.0
	Y				B	3 777 949	8 659 221	30.0
14	P				S			
	Y				B	11 116 762	26 741 542	96.1
15	P				S			
	Y				B	32 826 001	82 708 765	353
16	P				S			
	Y				B	131

Table 1. Results on the demand-driven token ring. “P” = plain, “S” = stubborn sets, “Y” = symmetries, and “B” = both. “–” denotes that 10^8 states was exceeded. “..” indicates memory overflow. The times are in seconds and do not include the compilation. The experiments were run on a year 2009 laptop with a 1.6 GHz dual core processor and 1.954 GiB (that is, 2.098 GB) of RAM.

Semantics analyzing expression editors in IP-XACT design tool Kactus2

Mikko Teuhu, Esko Pekkarinen, Timo D. Hämäläinen

Tampere University of Technology, Tampere, Finland
mikko.teuhu@tut.fi, esko.pekkarinen@tut.fi,
timo.d.hamalainen@tut.fi

Abstract. This paper presents parameter and expression editors of the design tool Kactus2. It is aimed at digital System-on-Chip (SoC) designs based on IEEE 1685 IP-XACT XML metadata standard. SoC's are constructed by assembling parametrized components using generators for hardware language code and design configuration. The key challenges are the management of dependencies between thousands of parameters, as well as immediate validation and evaluation while editing. The expression editor in this paper has been designed to overcome these challenges. The editors include real-time syntax, semantic analysis and the use of UUIDs behind user displayed parameter names. The implementations for these have been published in Kactus2 v2.8 open source code, written in C++/Qt5, and consisting of 3000 LoC in the release. An independent industrial user on the SoC domain has verified the correctness, completeness and usability of the new solutions. The designed editors significantly improve the SoC parameter editing and design configuration.

Keywords: IP-XACT · Kactus2 · Electronic Design Automation · System-on-Chip · Expression analysis · Semantic analysis

1 Introduction

This paper presents new features and implementation solutions for a design tool called Kactus2. It is aimed at digital systems design, especially System-on-Chip (SoC) designs which can include hundreds of Intellectual Property (IP) blocks possibly representing millions of lines of hardware description language (HDL) code [1]. Kactus2 uses IEEE standards 1685-2009 and 1685-2014 “IP-XACT” XML metadata that is aimed at unified specification for electronic design automation, IP vendors, and system design communities [2]. IP-XACT specifies how IP blocks are packaged and assembled independent of the design tool, implementation language or vendor used in creating the IP block [3]. IP-XACT 2014 templates include 24 XSD files, totaling of 790 elements and 241 attributes. In addition, so called vendor extensions can multiply these numbers, which shows that IP-XACT is a complex specification. It has been widely adopted in VLSI industry recently accompanied by FPGA vendors.

SoC designs are increasingly based on automatic generation of HDL code, HW dependent SW device drivers and automated building of the SW stack. IP-XACT at-

tempts to specify a standard for all of this by a single point of control. The main HW related tasks are parametrization of the IP blocks, instantiation and connection of the blocks and design configuration.

Parameters are heavily used e.g. to construct structures and memory regions accessible to SW. Parameters can be propagated through hierarchy, but global references are not allowed to guarantee reuse of each IP block. Parameters defined in an IP-block can be overridden for each instance in a design. There are six parameter attributes like strict typing or allowable numerical range for the different purposes. These can be used to explicitly define the values that a parameter can be given.

The challenge is how to manage even thousands of parameters and attributes that may depend on each other and form long chains in expressions. Since the user cannot comprehend the whole system at once, a tool should validate, and when possible, evaluate parameter values on-the-fly while editing. A mistake in referencing, renaming or copy-pasting can be very difficult to find afterwards.

Another challenge is how to enter the parameter information in a tool. Some related tools resemble XML code editors, and some offer very large table sheets. Both are prone to human errors and reduce the usability. An XML code should not be visible for a system designer.

The need for an open source tool easing the reuse, configuration and assembly of SoC designs motivated the development of Kactus2 at Tampere University of Technology. Even stronger motivation was to create an outstanding user interface hiding the complexity of IP-XACT, since the tools in this domain are commonly known to be very difficult to use.

Kactus2 was launched in 2010. It is implemented in C++ and Qt5, and the current release v2.8 has ~390k LoC. Kactus2 is actively being used in several companies, which also have supported the development. There are over 7800 installer downloads by Aug 2015, which is a very good number in this specific domain. Direct feedback from the users confirms the good usability.

This paper presents new solutions developed for Kactus2 for SoC design configuration, which were implemented in v2.8. The new contributions are following:

- A new approach to expression editors.
- Real-time validation and semantic analysis of the parameters involved in defining the high-level SoC design configuration.

This paper is organized in the following way. Chapter 2 discusses the related work of expression editors. Chapter 3 introduces the solutions for the expression editor of Kactus2. Chapter 4 contains the evaluation of the created mechanics. Chapter 5 concludes this paper.

2 Related work

The early IP-XACT tools were Eclipse-based XML editors, which are still available for many variations as commercial tools. The closest free tool to Kactus2 is EDATools IP-XACT Solution [4], but it has much less features and is basically an IP-XACT

XML editor. All true competitors are commercial tools, but they deny any comparisons and publication of the user interface in their licensing terms. In addition, commercial tools have often been customized, which complicate comparison. Thus, Kactus2 is currently the only usable open source IP-XACT tool.

For the related work, we consider more common solutions outside IP-XACT domain. The most important part is an expression editor that should

1. Have a WYSIWYG editor,
2. Support basic mathematical operators,
3. Support nested sections in parentheses,
4. Have constants, parameters and operators adding, copying, pasting and moving within and between expressions,
5. Offer only existing parameters to be inserted and rejecting any other strings,
6. Validate the formula when typing,
7. Evaluate the value of the expression immediately,
8. Hide the parameter IDs and display only parameter names to the user.

The parameter references are based on unique IDs (UUID), which means the parameter name can be changed without breaking the dependencies. In addition, when two IP blocks are integrated with the same parameter name but different meaning, UUIDs help resolving the issue.

Marques et al. have presented WIRIS OM Toolset [5] for managing mathematical expressions. This application tries to construct outputs of mathematical fragments written in a suitable markup language. WIRIS OM Toolset has been incorporated into the LeActiveMath and WebALT project [5]. LeActiveMath is a learning tool system designed for high school, college or university level teaching [6]. WebALT is an application that uses existing standard to represent mathematical equations on the web together with existing linguistic technologies in order to create a language-independent mathematical learning platform [7]. The TextMathEditor of WebALT allows the construction of mathematical expressions using a predetermined grammar. The WIRIS Formula Editor of the WIRIS OM Toolset has been incorporated within this math editor.

Lee et al. have developed MathCast [8], an open source equation editor for mathematical expressions. The produced equations can be used in documents, graphically rendered picture files or within MathML Presentation 2.0, format for describing mathematics [9]. A tool is included within MathCast to author XHTML web pages with mathematical equations.

Design Science Inc. offers MathType, a commercial equation editor tool for Windows and Macintosh [10]. It contains a wide range of mathematical functions and customization options, with an ability to export these equations to Mathematical Markup Language (MathML). MathType also understands the TeX and LaTeX typesetting. MathType is compatible with a multitude of applications & websites, such as Adobe InDesign and Microsoft Office tools.

Design Science Inc. offers another commercial mathematical equation tool, called MathFlow [11]. MathFlow is a MathML Toolbox standard, offering tools for editing, displaying and accessing mathematical notations on websites, applications and ser-

vices. It can be incorporated into a suite of other software, such as Oxygen [12]. The MathFlow toolbox consist of editors, equation composers and document composers.

All related editors have features from 1 to 7, e.g. WYSIWYG, use of previously constructed constants, parameters and operators and nesting. MathType also supports handwriting but this is not important for SoC design. However, none of the editors support our requirement 8. We also need to consider how operands and operations are defined for the expression editor.

2.1 Related standards

There are two standards for capturing mathematical equations: OpenMath [13] and MathML [9]. There is a large overlap between these two communities. Both attempt to create an extensible standard for representing the semantics of mathematical objects. As a web based system, MathML is based on XML in order to help browsers natively render mathematical expressions. OpenMath endorses XML and binary formats, although a custom encoding can be used. It is primarily targeted towards the semantic meaning and content of mathematical objects instead of focusing on representing the objects. This approach is used in MathML. However, OpenMath has received critique for its use of general mathematics [14].

Since both the MathML and OpenMath are based on XML [9] [13], it could be incorporated within Kactus2. Although these could help in displaying the constructed equations within the expression editor, the semantic analysis is the important part of the IP-XACT standard. Both of these standards contain semantic tags for constructing the expressions. Following the structure, an analysis can be formed. For example the following example sentence

$$x^2 + 4x + 4 = 0 \tag{1}$$

could be constructed using the MathML semantics[9] as follows:

```

1.    <apply>
2.        <plus/>
3.    <apply>
4.        <power/>
5.        <ci>x</ci>
6.        <cn>2</cn>
7.    </apply>
8.    <apply>
9.        <times/>
10.       <cn>4</cn>
11.       <ci>x</ci>
12.    </apply>
13.    <cn>4</cn>
14. </apply>

```

Since these standards describe the expressions in XML format, parsing one sentence for editing requires resources. Comparing to the expression editor developed for Kactus2, an expression is contained within a single string-variable. The managing of

the equation becomes resource consuming as the XML tags of the expression need to be re-applied every time the user wants to replace a value. Displaying the data of a table containing tens of parameters each containing between one to three expressions could possibly cause long loading times in opening and changing tables.

Regardless of the merits of existing solutions, none of these are easy to integrate into Kactus2. The equation editors presented here are applications, and removing the functionality of expressions from them could prove difficult. Kactus2 operates using IP-XACT standard defining the parameters that are used in its expressions. These expressions are structured according to System Verilog. To ensure that the parameter references can be handled properly, together with a reliable and immediate expression validation and evaluation, a custom expression editor was implemented for Kactus2. The editor would be difficult to implement in other, non-IP-XACT applications.

3 Equations in Kactus2

Expressions are used in Kactus2 to capture mathematical equations and parameter references. The references are constructed using the unique identifier of the referenced value.

3.1 Expressions in Kactus2

In IP-XACT, the expressions are stored in an XML file. The XML element depends on the correct location of the expression. Following is a fragment of IP-XACT file of a *component* containing parameters. The `**` is used as an operator for exponentiation:

```
<spirit:parameter type="int" usageCount="2">
  <spirit:name>port_width</spirit:name>
  <spirit:value spirit:id=
    "uuid_4087e902_e070_460c_b14f_41445660d950">-2
  </spirit:value>
</spirit:parameter>
<spirit:parameter type="real">
  <spirit:name>clk_enable</spirit:name>
  <spirit:value spirit:id=
    "uuid_0e46c746_ce3d_445b_977a_f9737eb3c505">
    uuid_4087e902_e070_460c_b14f_41445660d950**2+4*
    uuid_4087e902_e070_460c_b14f_41445660d950+4
  </spirit:value>
</spirit:parameter>
<spirit:parameter type="int">
  <spirit:name>port_range</spirit:name>
  <spirit:value spirit:id=
    "uuid_c45c68c9_1a57_4be6_a4e6_5e73ea74f197">14
  </spirit:value>
</spirit:parameter>
```

This example displays three parameters, with parameter *clk_enable* containing a similar equation as in equation (1). In this example, the string *uuid* followed by a list of characters is a universally unique identifier (UUID) [15] of a parameter value. Every parameter value within Kactus2 has this identifier. When an XML file is read, Kactus2 checks if a parameter contains a unique identifier. If this identifier is not found, a new UUID is created.

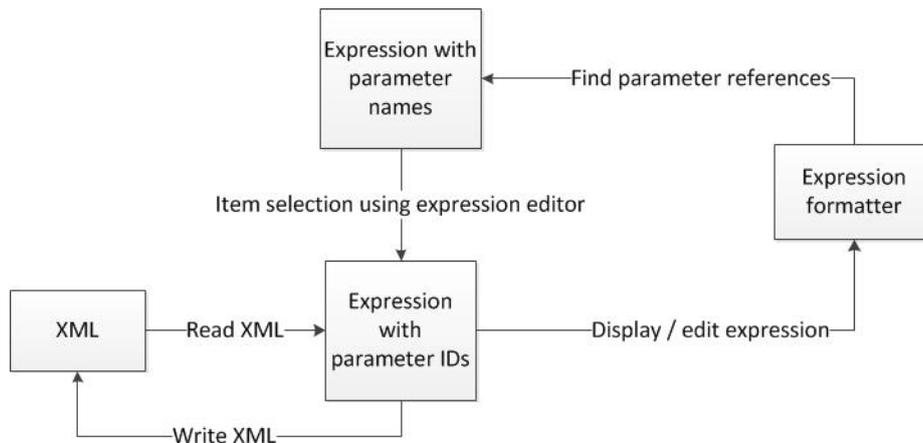


Fig. 1. Expression reading and displaying in Kactus2

The flowchart for handling of XML files in Kactus2 is shown in **Fig. 1**. As shown above, the XML file contains the references of an expression as unique identifiers to the referenced parameter value. When an expression is displayed in a table of Kactus2, the expression formatter of the table changes the unique identifier to a more human readable name of the parameter. Only the name is displayed for user editing the expressions.

3.2 The expression editor

A section of the *Parameters editor* of Kactus2 is shown in **Fig. 2**. This editor is used to create and manage the parameters of a component. The name column displays the parameter name, while the type column describes the type of the value. If the selected type cannot contain the given value, the type is displayed in red.

Name	Description	Data type	Type	Value, $f(x)$	Choice
port_width			int	-2	
port_range			int	14	
clk_enable			real	$\text{port_width}^2 + 4 * \text{port_width} + 4$	0

Fig. 2. Parameters editor of Kactus2

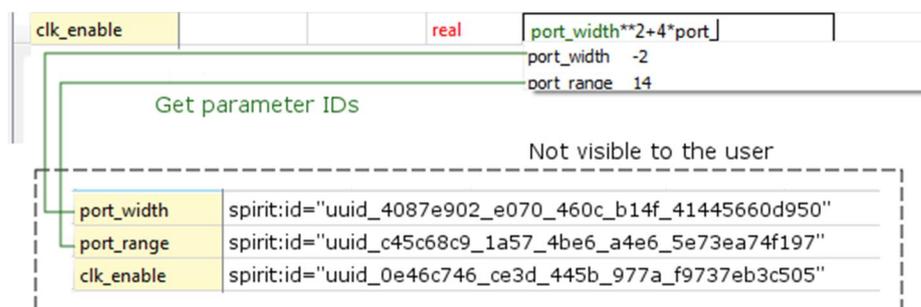


Fig. 3. Reference selection in the expression editor.

The symbol $f(x)$ in the header informs that the value can be given as an expression. The value of the expression is displayed as a tooltip of the containing field. The expression itself is visible when viewing or editing a parameter. The parameter values can also be given as arrays. Every value of an array can be given a different expression, but multidimensional arrays are not supported in Kactus2.

While constructing an expression, the Kactus2 expression editor offers a list of possible completions for an incomplete word. This can be observed in **Fig. 3**, where the value of parameter *clk_enable* is being edited. The available selections are determined by the location of the expression editor. In the **component editor**, these completions are selected from the parameters of the currently active component. In the **configurable element editor**, the completions are selected from the configurable element values.

Each selection consists of the name of the corresponding parameter and its value, while the unique identifier of the parameter is hidden from the user. When a completion is selected, the expression editor retrieves this unique identifier and inserts it to the edited value. Thus the expression itself is constructed using the unique identifiers of the referenced parameters. When the finished expression is displayed to the user, the **expression formatter** changes the unique identifier of the referenced parameter to the name of the parameter.

The expression editor will colour green the valid parameter references of an expression. Values containing invalid references are coloured red. All the references are made using the unique IDs of parameters, but are displayed using the parameter names. This is because the parameter IDs are not human readable.

The selection management is handled by the **parameter completer** class. **Fig. 4** displays the structure of this completer. This class creates the available selections for the user when selecting a completion to an unfinished word, as seen in **Fig. 3**.

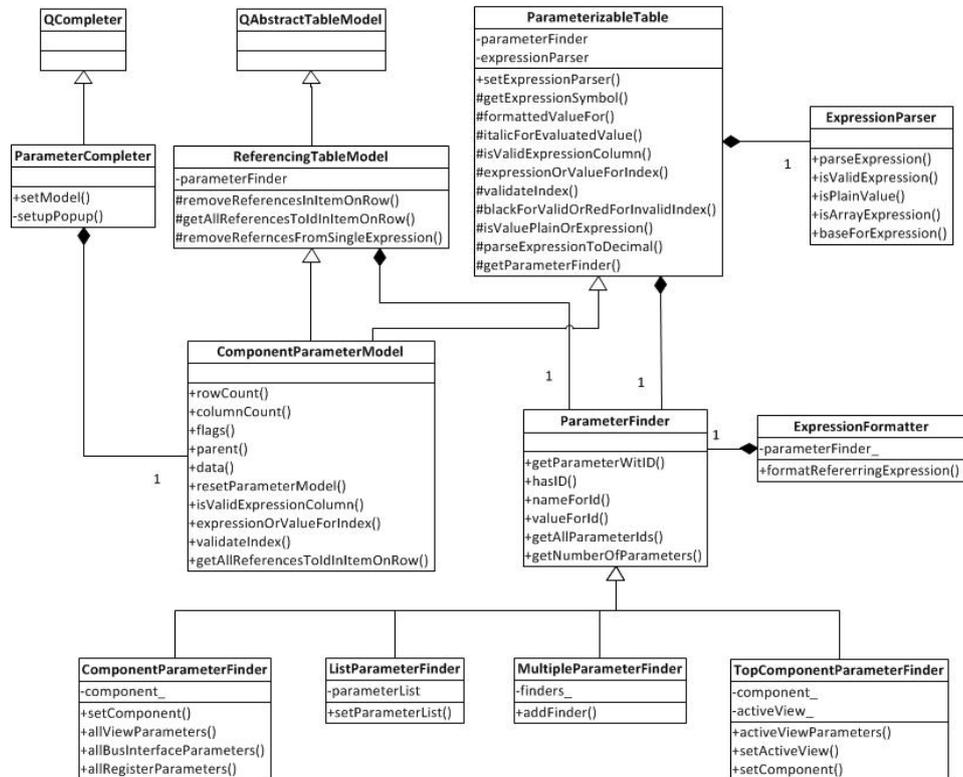


Fig. 4. The structure of **parameter completer** class in the expression editor.

The parameter completer is derived from the `QCompleter` class of Qt, and uses a component parameter model to construct the model seen in the selection. This model retrieves the parameters according to its currently selected **parameter finder** class. It is based on the table editors of Kactus2 and thus is derived from both the **referencing table model** class and the **parameterizable table** class. These handle the basic functionalities of a table view in Kactus2. An **expression parser** class is attached to the **parameterizable table** class. This parser evaluates the values of expressions. The **parameter finder** class is used to find referable parameters, with its subclasses determining the place where these parameters are found.

In version 2.8 of Kactus2, the classes contributing to the expression editor consist of approximately 3000 lines of C++/Qt code. This includes the expression managing and parsing, reference selection and expression formatting.

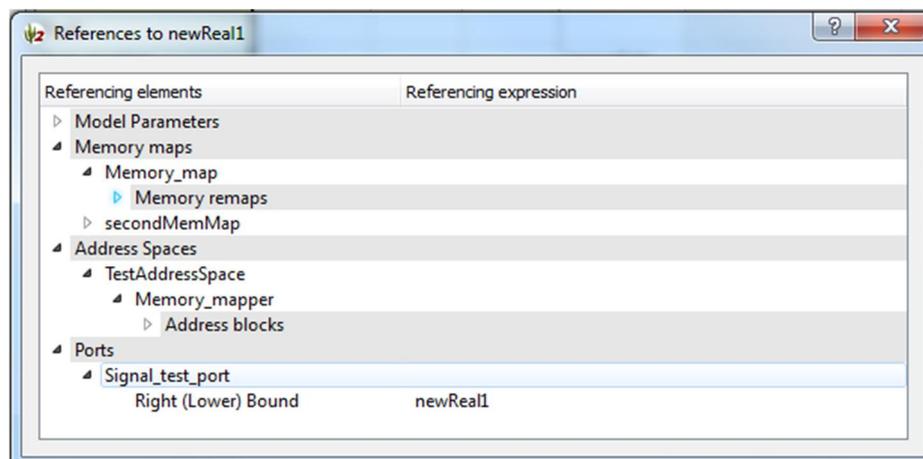


Fig. 5. View of the parameter reference tree.

3.3 Parameter reference tree

IP-XACT based designs can include very complex parameter reference chains. To help managing this, Kactus2 offers a parameter reference tree view with the expression editor. The tree visualizes all of the locations where the parameter has been referenced within the component together with the expression, in which the parameter has been referenced. This reference calculation helps the user to be informed of the importance of a parameter. A warning is given if a user tries to remove a parameter that has been referenced at least once. An example of the reference tree is depicted in Fig. 5.

The parameter reference tree is constructed on demand if a parameter has been referenced in an expression. This is determined by the usage amount column of a parameter. Keeping this usage count constantly in a correct state requires the tracking of all the expression editors of Kactus2.

Whenever a reference is selected in the expression editor, a signal is sent to the **parameter reference calculator** class. This signal informs the reference calculator class to increase the usage count of the referenced parameter. When a reference is deleted or changed, a signal is sent to inform the reference calculator to decrease the amount of references made to the parameter.

3.4 Editing an expression

When an expression is edited, the expression editor tracks the current location of the editing. When the user removes or inserts characters into the expression, this location is used to determine what is in the current position. If the position contains a reference to a parameter value, a signal is emitted to inform that the reference has been removed and the usage count of that parameter should be decreased, thus affecting the parameter reference tree.

When the user changes location in the expression, the **parameter completer** attached to the expression editor displays a list of possible references to complete the word at the current location.

If a new reference is not selected for the edited word, the XML file containing the edited parameter value is changed. The uuid of the referenced value is replaced with the edited name of the parameter. For example this XML contains parameter `clk_counter`. Its value is an expression containing a reference to parameter `port_range`:

```
<spirit:parameter type="int">
  <spirit:name>clk_counter</spirit:name>
  <spirit:value spirit:id=
    "uuid_5ecaeb33_fd7f_4512_acbf_5684d504af02">
    uuid_c45c68c9_1a57_4be6_a4e6_5e73ea74f197+4
  </spirit:value>
</spirit:parameter>
```

The uuid of parameter `port_range` is displayed in the value element of the parameter `clk_counter`. When the value of the `clk_counter` is edited by removing a character, the uuid is replaced by the edited name of parameter `port_range`:

```
<spirit:parameter type="int">
<spirit:name>clk_counter</spirit:name>
  <spirit:value spirit:id=
    "uuid_5ecaeb33_fd7f_4512_acbf_5684d504af02">
    port_rage+4
  </spirit:value>
</spirit:parameter>
```

The analysis of a modified expression is performed after the editing is finished. The new expression is evaluated and the results are displayed in the tooltip of the finished expression.

3.5 Semantic analysis of an expression

Kactus2 follows the System Verilog syntax for evaluating expressions. The validity of an expression is handled in the expression parser classes. Invalid expressions are colored red and the results are given as a string containing the text N/A.

Regular expressions are used to determine the validity of an expression. These define the available operators and functions that can be given in the expressions of Kactus2. A valid expression within the expression editor of Kactus2 contains at least one operator or a string literal. These operators can be negative or positive. Expressions can contain more operators, but they must be connected with mathematical operands. String literals are accepted as valid expressions.

In addition to the regular expression containing the form of the equation, the parentheses and braces within the expression are examined. Parentheses are used to con-

struct equations with priority calculations. Valid places for open parentheses are before the first operand or before any subsequent operands. The closing parentheses are placed after any subsequent operands. To handle the construction of multiple parentheses containing functions, the expression parser checks if the expression contains the same number of opening and closing parentheses.

Braces identify a value containing an array of values. A valid array contains an equal number of opening and closing braces. A value within an array can be constructed similarly as a basic value, i.e. it can contain expressions. Multidimensional arrays are not supported in the current version of Kactus2.

The expression editor is compatible with the decimal, hexadecimal, octal and binary number formats. The basic functionalities of the *ExpressionParser* class of Kactus2 parses any given expression to a decimal number. The *ValueFormatter* class allows formatting any expression to a desired number format.

3.6 Expression evaluation

The evaluation of the expressions in Kactus2 is performed by the *ExpressionParser* class and its subclasses. The flowchart for expression analysis is shown in **Fig. 6**. The expression parser starts by checking any given expression for its validity. An N/A is returned if an expression is found to be not applicable in Kactus2. It then checks if the given expression is an array of expressions. If it is an array, the expression parser evaluates all the expressions contained within it.

The equation is divided into a list of string type units. Each unit contains either an operator, or a word delimiter. Using this list of units, the expression parser calculates the value for the given expression using the standard arithmetic formulas.

Parentheses used in the expression are solved in a similar manner to the mathematical functions. The beginning and ending parentheses are first matched. Then each of the expressions contained within the parentheses is then calculated separately, beginning with the innermost expression.

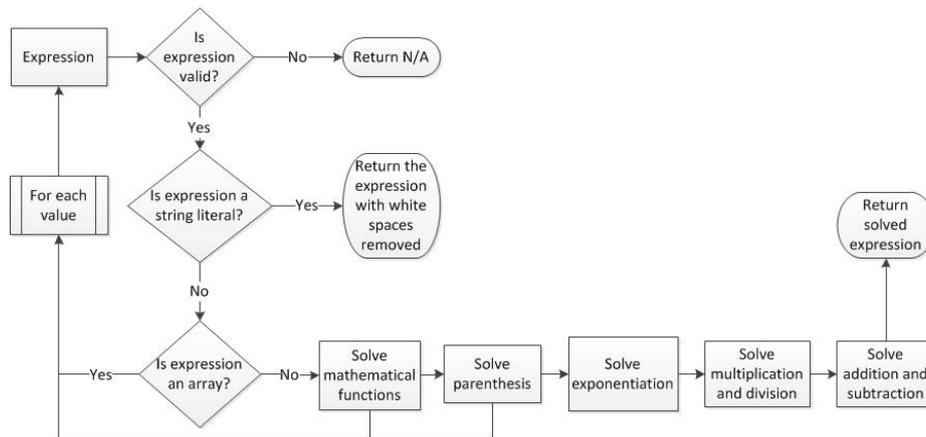


Fig. 6. The flowchart for expression analysis used in the expression editor of Kactus2.

4 Evaluation of the equation mechanics

The semantic analysis of Kactus2 handles the basic functionalities of a mathematical equation editor. Comparing to other software designed to create and manage complex mathematical structures, the expression editor of Kactus2 may seem simple. Kactus2 does not support the drawing of mathematical functions, or some the basic functionalities such as calculating square roots. However, the expression editor is currently purposed to handle the mathematical needs of IP-XACT based SoC designs.

The speed of the expression editor depends on the amount of parameters currently contained within a component. A test was conducted on a Win7 operating system and 4 processors (x86-64, WOW64) with a component containing approximately a hundred parameters. Writing an expression causes no lag. Less than a second of lag can be perceived in constructing the list of selectable references. Editing an expression does not incur any more lag than the construction of the reference selections when editing sections of the expression. The expression editor presented in this paper follows the System Verilog structure to validate and analyze any expressions given to it. This structure contains the `clog2` (ceiling of \log_2) function.

The standards OpenMath and MathML are not necessary to be implemented in Kactus2. The expressions in Kactus2 are stored in simple string variables compared to the XML tags used in both of these standards. Accessing the correct XML tag to change a single operator of an expression can be considered more resource consuming, compared to the handling of expressions contained a string. Additionally, as Kactus2 is developed for SoC design, the expression editor contained within it does not need very complex mathematical functions.

5 Conclusions

The Kactus2 expression editor presented in this paper allows the creation of mathematical expressions displaying parameter names to the user while hiding the references made to the parameter IDs. These parameters can be located within different parts of a component. The different parameter finders of Kactus2 are used to specify the location of the usable parameter references. This can be either the currently active IP-XACT component, the component referenced by the currently active IP-XACT component instance, the configurable elements of an IP-XACT component instance or the top component containing a design housing the IP-XACT component instance.

The expression editor shown in this paper allows the defining of mathematical equations. The editor can support the basic arithmetic formulas, as well as understand different formats for the input of the data. These are the decimal, hexadecimal, octal and binary formats.

Future work includes extension of the current semantic analysis for IP-XACT vendor extensions, which have specified nodes in the XML tree but custom content. This would be implemented by an extensible metamodel for the items of the expressions.

The benefits of the expression editor of Kactus2 are improved usability and reduced number of errors in SoC designs. Using the UUIDs as references allows relia-

ble equation construction with the desired parameters. The use of parameter names in place of the referenced UUIDs in the expression editor help in transforming these equations into more human readable.

Without the expression formula and references validation, the IP-XACT XML files could contain serious errors that can be difficult to find and verify in a large SoC design. As the expression validation is built into the expression editor, the IP-XACT based SoC designs cannot contain parameter reference errors in Kactus2. Thus the presented expression editor improves the SoC design process productivity and quality significantly compared to plain IP-XACT XML editors that are still widely used.

6 References

1. A. Kamppi, J-M. Määttä, L. Matilainen, E. Salminen, T.D. Hämäläinen. Kactus2: Extended IP-XACT metadata based embedded system design environment. 2012 1st International Workshop on Metamodelling and Code Generation for Embedded Systems, MeCoEs. 6p
2. IEEE Standard for IP-XACT, Standard Structure for Packaging, Integrating and Reusing IP within Tool Flows, IEEE Std 1685-2009. The Institute of Electrical and Electronics Engineers, Inc. 18 Feb. 2010. 373 p.
3. E. Salminen, T.D. Hämäläinen, M. Hännikäinen. Applying IP-XACT in product data management. 2011 International Symposium on System on Chip (SoC). 31 Oct 2011 – 2 Nov 2011 pp. 86–91.
4. EDAUtils. IP-XACT solution. [WWW] [Referenced on 14 Aug. 2015]. Available at: <http://www.edautils.com/ip-xact.html>
5. D. Marquès, R. Eixarch, G. Casanellas, B. Martínez. WIRIS OM Tools: a Semantic Formula Editor. Mathematical User-Interfaces Workshop 2006. 10 Aug. 2006. 8p.
6. Language-Enhanced, User-Adaptive, Interactive e-Learning for Mathematics. [WWW] [Referenced on 7 Aug. 2015] Available at: <http://www.leactivemath.org>
7. WebALT: Web Advanced Learning Techniques. [WWW] [Referenced on 7 Aug. 2015] Available at: http://www.webalt.net/index_eng.html
8. T. Lee, T. Chekam. MathCast: The open source equation editor. [WWW] [Referenced on 7 Aug. 2015] Available at: <http://mathcast.sourceforge.net/home.html>
9. W3C. MathML. [WWW] [Referenced on 7 Aug. 2015]. Available at: <http://www.w3.org/Math/>
10. Design Science. MathType 6.9. [WWW] [Referenced on 7 Aug. 2015]. Available at: <http://www.dessci.com/en/products/mathtype/>
11. Design Science. MathFlow. [WWW] [Referenced on 7 Aug. 2015]. Available at: <http://www.dessci.com/en/products/mathflow/default.htm>
12. Oxygen XML editor. [WWW] [Referenced on 19 Aug. 2015] Available at: <http://www.oxygenxml.com/>
13. Open Math Society. [WWW] [Referenced on 7 Aug. 2015]. Available at: <http://www.openmath.org/society/index.html>
14. R. Fateman. A Critique of OpenMath and Thoughts on Encoding Mathematics. University of California, Berkeley, Computer Science Division. 17 Jan. 2001. 10p
15. P. Leach, M. Mealling, R. Salz. A Universally Unique Identifier (UUID) URN Namespace. Network Working Group. July 2005. Available at: <http://www.ietf.org/rfc/rfc4122.txt>

Internal Marketplace as a Mechanism for Promoting Software Reuse

Maria Ripatti¹, Terhi Kilamo², Karri-Tuomas Salli¹ and Tommi Mikkonen²

¹Insta Defsec Ltd, Sarankulmankatu 20, FI-33901 Tampere, Finland
 maria.ripatti@insta.fi, karri-tuomas.salli@insta.fi

²Department of Pervasive Computing, Tampere University of Technology,
 Korkeakoulunkatu 1, FI-33720 Tampere, Finland
 terhi.kilamo@tut.fi, tommi.mikkonen@tut.fi

Abstract. Reuse is one of the classic ways to improve productivity in software development. Indeed, benefiting from software components, patterns, and solutions that have been developed in the company potentially leads to savings in all phases of software intensive work. However, putting such an approach to practice is far from being simple. In particular, when considering software companies that specialize in customer-specific software projects, it is common that similar designs and technology choices are made in parallel without project-crossing knowledge. In such settings, there is a lack of a systematic approach between projects to spread good practices or to eliminate bad ones. In this paper, we propose solving such problems with an information system that acts as a marketplace for promoting software reuse within a project organization, much to the same flavor as app stores are used to promote mobile applications. The paper provides insight to the design of our prototype system, as well as contains preliminary views from users in one organization.

Keywords: Reuse, software projects, inner source, marketplace.

1 Introduction

The business climate today is highly competitive for software companies. This leads to a constant need to look for improvement in development processes in order to maintain the competitive edge. One way to achieve this is reuse – benefiting from software components, patterns, and solutions priorly developed in the company [1]. The benefits reuse promises are improvement in software quality, performance, and reliability [2, 3]. When implemented well, code reuse can shorten development time, which in turn shortens time to market. It can also help to avoid redundant work in projects, such as analysis phase, thus improving project productivity and reducing development effort [3]. Reuse can also make software maintenance easier as reuse unifies coding practices between projects [4]. All in all, the promise of reuse is undisputable.

Project organizations still largely carry out their businesses in accordance to the old subcontracting model, where the customer defines requirements and the

project organization is optimized to perform technical activities needed for requirements elicitation, design, and final implementation. In this context, where each project is treated as a separate entity due to e.g. confidentiality reasons, spreading word regarding successful technology choices gets overly complex. Each technology selection will strictly remain in the project silo instead, and experiences regarding using them are only reused once the developers are allocated to future projects. Therefore, the developers are constantly faced with a challenge of finding suitable solutions to current programming tasks. This includes being constantly on the lookout for components to reuse as such are not readily visible across project barriers. Furthermore, and even more counterproductively, they end up assiduously solving the same problems over again. The challenge posed hence lies in making the components suitable for reuse visible for the developers across projects and over silo borders.

Commonly used approaches to implement reuse include product-line architectures [5] and inner source [6], both of which introduce established reuse processes. Product-line architectures provide a common platform for a typically domain-specific family of products. Each product is then developed by adding the product specific features on top of the shared platform. Inner source, sometimes coined internal open source or corporate source [7], sometimes progressive open source [8], in turn refers to the practise of utilizing suitable open source software development practices and tools within an organization. In general, both approaches build on creating software repositories [9] as a method to give developers access to reusable components. Still, finding suitable components from a large and typically constantly growing repository is challenging [10]. Additionally the lack of decent documentation makes it difficult to evaluate how suitable a component is for reuse. Furthermore, the amount of work required for using it as a part of another piece of software can be hard to estimate. In this paper, we propose an approach to turn this around much like modern online distribution platforms for mobile applications, colloquially app stores, have simplified installing compelling applications that were impossible to find before they were made available through an information system that also syndicates users' views.

To summarize, while the idea of reuse is decades old and different approaches to classifying and representing reusable components in repositories have been around for a while [10, 11] reuse is still not a fluent everyday practice at software companies. This paper addresses the question: how should reuse be implemented within a project organization in order to avoid the problems and challenges repositories bring forth?

The paper presents a pilot study on establishing a company internal component marketplace that engages the ideology of app stores in order to highlight fitting components to developers for reuse across in-house projects. The paper further presents the systematic reuse process alongside and applied to the marketplace. As our prototype, such a marketplace was implemented and taken into use in a mid-sized Finnish software company developing a range of software products mainly to large customers that require confidentiality, high quality and predictable delivery. In short, the paper contributes:

- the concept of using a component marketplace to promote reuse in an organization,
- an industry scale prototype implementation, and
- the reusability process adopted in conjunction with the marketplace.

Here, we address the very first views to deploying using such a system in a project organization. A detailed case study regarding the experiences will be reported later in a separate article, where the resulting increase in reuse will be evaluated.

The rest of the paper is structured as follows. Section 2 presents background on the business environment of the project organization, the challenges of software reuse and discusses the requirements in the adoption of systematic reuse. Section 3 describes the concept of a marketplace as a mechanism for supporting component reuse. Section 4 describes how requirements were gathered from the target organization. Section 5 presents the implemented marketplace solution and discusses how it takes the main challenges into account. Section 6 describes the reuse process adopted at the company. Section 7 discusses the key findings of the study. Finally, Section 8 concludes the paper with future research directions.

2 Background

The benefits of reuse – better quality and reliability, shorter development time, and unified practices – are so overwhelming, that it almost beyond understanding why so many organizations overlook this opportunity. Then again, when considered from the viewpoint of a software company, it is usually self-evident that a lot of effort must be invested in creating practices needed for systematic reuse. The enticing prospect of reuse can and often is hindered by the intimidating challenges in making it a successful ongoing process – if the processes of reuse are not planned and well-established the organization may end up in a situation where a better and faster solution would be to just implement each project on its own. One must be able to identify the reusable components as reuse is not a fit-for-all solution [12]. The components intended for reuse need to be generalized and documented properly [13]. Reuse itself requires finding the suitable components, getting to know them and making possible changes to them [4]. As a concept, reuse has been around from the sixties [14] – and we are still struggling with it.

2.1 Towards Software Reuse

The best results can be obtained with systematic reuse [15]. This also requires that the challenges and problems that reuse entails are identified in the project organization. Even the best laid plans do not alone guarantee a successful adoption of reuse. It needs to be encouraged as a best practice on an organizational level as well as supported by development practices and infrastructure.

Jacobson et al.[13] propose four key processes that are needed in successful reuse (see Figure 1).

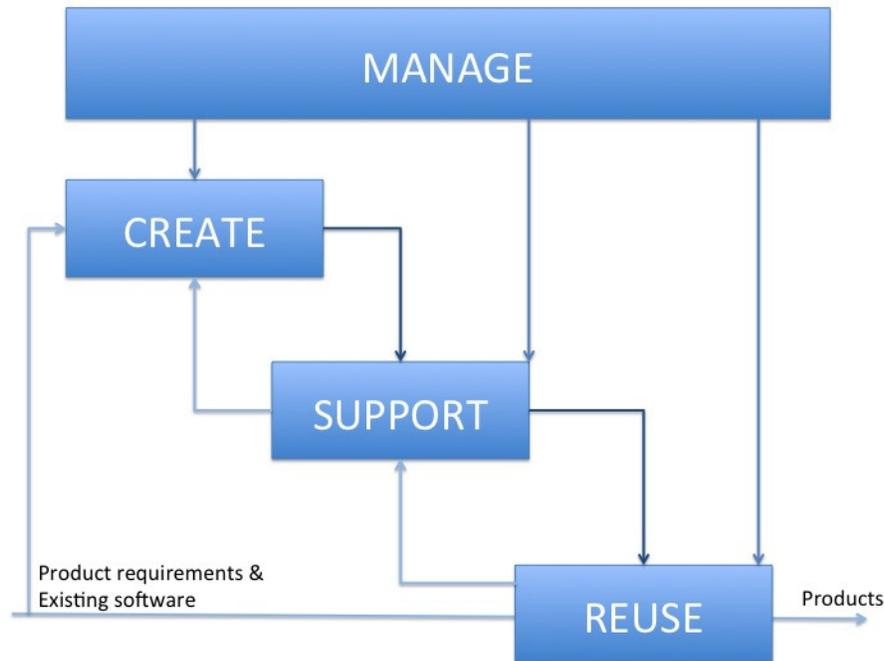


Fig. 1. Four concurrent processes of reuse [13].

- **Create:** The creation process focuses on identifying the needs of and provide for the projects. The development activities included comprise of numerous issues, such as domain planning, development and installation environment, selecting components that are to be reused, and tools that are used for reuse. In many cases, these need a lot of attention in order to introduce generic solutions instead of context-specific ones.
- **Support:** Supporting reuse includes numerous human-related issues. For instance, components to be reused need to be classified and packaged in order to support reuse; distributing them needs common practices; and instructions need to be in place to make all this happen in accordance to plans. The focus is on supporting the needed processes and maintaining the component collection.
- **Manage:** Leadership functions – including planning, funding, resourcing, prioritization, coordination, as well as many other leadership related functions – is often the main obstacle when considering reuse. Balancing between the long-term investment in coordinated reuse and everyday needs of going forward with projects is difficult. Moreover, many of the present agile software engineering approaches, such as Scrum [16] and Kanban [17], are often understood as focusing on satisfying customer requirements. They do not emphasize issues that will help the company in the long run.

- **Reuse:** Actual reuse takes place by selecting the components to be reused, which is a key characteristic for aiming at systematic reuse. Furthermore, systematic reuse entails customizing and combining the reusable components.

Putting all this to practice in a software organization requires management attention, organizational engineering, and hence time that is away from productive work. That in turn is the key ingredient for creating revenues in software companies who specialize in delivering software through customer-specific projects. There, it is common that problems hindering reuse emerge. These include considering only reusing individual components, overly generic designs, lacking scalability, legacy technology, and reuse only for reuse itself [9]. Boehm further emphasizes the danger of "the field of dreams": believing that building a software repository of reusable components suffices to make reuse an everyday practice.

The open source software movement [18] with its development ideology and community-driven approach [19] has risen as a force to be reckoned with when it comes to developing high-quality software products. The concept of a software forge [20] –an repository of projects that can be browsed and that provides the necessary development tools – comes from open source. In its wake, inner source – internal open source – has been utilized by companies to adopt the methodology and ideas of open source software development within a software company. As such, inner source on the development infrastructure level provides a plausible platform for reuse [21]. Inner source does however also pose its set of challenges [21] such as identifying the suitable components and selecting them based on evaluation, poor documentation, as well as integration and architecture issues. These as such are identifiable further as the challenges of reuse itself. However, the factors supporting adoption of inner source [22] – the idea of a seed software product, practices and tools, and organizational and the community-driven approach can also be seen as key factors in supporting reuse.

2.2 Context

The company where the experiment for the proposed approach is carried out, Insta DefSec Ltd¹, is a Finnish software organization that specializes in developing a range of company software projects. The company delivers software mainly to large customers that require confidentiality, high quality, and predictable delivery time. Their potential customer-base includes such governmental organizations as the defence forces and other organizations that often require a certain level of independence of other projects in their execution largely due to their confidential nature. In fact, each project may have different requirements on levels of confidentiality, which may have an effect on the personnel that may participate in them. Despite these limitations and project boundaries systematic, well-organized reuse is beneficial also to the customers. The major programs affect the

¹ <http://www.insta.fi/en/>

company line structure, for example, due to the requirements and project restrictions. Each program usually holds and manages its own resources, including software engineers working on the project as well as almost all the technical project data. Consequently, information regarding successful or unsuccessful technology decisions or design practices is mostly distributed across project borders through word-of-mouth from developer to developer, which though effective in breaking the project silo in separate cases lacks in organization wide governance. For instance, if in one project, the developers perform an analysis regarding the automated mapping from an object-oriented design to a relational database, the results of the analysis would be shareable across all in-house projects. Moreover, experiences from using a certain application-independent component – be it developed internally or a third party system – could be beneficial across the projects, as many of them deal with similar technologies simply due to domain requirements. Without a systematic approach the benefits, such as joint maintenance of common technology, are not gained. The company is constantly looking for ways to improve its ways of working. Thus working towards new methods for systematic reuse and organization wide dissemination of best practices is one area where new directions are tried out.

3 Marketplace as a mechanism for reuse

Despite the promise of reuse code repositories do not seem to provide a sufficient solution to reuse. Instead they seem to bring issues similar to the challenges of reuse itself to the organization. Aiming for systematic reuse is not hence solved by setting up a code repository for the reusable components.

The goal of a company internal marketplace is to combine the idea of a digital online distribution platforms known from mobile apps such as Google Play² and Apple's App Store³ – colloquially coined simply app stores – to these traditional reuse approaches in order to meet the challenges of reuse. This should increase the amount of reuse within the company and help to lower the boundaries between projects. The marketplace aims to solve the challenge of projects acting as knowledge silos. It is used to market components developed with the organization as well as third party components across projects thus taking them into use more straightforward for the projects. The app store features of the marketplace should further make locating of reusable component easier. The marketplace supports making a decision on reuse with clear descriptions on the available components as well as developer comments and instructions of use for them – all features familiar from the app store markets.

Philosophically, the marketplace incorporates the feel of inner source. It promotes transparency, acts as the "seed" product, and attracts contributions across the organization [22]. A component shared through the marketplace can, in addition to the components developed in the projects, also be a third party solution that could be valuable for several in-house projects. For example, in addition to

² <https://play.google.com/store/apps>

³ <https://itunes.apple.com/app/apple-store/>

traditional software components or applications, the reusable component can be a reference implementation, a description of, or a design decision on, an architecture. A software library is shareable through the marketplace as well.

The marketplace can also decrease the effort required to reuse the components by offering the components directly in the marketplace. For reuse, the most important factor is that the marketplace offers an easy way to locate, add and describe components. Hence the components can physically be located outside the marketplace in a separate repository or online as long as the location is explicitly shareable through the marketplace. As summary, in order to meet the needs of systematic, successful reuse the marketplace should:

- act as an internal information channel for the organization
- make adopting reusable components more fluent
- guide development of reusable components
- guide technology choices made in projects
- motivate development toward reusable components
- help keeping track of available reusable components.

The marketplace aims to motivate further development of reusable component by enabling advertizing them through the marketplace [23]. Through the marketplace the projects can also give component recommendations. Finally, open source components are also shareable at the marketplace which allows the organization to better keep track of them. This way they can also ensure that the licence terms are known by the developers.

4 Requirements Gathering

Requirements elicitation was done in two stages to ensure that the internal marketplace would address the true needs of the developers. In the first stage, a rough list of requirements were gathered in meetings among a small group of people taking part in the marketplace research. Based on the requirements recognized there and from the conversations in the meetings, an inquiry about software reuse was created. The purpose of the inquiry was to identify more specific requirements.

In the second stage of the requirements gathering, the drafted inquiry was sent to the project managers, architects and developers of the company. The results confirmed that the requirements identified by the marketplace research team were accurate. The results also defined the priority of the requirements.

According to the results, the marketplace should enable sharing the following information regarding reusable components:

- basic information such as the name and version of components and other useful details
- the technical and functional descriptions,
- the locations and contact persons of the components, and
- prices and licences, if a 3rd party component was included.

Respondents also considered important that jar packages can be uploaded directly through the marketplace. If a direct marketplace access is not possible, it should offer direct links to repository locations of the components and any other important files such as license conditions, version history or bug databases. Respondents also pointed out that the marketplace can not be implemented as a cloud service because of the information security reasons. The gathered requirements were used in evaluating the possible marketplace implementations.

5 Marketplace implementation

The goal of deploying the marketplace within the organization is for it to support the systematic reuse process. Hence, the implementation should simplify the reuse process as a whole, including all the activities listed above, but with a particular focus on managing the reuse. Additional focus is put on promoting people to reuse both software assets as well as research work invested in selecting best possible third party libraries as these were the missing link between ad-hoc reuse and systematic reuse processes. As a solution, an information system was introduced. The system would gather all the necessary data into one, similarly to the digital online distribution platforms — app stores — that have become common in the mobile domain and in online stores.

Since several app and web stores exist already, we next performed an evaluation regarding the already existing implementations. As a result of the evaluation process, OpenCart⁴ was chosen to the pilot use of the internal marketplace. OpenCart is free open source e-commerce platform for online merchants. OpenCart provides a professional and reliable foundation from which to build a successful online store. This foundation appeals to a wide variety of users; ranging from seasoned web developers looking for a user-friendly interface to use, to shop owners just launching their business online for the first time. OpenCart has an extensive amount of features that gives you a strong hold over the customization of your store.

OpenCart offers an e-commerce platform and admin portal. The e-commerce contains for example a front page, category pages, product pages, a shopping cart, a product comparison and the search of products. The front page is used for advertising products. Products are introduced at the category, product and comparison pages and they can be bought via shopping cart. The admin portal provides user and product management functionalities, reports about the e-commerce usage and the customization of the store.

Transforming OpenCart to a software component marketplace required some changes to both the e-commerce and the admin portal. OpenCart is primarily intended to selling tangible products and by default it does not support sharing electrical content. Due to this the product descriptions and the language used required customization. Furthermore, the e-commerce and admin portal contained many features and pages that were unnecessary for the marketplace. For

⁴ <http://www.opencart.com>

example, the shopping cart, payment features and unused reports were removed. These changes were made directly to the source code.

As shown in Figure 2, the OpenCart design is based on the MVC design pattern and the language specific information is separated from the other content. The OpenCart directory structure contains separate folders for e-commerce and admin model, view, controller and language files. E-commerce files can be found in the catalog folder and admin files in the admin folder. The OpenCart installation contains also other folders such as system and image folder that contains classes that are used by both the e-commerce and admin portals. The structure of the OpenCart and the developing process of modules are extensively explained in the OpenCart documentation, which makes creating new modules and customization existing modules easy.

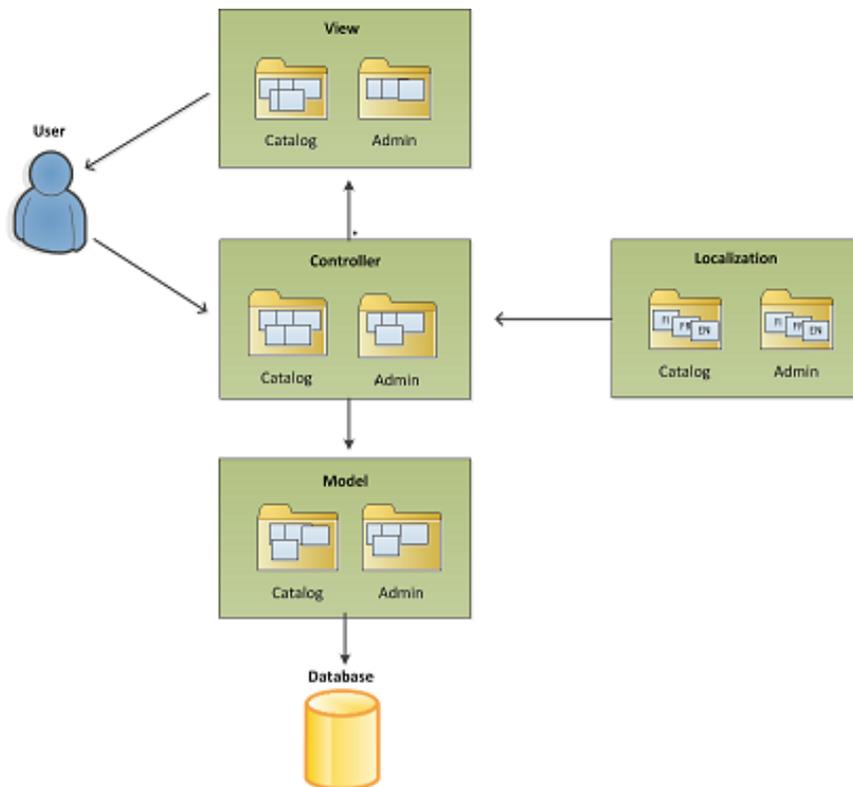


Fig. 2. OpenCart architecture.

The marketplace is used for promoting reusable components but actual components are located in the reuse repository because of the absence of electrical sharing. The modified marketplace (Figure 3) contains the front page that is

used for advertising reusable software components. The content of this page can be customized using the admin portal. In the front page given in the figure, there is a welcoming message, links to two components, as well as references to some topic areas according to which components can be grouped.

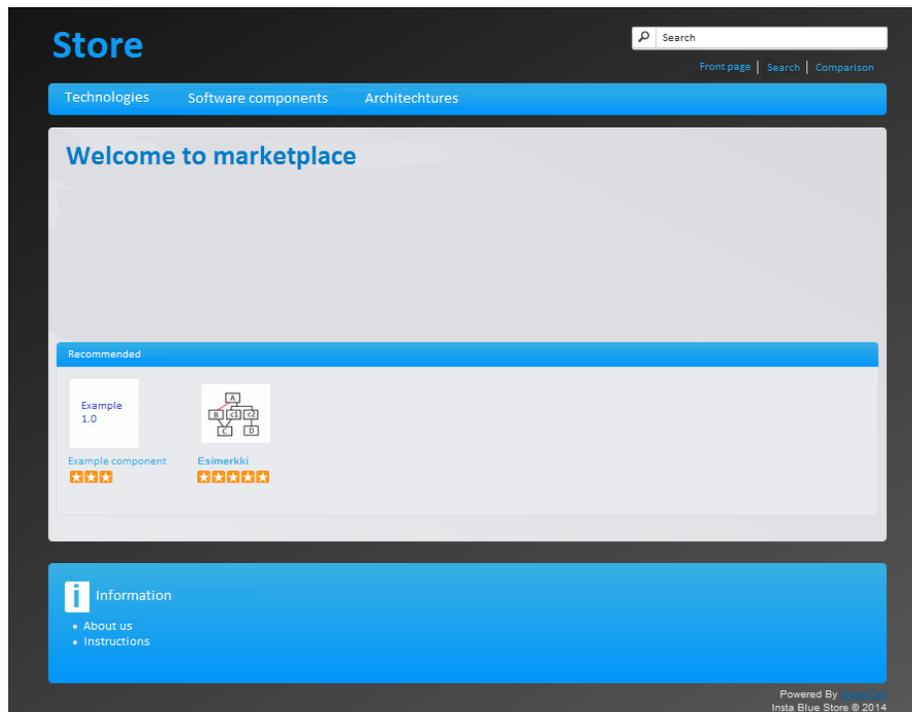


Fig. 3. Minimal user interface of the marketplace.

Within the system, software components are divided into different categories. The marketplace offers also an option to search for components. OpenCart's modified product pages are used for introducing components and they contain the essential information about the component deployment and usage. For instance, the description of a component might contain the following information:

- the version of the component,
- the purpose of the component,
- a contact person,
- a technical and functional description,
- a maintenance information,
- requirements and constraints,
- the location of the component, and
- licence term.

In addition, the OpenCart system supports user reviews, which in the marketplace are used for evaluating components.

6 Reuse process and marketplace

A company wide inner source approach was adopted for collecting and selecting the reusable components. The marketplace acts as one component in this process. The goal of the process is the systematic management of the reuse processes and the ability to control and guide the development of reusable components. This in turn aims to guide the selection of the technologies used. The process supports both the viewpoint of design-by-reuse as well as design-for-reuse [24] with the marketplace as a tool for advocating both views. The goal of the process is to identify and evaluate reusable components [25].

Figure 4 illustrates the complete reuse process within the organization with relation to the marketplace. The starting point of the component reuse is the company's internal developer community as they develop the product components. It is the job of the community, including the projects and product management, to initially recognize the reusable components and offer them to the Product Decision Board (PDB) for the evaluation process. As the company's business domain covers governmental organizations, some of the software components are tightly coupled with customer project-specifics and as such project-sensitive with a requirement of confidentiality. Naturally such components cannot be used in favor of other projects without major modifications.

The role of the PDB is to evaluate the components offered by the developer community and prepare a decision proposal in the form of Product Decision Card (PDC). PDB evaluates and selects the suitable components for the reuse and instructs the developer community to make the necessary changes to the component for reuse. After the decision on reusability the selected project will be responsible in modifying and documenting the component for reuse.

When the component modification for reuse is completed, it is added to the marketplace together with a description of the component. The description includes a list of features, an architecture description and user guide instructions. The administrator of the marketplace is responsible for making the component accessible for the company's developer community. That means adding the component and its artifacts to the marketplace. The customer projects can utilize the available components distributed through the marketplace. The developers can share reuse experiences with the chat tools offered there. The administrator ensures the reliability and correctness of the components. From the maintenance point of view the selected project takes care of the source code and the configuration of a component.

In the following, we evaluate PDB's reusability decision from three different perspectives – business, technology, and customers and stakeholders.

Business. From a business perspective the decision to productize is based on the evaluation of business model, product strategy, distribution strategy and

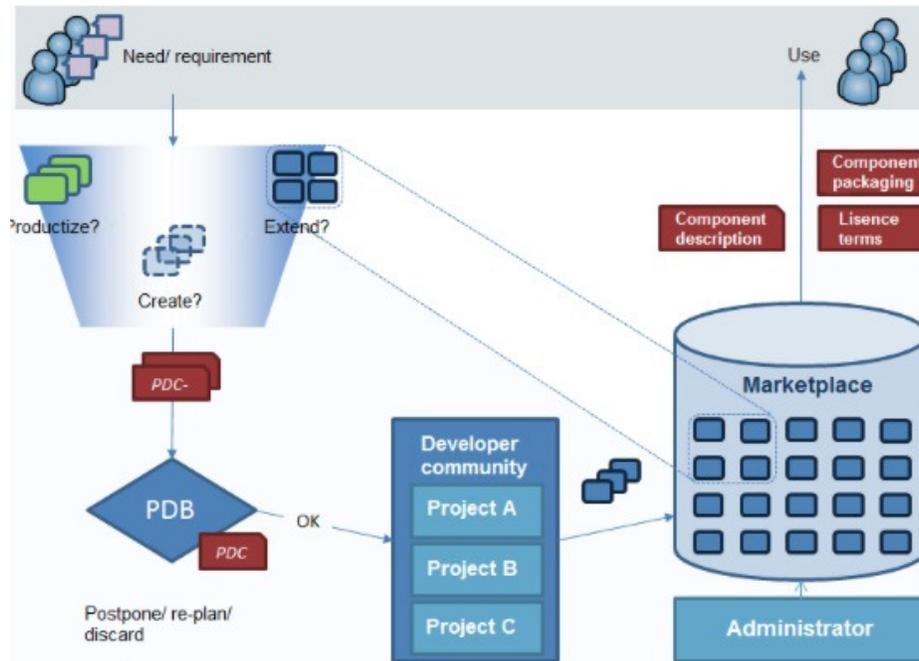


Fig. 4. Reuse process in the target organization

revenue model. In addition, PDB takes into account the productization costs required for changes and possible 3rd party license costs.

Technology. From the technological point of view, the evaluation includes the technical feasibility and quality assessment of the code and the software architecture. The decision to productize a component also needs assessing the maturity and the component life cycle as packaging a component too early can cause the need for multiple updates which in turn will cause extra work and costs. On the other hand later updates can further lead to software incompatibility issues.

Customers and stakeholders. Overall, the customer projects and stakeholders' point of view the component must be utilized in a number of projects and component deployment process must be faster and cheaper than the creation of a new solution from the scratch.

7 Discussion

Next, we will revisit the four key processes proposed by Jacobson et al. and then we will discuss the marketplace in relation to them.

Create: The role of the process is to provide for the reusers. The marketplace aims to act as a clearer, easier and more accessible platform to share and

find reusable components. The development of the marketplace has been largely a part of the creation process. The needs of the developers were taken into account prior and during development. The requirements of the marketplace were prioritized based on developer needs.

Support: The role of the support process is to maintain the reusable components, the repository and the reuse processes. The marketplace plays a key role here. The role of the component descriptions is seen pivotal by the developers. Good, clear descriptions support reuse and make selection of components easier while the opposite can be quite harmful for the support process.

Manage: With the marketplace, the reuse process depicted in Section 5 provides the management for the other processes. The developer's see the role of the PDB and making correct reusability decisions as key.

Reuse: To what extent the marketplace increases reuse remains to be seen at this point. Based on the feedback, the developers see the marketplace as a welcome addition. The response of the developers will be discussed next.

Since the marketplace has just been deployed at the target organization, as a part of the prototype deployment an interview of the key stakeholders was conducted. Next, we discuss the feasibility of the marketplace through the estimation interview. In it three representatives of the intended target community – a software developer, a software architect and a project manager – was interviewed in an open interview session. The results indicated that there is a call for the marketplace at the organization. All interviewees seem to have identical opinions on the marketplace and how it should be developed further.

The content of the marketplace was seen as a key element in attracting the developer community. The content needs to be attractive to the developers and it needs to be maintained continuously. The interviews also highlighted that the use of the marketplace needs to be smooth in order to enable updating the component descriptions as well as to support creation of new components. In order to get content, all developers should be able to add components to the marketplace.

The active role of the developers was also valued. As many developers currently look for and reuse same components available online, a recommendation feature for them is seen beneficial. The feature would avoid a similar approach to be applied to the marketplace as it would relieve projects from doing the searching themselves in every case. Furthermore, an egalitarian approach to adding content to the marketplace was emphasized.

The reusable components themselves were seen as a valuable asset. The developers should however be able to estimate based on the description alone if utilizing the component is worth it or not. Hence the description should include the most restrictive knowledge that can affect the developers decision. Such are, for example, the licence, the price of the component and the process needed to reuse. Especially the ability the evaluate 3rd party components was seen as an asset, as it could save time and money. The fact that the components can include best practises and architectural decisions was valued.

8 Conclusions

At this point, the marketplace has been successfully deployed in the target organization. The developers comments on the marketplace and the organizational needs for it support the claim that it can make reuse more systematic and help to put in place and maintain the four key reuse processes. As future research we also wish that we have enough industrial data to validate these claims with experience data.

Presently, the marketplace is meant for company use only but as a future direction the possibility to share the marketplace with organization partners is considered. This lends way to future research on reuse over organization boundaries. The initial results are encouraging and the marketplace has shown its potential in enabling and supporting reuse over project silos to entire ecosystems that comprise of several companies. The next research steps are to collect data on the amount of reuse as well as evaluate what kind of reuse gets done; so far, the initial experiences look promising.

References

1. C. W. Krueger, "Software reuse," *ACM Computing Surveys (CSUR)*, vol. 24, no. 2, pp. 131–183, 1992.
2. D. Bauer, "A reusable parts center [technical forum]," *IBM Systems Journal*, vol. 32, no. 4, pp. 620–624, 1993.
3. W. C. Lim, "Effects of reuse on quality, productivity, and economics," *Software, IEEE*, vol. 11, no. 5, pp. 23–30, 1994.
4. J. Sametinger, *Software Engineering with Reusable Components*. Springer, 1997.
5. K. Pohl, G. Böckle, and F. V. D. Linden, "Software product line engineering," *Springer*, vol. 10, pp. 3–540, 2005.
6. J. Wesselius, "The bazaar inside the cathedral: Business models for internal markets," *Software, IEEE*, vol. 25, no. 3, pp. 60–66, 2008.
7. R. Goldman and R. P. Gabriel, *Innovation Happens Elsewhere: Open source as business strategy*. Morgan Kaufmann, 2005.
8. J. Dinkelacker, P. K. Garg, R. Miller, and D. Nelson, "Progressive open source," in *Proceedings of the 24th International Conference on Software Engineering*. ACM, 2002, pp. 177–184.
9. B. Boehm, "Managing software productivity and reuse," *Computer*, vol. 32, no. 9, pp. 111–113, 1999.
10. R. Prieto-Diaz and P. Freeman, "Classifying software for reusability," *Software, IEEE*, vol. 4, no. 1, pp. 6–16, Jan 1987.
11. W. B. Frakes and T. P. Pole, "An empirical study of representation methods for reusable software components," *Software Engineering, IEEE Transactions on*, vol. 20, no. 8, pp. 617–630, 1994.
12. D. Garlan, R. Allen, and J. Ockerbloom, "Architectural mismatch: Why reuse is so hard," *Software, IEEE*, vol. 12, no. 6, pp. 17–26, 1995.
13. I. Jacobson, M. Griss, and P. Jonsson, *Software Reuse: Architecture, process and organization for business success*. Addison-Wesley, 1997.
14. M. D. McIlroy, "Mass produced software components," in *Software Engineering: Report of a conference sponsored by the NATO Science Committee*. NATO, 1968, pp. 79–87.

15. W. B. Frakes and S. Isoda, "Success factors of systematic reuse," *Software, IEEE*, vol. 11, no. 5, pp. 14–19, 1994.
16. K. Schwaber, "Scrum development process," in *Business Object Design and Implementation*. Springer, 1997, pp. 117–134.
17. D. J. Anderson, *Agile Management for Software Engineering: Applying the theory of constraints for business results*. Prentice Hall Professional, 2003.
18. "Open source initiative," <http://opensource.org/>, last visited: September 2014.
19. E. Raymond, "The cathedral and the bazaar," *Knowledge, Technology & Policy*, vol. 12, no. 3, pp. 23–49, 1999.
20. D. Riehle, J. Ellenberger, T. Menahem, B. Mikhailovski, Y. Natchetoi, B. Naveh, and T. Odenwald, "Open collaboration within corporations using software forges," *Software, IEEE*, vol. 26, no. 2, pp. 52–58, 2009.
21. K.-J. Stol, M. A. Babar, P. Avgeriou, and B. Fitzgerald, "A comparative study of challenges in integrating open source software and inner source software," *Information and Software Technology*, vol. 53, no. 12, pp. 1319–1336, 2011.
22. K.-J. Stol, P. Avgeriou, M. A. Babar, Y. Lucas, and B. Fitzgerald, "Key factors for adopting inner source," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 23, no. 2, p. 18, 2014.
23. D. Ansoorge, K. Bergner, B. Deifel, N. Hawlitzky, C. Maier, B. Paech, A. Rausch, M. Sihling, V. Thurner, and S. Vogel, "Managing componentware development – software reuse and the v-modell process," in *Advanced Information Systems Engineering*, ser. Lecture Notes in Computer Science, M. Jarke and A. Oberweis, Eds. Springer Berlin Heidelberg, 1999, vol. 1626, pp. 134–148.
24. S. Castano and V. D. Antonellis, "Reusing process specifications," in *Proceedings of the IFIP WG8. 1 Working Conference on Information System Development Process*. North-Holland Publishing Co., 1993, pp. 267–283.
25. A. B. Al-Badareen, M. H. Selamat, M. A. Jabar, J. Din, and S. Turaev, "Reusable software component life cycle," *International Journal of Computers*, vol. 5, no. 2, pp. 191–199, 2011.

Lean Startup Meets Software Product Lines: Survival of the Fittest or Letting Products Bloom?

Henri Terho¹, Sampo Suonsyrjä¹, Ari Jaaksi², Tommi Mikkonen¹,
Rick Kazman³, and Hong-Mei Chen³

¹ Tampere university of technology, Korkeakoulunkatu 1, FI-33720 Tampere, Finland
henri.terho@tut.fi, sampo.suonsyrja@tut.fi, tommi.mikkonen@tut.fi

² Idean Enterprises Ltd., Hämeenkatu 18, FI-33200 Tampere, Finland,
ari.jaaksi@linux.com

³ University of Hawaii, Honolulu, HI, USA
kazman@hawaii.edu, hmchen@hawaii.edu

Abstract. Typical management strategies proven to work in already established businesses do not work as expected in startups. Startups do not yet have a business model and product that they could focus on, but are still looking for a working business model. Lean Startup is a method for startup management that focuses on quick iteration and on fast learning to find an iterable business model. As a method, Lean Startup is still quite novel. It does not have much scientific literature written about it, but it is used by startups. The two case study companies were both positive about Lean Startup and felt that the method had given them a helpful approach.

Keywords: lean startup, iteration, case study

1 Introduction

New product development and business model evolution are critical competencies for any company. Their value is intensified, however, in the case of startups, where the entire business model can be unclear or at least remain uncertain and untested. To find a fast-track to profitability, a startup needs to streamline and speed up the two vital processes – finding new markets and developing novel products. This requires highly optimized techniques and methods for the management of products [4].

An emerging choice for such a management method is Lean Startup [19, 5]. As the name suggests, the method has emerged from the niche of small companies that are starting up their businesses. Such companies form an interesting field of study, as they are seeking to validate their ideas and products as quickly as possible, but, at the same time, efficient execution of such processes is vital. Lean Startup defines a process for validation, where companies build, measure and learn by creating Minimum Viable Products (MVP) [19]. In the process of

getting to a finalized product, companies might go through a number of different MVPs. Depending on the company, these MVPs might share characteristics, build on common tools and technologies, or aim at the same market.

Efficient, rapid development of new products has long been a desire for most software organizations, and there have been other attempts to manage families of related products, such as Software Product Lines (SPL). Similarly to Lean Startup, SPLs promise increased productivity and reduced time-to-market [24], in the context of SPLs achieved by reusing common core assets for building families of related software products. Although the initial development of reusable software and a common product-line architecture requires additional up-front effort, this effort will later be more than compensated over time. For example, maintenance and evolution of the different products in the SPL can be centrally planned and coherently staged.

As Lean Startup and Software Product Lines have some similar goals, we have decided to analyze their commonalities and differences. After we first consider the background and relationships of these two concepts, we provide an empirical study, where we investigate how two Finnish software startups have implemented Lean Startup in their software development. In particular, we analyze how multiple MVPs developed by these companies compare to an SPL, and whether we could use the established body of knowledge regarding SPLs to analyze Lean Startup's MVP development practices to avoid some of its known risks.

The main research questions we have formulated for this paper are listed as follows:

- RQ1: What parallels can be drawn between SPLs and Lean Startup?
- RQ2: How did the case companies use the Lean Startup method in their software development?
- RQ3: What kind of similarities are there between the outcomes experienced by the case companies and can we relate these to the similarities (and differences) between SPLs and Lean Startup?

The rest of this paper is structured as follows. In Section 2 we go through the theory of SPLs and Lean Startup. We also place particular focus on the MVP aspect of the Lean Startup approach. In Section 3 we go through our research approach and case study companies. In Section 4 we go through the case study results. In Section 6 we provide an overview of lessons we have learned in the process. In section 5 we take a look at the validity and reliability of the study. Finally, in Section 7 we summarize the paper by drawing final conclusions.

2 Background

2.1 Software Product Lines

A Software Product Line (SPL) is a systematic way to share a common set of core assets used in a series of related products, targeted for a certain market or

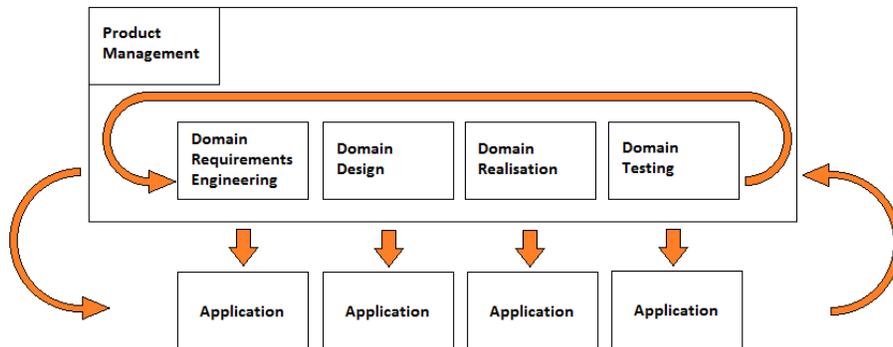


Fig. 1. Software Product Lines

for a specific mission [6]. In technical terms, the creation of an SPL culminates in the creation of an infrastructure that allows rapid, organized production of similar software systems [9]. Such an approach has proven to be efficient for both architecture and component-level reuse [24]. Moreover, an SPL can be seen to include a broad field of different subjects ranging from business to architecture and from processes to organizations [12].

A major goal of an SPLs is strategic reuse: managers, analysts, architects, developers, and testers can avoid performing the same activities over and over again by reusing existing (parameterized, tailorable) assets. SPLs have been regarded as a methodology for developing software products and software-intensive systems in short time and with higher quality [21]. This also allows companies to produce products that closely related to each other with lower cost and higher quality (for example, reduced rates of defects) [11].

There are two principal ways to develop an SPL, and they balance their risks and the aforementioned benefits somewhat differently:

- A proactive reuse-based approach may be adopted in cases where the risk of developing possibly useless assets is accepted. As the shared assets of an SPL are developed before they are used in products, an up-front investment is obviously required. This requires added work in software asset management and introduces the risk of increased time to market for the first products [10]. This approach is typically taken in more mature domains, where the company already has experience in creating similar products and has the expectation of creating many such products in the future.
- A reactive approach—where assets are created and made general on an as-needed basis, and in an evolutionary fashion—can significantly reduce this up-front cost, but at the same time it requires closer coordination within the SPL project [24]. Moreover, this can also lead to a shorter time to market but at the expense of greater levels of re-work and waste.

Typically SPL product development is divided into two distinct software development processes: the domain and application development processes. The domain process focuses on creating shared, reusable software artifacts for the different applications created in the applications process. The applications build on the domain assets and add functionality, as needed, that differentiates the applications one from another. This is outlined in Figure 1. The nature of the software artifacts in the different application instances is constantly evaluated, and if recurring artifacts are discovered in those applications they are integrated into domain engineering. The split between domain and application development requires domain expertise from the software developers and architects who evaluate what should be part of the domain. [11]

Despite all the advantages of using an SPL, there are some risks and issues related to them as well. For example, [14] describes several challenges, especially for smaller companies. Smaller companies usually have more limited resources for creating holistic product lines and have difficulties in affording full-scale platform development and maintenance. Moreover, their environments are often highly volatile, which increases the risk that the costs of creating the product line exceeds the benefits. In addition, the difficulty of using objective methods without historical data leaves these companies relying on their personal opinions as to whether a given reuse strategy is actually cost-effective.

2.2 Lean Startup

Lean Startup is a model created by Eric Ries and Steve Blank for the development of startups [19] [5], building on the idea that the goal of a startup is to transform new ideas into products. An iterative cycle of building, measuring, and learning is proposed. First, an idea is turned into a piece of software. When customers interact with the produced software, they generate feedback, typically both qualitative and quantitative. Based on this feedback, the startup learns more about their business space, the performance of their designs, and hence the acceptance of their products. In the practical implementation of these cycles, each cycle is typically linked with its own MVP, which is used to test the hypothesis of the current cycle.

The iterative cycle is run as follows (Figure 2). A company initially enters the loop in the ideas state. This means that one has an assumption: a hypothesis of a business plan that is being refined into the first product. This first assumption is called a leap-of-faith assumption, which is based on data outside of the build-measure-learn cycle. The original leap-of-faith assumption is one of the most critical points, but the Lean Startup method provides no way to test it beforehand. The problem should be assessed with customer interviews or other methods for determining initial feasibility. This is then fed into the build-measure-learn as an initial assumption and iterated upon to reach a valid product hypothesis.

The build phase is the transition from the ideas state to the code state. During this phase a version of the product is built, based on the most recent ideas. This product is an MVP, meaning a product that only contains the minimum

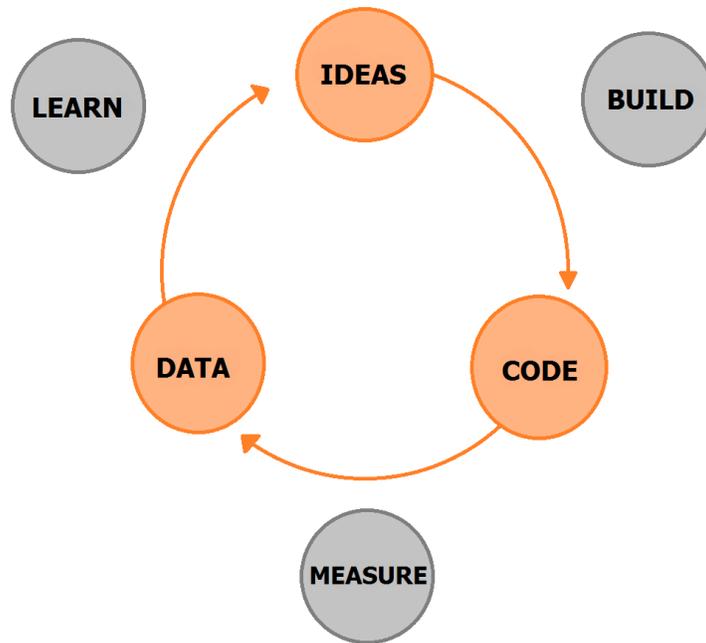


Fig. 2. Build-measure-learn loop

amount of features to make it viable with the minimum amount of work. The goal is to maximize the learning gained from multiple iterations through the build-measure-learn loop.

In the code state, startups have a ready product version of their program. The code is not totally ready—for example, it may not be robust and may not cover a broad range of exceptional conditions—but it fulfills the goals to test the current hypothesis. This is a minimum viable product version. The software should also include analytics code to enable the collection of data about customer behavior, or there may even be two versions of the MVP, to enable A/B testing of the product.

The measure phase is where the product is deployed to the customers. The product is used by real customers and data about their behaviour using the product is collected by the analytics code in the program.

The data state is where the startup has collected data from the usage of the MVP. The purpose of this data is to decide whether product development efforts have led to progress. The data collected from the product should be sufficiently concrete and sufficiently connected to the desired customer outcomes so that it can be immediately acted upon.

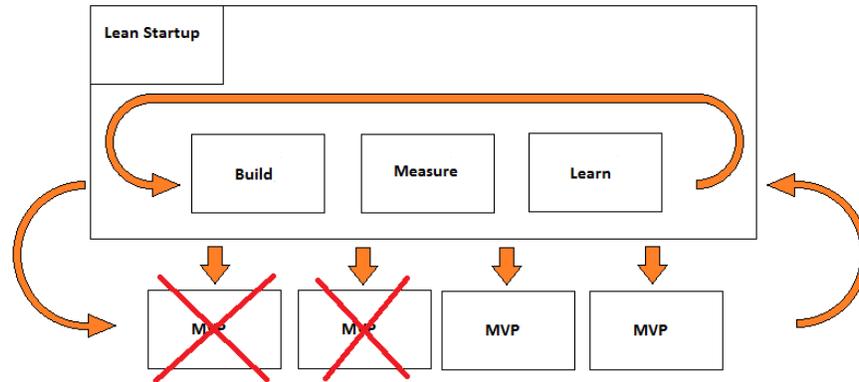


Fig. 3. Lean Startup process

After this comes the learning phase. In this phase the startup compares the data collected to their original product hypothesis and assesses whether learning milestones from the hypotheses have been fulfilled. The data can be used to see if the customer behavior matched the startup's expectations and if the changes made in this iteration have improved the software. For example, one might learn whether the new landing page has increased the amount of subscriptions. If the hypothesis was successful, a new hypothesis should be formulated to further improve the software, ending up back in the ideas phase.

In Lean Startup software development, multiple MVPs are created to help find the optimal business model for the company and to find information about the business space the company operates in. Typically multiple MVP versions are done during the lifecycle of the company, as multiple turns of the build-measure-learn loop are iterated. This iterative process is outlined in Figure 3. These MVPs and its subset minimum viable features (MVF) are then used to iterate the different aspects of the startup. During this iterative process multiple MVPs and MVFs are created and evaluated, and further MVPs and MVFs can be built upon the older versions with increasing efficiency.

An MVP might not be used just for finding the optimal business model for the company. The minimum viable product is defined as a version of the product that enables a full turn of the build-measure-learn loop with minimum amount of effort and the least amount of development time. It should contain the features that realize the software solution's unique value proposition and little else, except for logging and metrics integration. The idea is to cut out all non-essential features and leave just the core features of your application. In the same way a minimum viable feature (MVF) is a single software feature with just its basic aspects implemented. [13]

The minimum viable product is not always the simplest possible product if the problem the startup is trying to solve is not simple. Rather, the MVP should solve the core problems or jobs that the customer wants to get done.

Regarding actual software development, Lean Startup does not define any particular model. Instead, various approaches, including Scrum [22, 23], Lean software development [17, 18], and Extreme Programming (XP) [3, 2], are seen to be applicable. The main characteristics desired of the development model are the ability to provide cost-efficient designs that can be experimented with and their rapid development. Moreover, one can consider development approaches (e.g. [15, 16, 8, 7]) that build on experimentation as derivatives of the same mindset.

2.3 Synthesis

Software product lines have been developed to allow a company to produce multiple similar variants of its core product. This product line approach enables the company to push products out to the market faster and more efficiently than before, once the common assets and common architecture have been defined, by reusing shared domain assets for each product version.

Similarly, Lean Startup produces MVPs as fast and efficiently as possible to the market to gain information about the business space of the company and to iterate the company's core product towards a better one. In a sense, these iterations with MVPs eventually produce similar kind of a product line with specific variations to each product that answer the current hypotheses of the company.

The first few MVPs that a startup produces with Lean Startup might differ considerably from each other. This is due to the fact that the startups have limited domain knowledge and are working mainly on the basis of their leap-of-faith assumptions. After validating these first assumptions through MVP iterations, their MVPs should be more and more focused and resemble each other increasingly later on. This is because the company should first learn the things that are the most vital to their business, such as their business model, targeted markets, and the problem they are trying to solve for their customers. This avoids larger modifications in the later stages of product life.

After validating these hypotheses about their initial business space with MVPs, new MVPs should be developed to assist in learning ever smaller and more focused things. This again allows the increase in the reuse of software components from the previous MVPs. For example, if the software company focuses their product on a specific domain, such as web applications, the product line to produce multiple web MVPs could be constructed based on the existing knowledge of what is common in web applications and on top of this create different MVPs. The production (line) of such focused and similar MVPs could be seen as an "Iterative Innovation Engine" - a platform for a core product on which the startup builds its experiments to validate new hypotheses.

Thus, both Software product lines and the Lean Startup method promise increased productivity and faster time to market. Even though they have a different originating context, one from the startup domain and one from the corporate domain, they both are processes to produce multiple similar products. Whereas Lean Startup talks about MVPs, or MVFs, these end up resembling

similar software artifacts as the different applications in reactive software product lines. One key difference, however, is that the SPL approach plans to produce similar products in parallel, whereas the MVP approach plans to produce them sequentially. Thus the up-front costs and time-to-market of MVPs is lower, but the total lifecycle costs may be higher.

3 Case Study

3.1 Research Approach

The goal of this study is to investigate how our case startup companies used Lean Startup for their software development in practice. In particular, we address the effect of using MVPs in the creation of new businesses and business models, and the consequences of executing such a process.

This study was executed by performing semi-structured interviews and based on the knowledge of the authors from working in the two case companies. In general, a case study approach is particularly useful in situations where the phenomenon and context are difficult to separate [25, 20]. These semi-structured interviews also allowed for more in-depth discussions, which in turn made it possible to gather more information from the processes of the companies than pre-selected questions would have enabled.

3.2 Case Study Companies

Movendos Movendos is a new software startup focusing on creating effective tools for health and wellness coaching. The main product of the company is the Movendos health coaching platform. The first author of the paper has worked in this company. The goal of the company at the time of writing is to create an online cloud tool to help coaches to keep better track of their trainees and clients and thus enable cost savings for the health service provider.

The company has used the Lean Startup product development and multiple MVPs to iterate its core product to its current state through multiple iterations on it. The original business idea has evolved through the different versions from personal training data tracking, remote heart rate tracking to its current form as a remote training tracking tool.

Taplia Taplia is a software company creating simple and lightweight web applications that can be used to automate the logging of work hours. The products are targeted for field service engineers and other professionals working on time-based assignments. The first and the second author work in this company. Taplia aims at making work time tracking more efficient by removing paper shuffling and manual entry of time slips into a payment system. The company offers product customization on a per-client basis for an additional fee.

The company is still in its early stages of trying to find a valid business plan on which to iterate and produce a profitable business. A few business cases have

been developed with Lean Startup, but it still remains to be seen if the current business model is the final one.

3.3 Software Development in Case Companies

First, we investigated what kind of processes the case companies used for their software development. It turned out that both case study companies followed the iterative development model laid out by the Lean Startup. Although both companies dropped some parts of the Lean Startup methodology, both used the build-measure-learn based iterative method with MVPs. Taplia has produced four MVP products, Movendos implemented three during the time investigated in the study. The MVPs were used to test new product concepts or refine existing ones to guide the company decision making.

Secondly, we compared the different MVP versions developed by both companies and identified some common parts and patterns. The first MVP for Movendos was a test of a product concept for mobile heart rate and gym log tracking on a mobile platform. The second MVP was a more generalized version intended for tracking all exercises and food diary data with a mobile platform. The third product was expanded to include more focus on the coach using the system and transferred to the web platform. In total Movendos created three MVPs in the web application domain.

For Taplia, the first MVP done was a work-hour-logging web application for the mobile environment. The second was an hour-logging and planning application for gym employee hour tracking. The third was a work-logging and customer work order tracking tool for constructions companies. The fourth was an expanded gym employee hour planning and tracking software in the cloud. In total Taplia created four MVPs in the web application domain.

In both companies, the MVPs were based on the same core web technologies, which were also used in different MVPs. Although Lean Startup promotes the possibility to make radical pivots if an MVP is unsuccessful, both of the companies were highly web software oriented from the beginning and thus neither of the companies changed their initial decision on producing web software. Therefore, the later MVPs in the companies were generally developed faster as there was considerable opportunity to reuse gained knowledge of these technologies. For example, parts of the UI components were reused between the different MVP versions. However, in some cases technology switches were made and the core technologies were updated or changed. All in all, the case companies felt that the Lean Startup allowed the companies to develop their MVPs efficiently and learn rapidly from customer feedback.

Finally, we looked into the situations where the case companies ended up by following Lean Startup. Despite the advantages of reuse options, both companies also found out difficulties in their implementations of Lean Startup. In both of them, the termination of completed MVPs turned out to be difficult, and consequently the companies ended up with multiple products, each with a small user base each in use at the same time. By the book, however, MVPs are

primarily intended for learning purposes, and therefore only the fittest of the MVPs should survive (Figure 4).

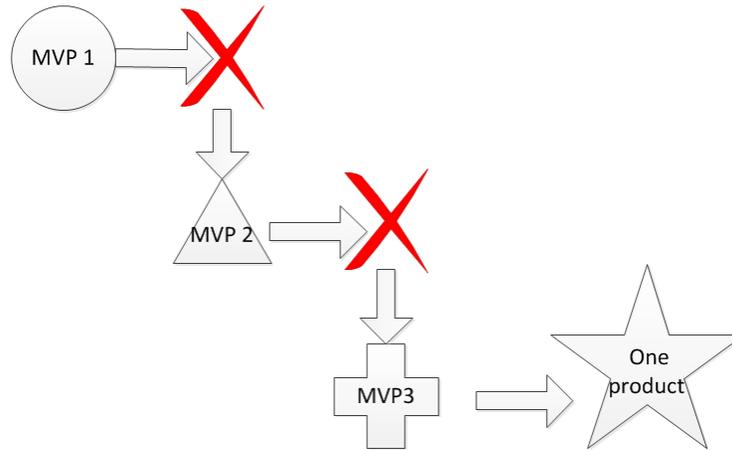


Fig. 4. Survival of the fittest MVPs

In our view, the closing of previous MVPs and freeing their resources could allow companies to make more extensive pivots, and therefore end up with a stronger final product. However, the situations our case companies ended up with resembled an involuntary (and sub-optimal) product line depicted in Figure 5.

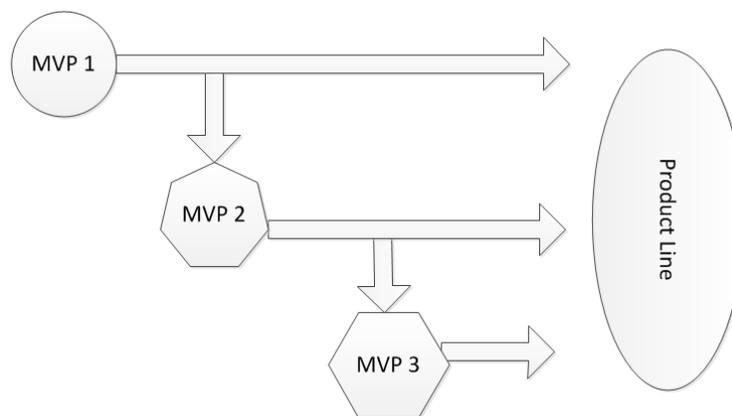


Fig. 5. Involuntary product line from MVPs

Both companies faced the same major challenges in the latter phases of the MVP development. When an MVP was developed and introduced to the customer, it became very difficult to stop developing it further, even if the customer feedback was not very good. Taplia customers even wanted to continue using older MVPs that were created for early prototyping and testing. Thus, shutting down the MVPs was not possible without risking the unhappiness and potential loss of these customers. This led to a situation where Taplia had to support multiple versions of practically the same MVPs at the same time but in practice mostly separately from each other. Movendos also had similar situations, where multiple versions of an MVP were in development at the same time. There was an overlap between finishing one MVP and starting to design another one. The first MVP was also in use with a customer at the same time as the development of the second one was progressing.

In these cases some individual features were beneficial to the customer and even though the MVP itself was not successful enough for further development. The customer, and therefore also the business, relied on those few individual features. If the MVP had been discontinued, the customer would have been unhappy, and the companies would have lost significant amount of business they had been able to attract. This led to the situation, where both companies decided to further maintain unsuccessful MVPs, despite the fact that they tied valuable development resources to relatively unprofitable applications when considering the full potential of companies.

4 Discussion

SPLs are typically seen as means to reuse existing assets. They require the separation of a common core from which different products can be efficiently derived over their lifetime. In MVPs, on the other hand, reuse is mostly opportunistic, and the emphasis is in the speed of development and meeting a minimum set of key requirements. Thus, reuse with MVPs only takes place, if a new MVP happens to be similar to the previous ones. Moreover, variability can only be seen in hindsight, by comparing a longitudinal set of features included in designs. For example, in the studied cases, the different MVPs used the same web technologies and even the same UI components. In this sense, both of the cases produced their MVPs by sharing some important core assets, which could be interpreted as a realization of an SPL, if executed properly. In a true SPL, however, the variability is created and evolves over time as well, but in a planned, managed, and controlled fashion.

Keeping multiple MVPs alive simultaneously led to resourcing problems in Movendos and Taplia. Every product was allowed to bloom. We propose that this kind of a situation can be seen as the creation of an unhealthy, accidental SPL, where the scope of the SPL is not managed—that is, where each MVP and their features were all maintained as core assets forever. This inevitably leads to a situation of “death by 1000 cuts”, where the excessively broad scope bleeds the companies of resources. To summarize, while it is true that a Lean Startup

must be focused primarily on the short-term, and an SPL is focused more on the medium and long-term, the short-term focus of the Lean Startup does not abdicate the company from the responsibility of doing cost-benefit analyses [1].

Additionally, we propose that resource spending in the above situation steers the MVPs to be more similar with each other, since more pressure tends to emerge towards reuse than towards closing more MVPs. However, the technical basis of the MVP-driven product line cannot become as solid as an SPL, because the ad-hoc reuse practices will not produce as sustainable and well-thought set of assets. Still, the eventual situations inside the case companies studied in this paper resemble the outcomes of using SPL method, with their multiple MVPs alive and kicking. Thus, the main difference between the SPL and MVP is in how the eventual outcome is understood and valued. In SPLs the outcome is a basis for new products, worth being developed further, and adequately maintained. In MVP, on the other hand, the outcome is often something to be thrown away and used only as a means to learn and understand the customer needs.

To study these similarities between SPLs and Lean Startup's MVP development even further, we also compared the two methods and the risks they involve. We found that creating MVPs introduces similar risks to the organization as developing an SPL. These are discussed in more detail in the following.

Developing useless assets. The risk of developing possibly useless assets can be seen as the leap of faith assumptions taken with the first MVP versions. Also improper validation of the build-measure-learn cycle could lead to creating a new useless MVP version if the assumptions which upon it is based are false. In SPLs, by contrast, similar risk is associated with support for products that will never be produced—in other words, getting the SPL scope wrong.

Reuse related risks. The need for close coordination of reusing assets in an SPL can be seen similar as making unnecessary changes to an MVP, for example switching technologies, even though the switch is useless. This requires larger development effort than the simplest MVP production, and if the technology switch is not directly associated to an MVP hypothesis, it is a wasted effort. The same coordination is needed in SPLs where decisions on when to reuse assets or create new ones are also crucial.

Resource limitations. The limited resources in a startup environment and MVP development are directly related to the way SPLs are used to optimize resource usage by reusing components between different MVP versions. Startup companies have highly limited resources and resource management is critical. SPL theory could be used to guide these decisions in a startup environment.

Volatility and uncertainty risks. Highly volatile and uncertain environments are typical for startup companies that do not yet know their business space fully and are still exploring for the right product. The same problems of exceeding benefits while developing a reusable platform in SPL can be seen as the same risk, as creating reusable MVP components. We cannot clearly say what part should be reused without the use of historical data that recent startups do not have. The uncertainty is also tied to the lack of objective methods for evaluating component reuse in SPLs. The same problem surfaces in MVP development

when the company has to create MVP metrics and hypotheses to evaluate the changes made in the different MVP versions. At the same time component reuse and what to change to the next MVP should be analyzed. This might be highly variable, based on the results of the test MVP, and because one cannot be sure if the MVP test will be a success before executing the tests, planning component reuse becomes extremely difficult.

5 Validity and Reliability

Internal validity is a critical examination of whether the experimental treatment makes a difference; that is, whether the independent variable actually causes the changes seen in the dependent variables being examined [25]. Given that this was a relatively small case study, there was no control group, so it was impossible to run this study as a classic controlled experiment. Hence we can make no inference of causality, but can only observe relations that may merit further study.

External validity is a critical examination of whether the results of the study are generalizable [25]. The external validity of the case studies presented here is hampered by the fact that it is based on just two case study companies. These companies volunteered their data and as such they do not represent a random sample of software startups in Finland. Also as both companies are from Finland this study represents the behaviour of companies in the Finnish operating environment.

6 Lessons Learned

The most important lesson learned from this study is that both Lean Startup and Software Product Lines require long-term strategies to be successful. A focus on gaining short-term benefits by first creating products rapidly and then iterating them quickly results in the achievement of some characteristics of both approaches, but not really in the benefits promised by neither of the approaches. In practice, however, startups—if they are operating on severely limited budgets—will likely choose short-term gains over following either of the approaches by the book simply because there are daily operations and practicalities to take care of. In this case they may experience the worst of both worlds: little strategic advantage is gained, because the assets created can not be broadly reused, and reduced agility over time as the software needing to be maintained grows in size and complexity, while losing its conceptual integrity due to rapid small modifications. The results indicate that it is reasonable to advocate closing MVPs without transferring any code forward in the development.

In the best possible world, combining Lean Startup and SPLs can result in an SPL that is actually the MVP for the company that is being built. However, this requires 1) carefully analyzing and controlling the scope of SPL, and 2) the ability to rapidly validate the business value of the technical construct. However, since creating a successful SPL requires considerable investment, aiming at

such an MVP will be even more complicated and costly than simply creating a more traditional product, which would also in most cases better correspond to Lean Startup ideals. Therefore, while displaying some promise, combining the two approaches is not a straightforward recipe for automatic success. Instead, a successful execution of Lean Startup and SPL creation requires careful thought, strategic planning, and good insights into future trajectories for the MVP. However, we do acknowledge that the MVPs can be created with a faster cycle using common core components, provided by an SPL.

Regarding the literature review performed for this paper, we found that while SPLs have been extensively studied, there are fewer reports regarding the use of the Lean Startup methodology in practice. Consequently, we believe that the experiences reported in this paper help in understanding how Lean Startups work in practice, as well as the risks associated with the approach. However, further research is required to gain more conclusive data on benefits and pitfalls of this combination.

7 Conclusions

Software Product Lines and Lean Startup, via its MVP model, promise increased productivity and reduced time to market. In this paper, we are reporting findings from our case study based on observing two companies and investigating their style of Lean Startup software development, where SPLs were formed by the differing MVP versions. These SPLs closely resemble classic SPLs – the handling of MVP versions, their component reuse, customer adherence, and lifecycle management are similar to SPLs, albeit on a faster time scale. However, the creation of SPLs was at least partially accidental due to mishandling MVP lifecycles, which is the root cause for the creation of an involuntary SPL. This multitude of products then consumes resources that could benefit companies more when invested in a more focused fashion.

Finally, there are numerous directions for future research. To begin with, in this paper we are reporting experiences from Finland only, and therefore extending the research to cover other countries is an obvious possibility. In addition, studying the different funding models of startups and their role in the resulting product and company strategy is also a subject for future research.

References

1. Asundi, J., Kazman, R., Klein, M.: Using economic considerations to choose among architecture design alternatives. Tech. rep., DTIC Document (2001)
2. Beck, K.: Embracing change with extreme programming. *Computer* 32(10), 70–77 (1999)
3. Beck, K.: *Extreme programming explained: embrace change*. Addison-Wesley Professional (2000)
4. Blank, S.: *The four steps to the epiphany*. K&S Ranch (2013)
5. Blank, S., Dorf, B.: *The startup owner's manual*. K&S; Ranch (2012)

6. Clements, P.C., Jones, L.G., Northrop, L.M., McGregor, J.D.: Project management in a software product line organization. *Software*, IEEE 22(5), 54–62 (2005)
7. Fagerholm, F., Guinea, A.S., Mäenpää, H., Münch, J.: Building blocks for continuous experimentation. In: *Proceedings of the 1st International Workshop on Rapid Continuous Software Engineering*. pp. 26–35. ACM (2014)
8. Feitelson, D.G., Frachtenberg, E., Beck, K.L.: Development and deployment at Facebook. *IEEE Internet Computing* 17(4), 8–17 (2013)
9. Gamez, N., Fuentes, L.: Software product line evolution with cardinality-based feature models. In: *Top Productivity through Software Reuse*, pp. 102–118. Springer (2011)
10. Jaaksi, A.: Developing mobile browsers in a product line. *IEEE software* 19(4), 73–80 (2002)
11. van der Linden, Pohl, B.: *Software product line engineering: Foundations, Principles and Techniques*. Springer (2005)
12. van der Linden, F.J., Schmid, K., Rommes, E.: *Software product lines in action: the best industrial practice in product line engineering*. Springer Science & Business Media (2007)
13. Maurya, A.: *Running lean: iterate from plan A to a plan that works.* ” O’Reilly Media, Inc.” (2012)
14. Nobauer, M., Seyff, N., Groher, I., Dhungana, D.: A lightweight approach for product line scoping. In: *Software Engineering and Advanced Applications (SEAA), 2012 38th EUROMICRO Conference on*. pp. 105–108. IEEE (2012)
15. Olsson, H.H., Alahyari, H., Bosch, J.: Climbing the” stairway to heaven”—a multiple-case study exploring barriers in the transition from agile development towards continuous deployment of software. In: *Software Engineering and Advanced Applications (SEAA), 2012 38th EUROMICRO Conference on*. pp. 392–399. IEEE (2012)
16. Olsson, H.H., Bosch, J., Alahyari, H.: Towards r&d as innovation experiment systems: A framework for moving beyond agile software development. In: *Proceedings of the IASTED*. pp. 798–805 (2013)
17. Poppendieck, M.: Lean software development. In: *Companion to the proceedings of the 29th International Conference on Software Engineering*. pp. 165–166. IEEE Computer Society (2007)
18. Poppendieck, M., Poppendieck, T.: *Lean software development: an agile toolkit*. Addison-Wesley Professional (2003)
19. Ries, E.: *The Lean Startup*. Penguin New York (2011)
20. Runeson, P., Höst, M.: Guidelines for conducting and reporting case study research in software engineering. *Empirical software engineering* 14(2), 131–164 (2009)
21. Santos, W.B., de Almeida, E.S., de L Meira, S.R.: Tirt: A traceability information retrieval tool for software product lines projects. In: *Software Engineering and Advanced Applications (SEAA), 2012 38th EUROMICRO Conference on*. pp. 93–100. IEEE (2012)
22. Schwaber, K.: Scrum development process. In: *Business Object Design and Implementation*, pp. 117–134. Springer (1997)
23. Schwaber, K., Sutherland, J.: *The scrum guide*. Scrum Alliance (2011)
24. Wu, Y., Peng, X., Zhao, W.: Architecture evolution in software product line: an industrial case study. In: *Top Productivity through Software Reuse*, pp. 135–150. Springer (2011)
25. Yin, R.K.: *Case study research: Design and methods*. Sage publications (1994)

Model-Based Technology of Software Development in Large

Jaan Penjam and Enn Tyugu

Institute of Cybernetics at Tallinn University of Technology
 Akadeemia tee 21, 12618 Tallinn, Estonia
email: {jaan | tyugu}@cs.ioc.ee

Abstract. The present work describes a technology for developing software in unique and large projects. The present model-based technology supports the projects where a single software product is developed. This is different from the block languages and model-based software tools on the market, which provide a set of components where the reusability of the components is an important requirement. A distinguished feature of the technology is a support that it gives to the software design at an early stage of the design process. The design process begins on the architectural level where implementation details can be ignored. Components are introduced considering their functionality, but the implementability of a component is taken into account at the early stage of the design process only based on an experience of a designer.

1 Introduction

The present paper describes a technology for developing software in unique and large projects. Contrary to the model-based software tools on the market, which support the development of a set of components where the reusability of the components is an important requirement, the present tool and technology support the projects where a single software product is developed. The reusability is a beneficial, but not necessary property of the components designed and developed with this technology.

A distinguished feature of the technology is a support that it gives to the knowledge-based design of software at an early stage of the design process. One can say that *the design process begins on the architectural level where implementation details can be ignored*. Instead of classes, components are introduced. The implementability of a components can be taken into account only based on an experience of a designer. This design technology is intended to be analogous to the architectural design in other engineering areas like civil engineering or mechanical engineering.

Implementation of a component results in a class, but an implemented component has also a formal specification used in composition of the software system – the metainterface. Beside that, a component may support (local) protocols for communicating with other components. A component can be considered as a knowledge module, or even an agent, operating in a coordinated way with other components.

2 Visual description of software architecture

We present here a definition and a notation of knowledge module that can be used for describing software architecture on the knowledge level. A *knowledge module* is considered as a pair of sets: a set S of notations (objects) and a set M of denotations (meanings of notations) together with a notation-denotation relation between these sets. (This gives interpretation of the notations.) Also means to perform operations on the set S must be given, although we do not specify these means here, see details in [14]. They are specific to every knowledge module, and can be abstractly represented as inference rules. An abstract representation of a knowledge module is a deductive system with interpretation, see S. Maslov [11].

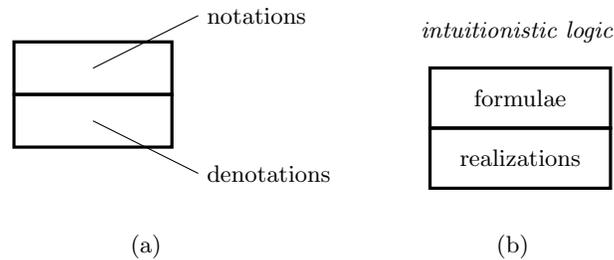


Fig. 1: Visual notation of a knowledge module (a) and of knowledge module of logic (b)

Visual representation of a knowledge module is a pair of rectangles as shown in Fig. 1a. An example of a meaningful knowledge module is given in Fig. 1b. It is a knowledge module of intuitionistic logic.

Knowledge modules can be bound in various ways: hierarchically, semantically and operationally [14]. Let us have two knowledge modules K_1, K_2 with sets of objects S_1, S_2 , sets of meanings M_1, M_2 and notation-denotation relations R_1, R_2 respectively.

Hierarchical connection. We say that knowledge modules K_1 and K_2 are hierarchically connected, iff there is a relation R between the set of meanings M_1 and the set of objects S_2 , and strongly hierarchically connected, iff there is a one-to-one mapping between the elements of a subset of M_1 and of a subset of S_2 , see Fig. 2. A hierarchical connection of knowledge modules can be observed quite often in real life. An example is deductive program synthesis. The knowledge system of logic and the calculus of computable functions (CCF) are strongly hierarchically connected, because there exists a Curry-Howard isomorphism of proofs and formulae as types, see Fig. 2b.

Semantic connection. Knowledge modules that have one and the same set of meanings are *semantically connected*. This is the case, for instance, with classical

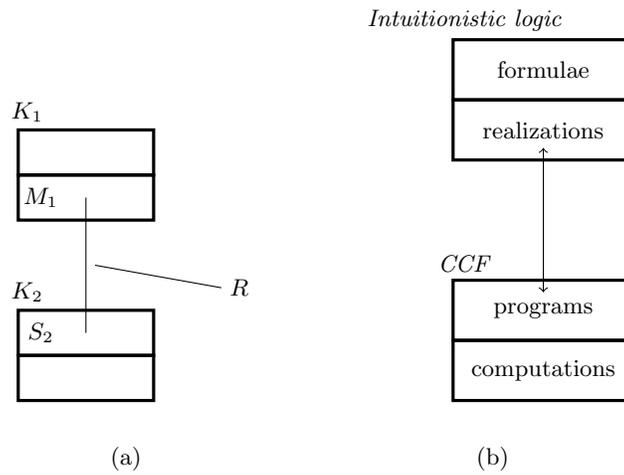


Fig. 2: Notation of hierarchical connection (a) and example of strongly hierarchically connected knowledge modules of deductive program synthesis (b)

logic systems that have different sets of inference rules, or even with natural languages that belong to closely related cultures (i.e. that have the same set of meanings). Graphical notation of semantic connection is shown in Fig. 3a.

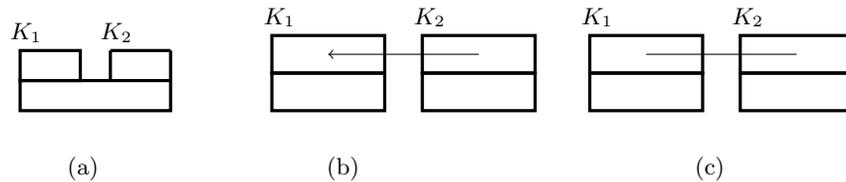


Fig. 3: Semantic connection (a), operational dependency (b) and operational connection (c)

Operational dependence and operational connection. Knowledge module K_1 is *operationally dependent* on a knowledge module K_2 , if some of its derivation rules use K_2 in deriving a new object, i.e. the result of derivation in K_1 depends on knowledge processing in K_2 , Fig. 3b.

Knowledge modules K_1, K_2 are *operationally connected*, if K_1 is operationally dependent on K_2 , and K_2 is operationally dependent on K_1 . Graphical notation of this connection is shown in Fig. 3c. The notations presented here are used for the architectural design of software at the first stage of a software project.

3 Architectural design of software

The first stage of design of a software system is its architectural design. At this stage, only the most general structure of the system is developed and specified by the knowledge architectural means. It is important to decide, which knowledge modules are needed, and how they will be connected. The input for this stage is a specification of functional requirements.

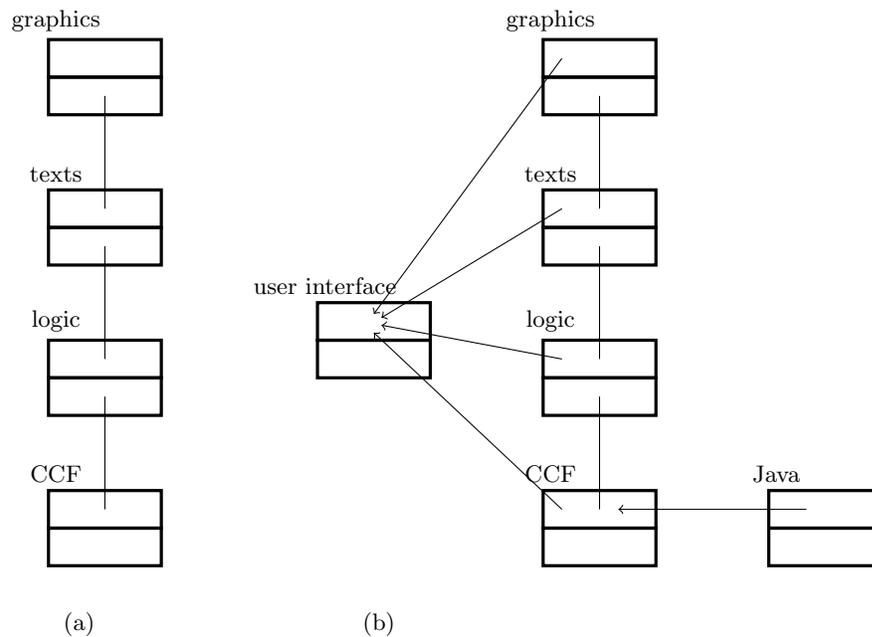


Fig. 4: Basic knowledge modules of CoCoViLa (a) and complete knowledge architecture of CoCoViLa (b).

We present our technology on an example of design and development of a model-based software tool CoCoSynth that includes also a program synthesis functionality. For a specification of functional requirements, we refer to [4] and the documentation of an existing tool CoCoViLa in web¹. This means that we are applying our technology to the development of a new version of CoCoViLa.

It follows from the documentation of CoCoViLa that there must be two hierarchically connected knowledge modules *logic* and *CCF* that support the deductive program synthesis as it has been shown in Fig. 2b. This is the core of the software tool. We see also from the documentation that the input of the tool is not in a logical language, but in a textual domain-oriented specification

¹ <http://cocovila.github.io/>

language and/or in a visual language. This adds two more knowledge modules: *texts* and *graphics* to the architecture of the tool. These knowledge modules are hierarchically connected with the logical module, Fig. 4a. Operation of the tool is controlled by a user through a visual *user interface* that is also a knowledge module. We can describe this connection by operational dependence of this module on other modules. If we wish to show also the functionality that reflects the usage of Java at runtime, then we have to add a *Java* knowledge module tool as shown in Fig. 4b.

4 Design of components

This stage includes a conceptual analysis of the domain, and can be called also domain engineering, although the domain is presented by a single new software product in this case. Having an architectural description of software, one can take the knowledge modules of this description as components in the first place. However, these components may be too large or too small. The knowledge modules can be sometimes divided into smaller components, or collected into a single component, considering their functionality and realisability. A separate component may be required for representing a hierarchical relation between the knowledge modules. This will be demonstrated on the example below.

Table 1: Table of components

Notation	Input	Output	Comments
specification			a data structure
problem			a data structure
algorithm			a data structure
code			a data structure
Controller			will be a superclass
EDITOR		specification	
PARSER	specification	problem	
PLANNER	problem	algorithm	
GENERATOR	algorithm	code	
EXE	code		
ALGORITHM VISUALISER	algorithm		

In the present example, we keep the user interface knowledge module as a separate component *controller*. The knowledge modules *graphics* and *texts* will be joined into a single component *editor*, in order to facilitate their usage by *controller*, because the latter will produce the text in parallel with the graphics. We keep the knowledge module of logic as a separate component *planner*. This name reflects the purpose of the logic component that synthesises an algorithm of the software product. The hierarchical connection between *editor* and *planner*

will be represented by a separate component *parser* which transforms a text into logical formulae. The computational knowledge module CCF gives a component called *exe*. Also a hierarchical relation between *planner* and *exe* will be represented by a separate component generator which generates a Java program that has to be run. We introduce an additional component *algorithmVisualiser* for the visualisation of synthesised algorithms.

5 The CoCOViLa system overview

The tool used in our technology must support convenient implementation of components, preferably visual specification of software models and automatic code generation from a model. The tool CoCoViLa [4] used in our technology consists of two almost independent programs: *Component Editor* and *Specification Editor*. The first is a relatively small program for developing visual images of components, specifying their general properties and collecting them in domain-oriented packages.

The specification editor, referred further as CoCoViLa itself, uses a package of components for specifying tasks in respective domain or, in the present case, specifying a software model, and it supports code generation from the model. Essential working principles of this tool are the following:

- besides a visual representation, each software component has two parts: 1) logical specification of the component (LO), called metainterface, 2) object-oriented (OO) realisation of the component – a Java class;
- a component is called metaclass, because after program synthesis it may be transformed into several different classes;
- object-oriented and logical parts have separate namespaces, except for names of methods from OO used in LO – this enables one to write specifications of components almost independently of their Java realisations;
- OO and LO have a common type system.

Metainterface has a precise logical semantics given as a set of formulas – axioms with realisations given by methods of its Java class. These formulas constitute a theory in intuitionistic logic that is used by structural synthesis of programs [12] for automatic construction of programs in CoCoViLa.

Components can be defined hierarchically, i.e. a metainterface of a component may contain components whose types are given by metaclasses, i.e. by other components. Also equations, as well as some other language constructs can be used in the metainterface. A metaclass may consist of a metainterface only, e.g. in a case when computations are specified by equations. Metainterface is written in a specification language, and it is included as a comment in the Java class of the component between `/*@ . . . @*/`. We give an example of a metaclass now. The metaclass **And** represents a logical element (and-gate) for signals represented by 0 and 1. It includes a Java method `calc` as a realisation of the axiom `in1, in2 -> out`.

```

01 public class And {
02     /*@
03         specification And {
04             int in1, in2, out;
05             in1, in2 -> out {calc};
06         }@*/
07     public int calc( int x, int y ) {
08         return Math.min(x, y);
09     }
10 }

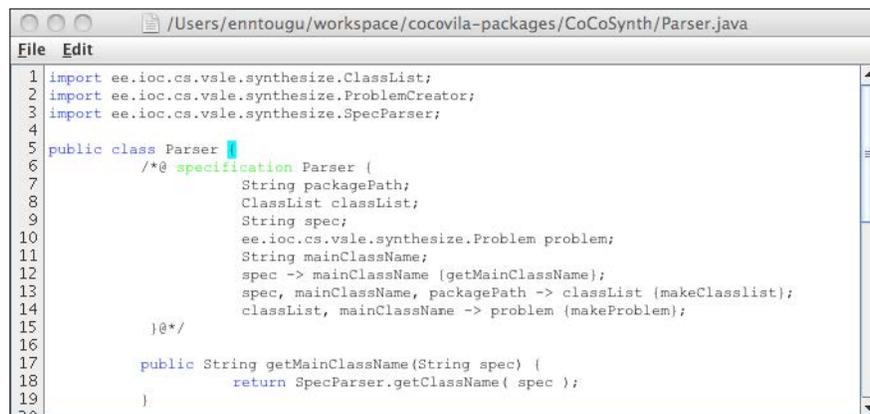
```

Lines 02 to 06 of the example are a metainterface. Line 04 is a variable specification of integer variables, and line 05 is an axiom. All Java classes and metaclasses can be used as type specifications. Lines 07 and 08 are in Java, and describe the method `calc` referred to in the axiom. An axiom is a formula of a conjunctive-implicative fragment of intuitionistic propositional logic, where commas represent conjunction symbols.

6 Implementation of components

Each implemented component has graphics, metainterface and Java class. It is reasonable to start with writing a metainterface, although the development of all three components can occur in parallel.

Writing metainterfaces. A metainterface is written in the specification language and included in the class of a component as a comment. Lines 6 to 15 of Fig. 5 show a metainterface for the component `PARSER` as an example. The metainterface shows that *problem* can be computed in three steps specified by the axioms in lines 12, 13 and 14.



```

/Users/enntougu/workspace/cocovila-packages/CoCoSynth/Parser.java
File Edit
1 import ee.ioc.cs.vslc.synthesize.ClassList;
2 import ee.ioc.cs.vslc.synthesize.ProblemCreator;
3 import ee.ioc.cs.vslc.synthesize.SpecParser;
4
5 public class Parser {
6     /*@ specification Parser (
7         String packagePath;
8         ClassList classList;
9         String spec;
10        ee.ioc.cs.vslc.synthesize.Problem problem;
11        String mainClassName;
12        spec -> mainClassName {getMainClassName};
13        spec, mainClassName, packagePath -> classList {makeClasslist};
14        classList, mainClassName -> problem {makeProblem};
15    }@*/
16
17    public String getMainClassName(String spec) {
18        return SpecParser.getClassName( spec );
19    }
20

```

Fig. 5: Metainterface of PARSEr

Developing graphics. For developing graphics of a component one uses the *Class Editor* program of CoCoViLa that supports the development of visual representation of the component, definition of ports for binding components, as well as description of properties of component as it is described in the documentation of the Class Editor.

Implementing methods. Class of a component is created already when a metainterface is written. This class must be completed by writing all methods referred to in axioms of the metainterface. In the case of the component `PARSER` in our example, the methods are

```
getMainClassName ,
makeClassList ,
makeProblem .
```

It is obvious that in these methods, other methods may be used that need to be implemented as well. This is a usual program development in Java. For instance, we see that the method `getClassName` of a class `SpecParser` is used in addition in the method `getMainClassName`.

As our example is in essence redeveloping CoCoViLa, it is reasonable to use its classes as much as possible for the implementation of methods. We have used the source code of CoCoViLa, that consists of 240 Java classes, totally about 30K lines of code. These classes were developed without any restrictions on programming in Java. Our experiment has shown that most classes could be used as is, or with only minor changes, depending on the developed metainterfaces.

Static model of software. When components are implemented, a high-level structural model of software can be written immediately in CoCoViLa specification language or drawn as a diagram. This is called static model. For the synthesis, the *static model* is automatically translated into a metainterface of a new Java class.

The static model is a specification for the tasks that the system has to perform. Programs for the tasks are synthesized automatically. When translated into logic, the static model describes a theory representing all possible computations on the model. Let us denote by G the set of the goals that describe the tasks solvable on the model. Each goal is written in the form

$$x_1, x_2, \dots, x_m \rightarrow y_1, y_2, \dots, y_m,$$

where $x_1, x_2, \dots, x_m, y_1, y_2, \dots, y_m$ are variables of the static model, e.g. *specification, problem, algorithm* etc. in our example. These variables are considered in logic as propositional variables of the theory, and commas as conjunction symbols. Fig. 6 shows the static model where all components described in the previous section are present.

Dynamic model. The static model describes only tasks that the software can perform, but not a user interface to invoke these tasks. In order to describe the

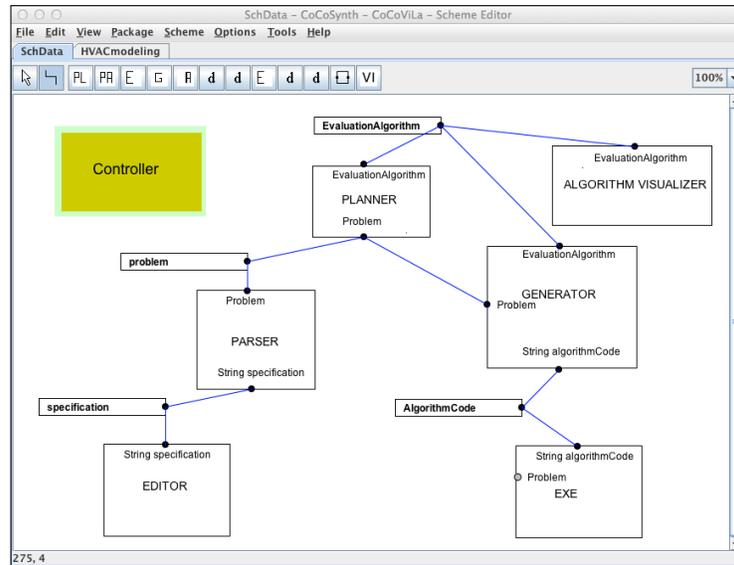


Fig. 6: Static model of CoCoSynth

interaction between a user and the system, a *dynamic model* of GUI is introduced. This model is specified as a statechart. This statechart may be explicitly given as a part of a requirements specification, or it may be implicitly described by other requirements. In the latter case, the development of the statechart is performed in a conventional way, e.g. as recommended by some UML-based technology.

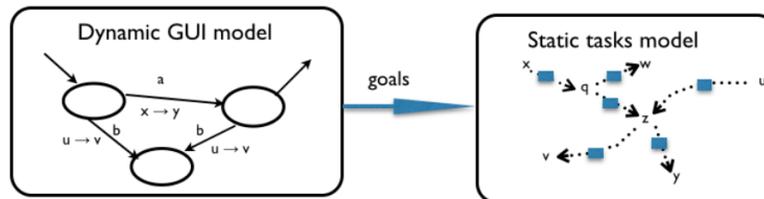


Fig. 7: Static and dynamic models are connected by goals specifying the tasks performed on the static model

Each transition t in the dynamic model is marked by an event $e(t)$ and a goal $g(t)$. The event is either a user action (e.g. pushing a button) or an event created by the system that triggers some transition. The goal describes a task to be performed on the static model when the transition t occurs. The tasks for all transitions must be solvable on the static model, i.e. for every task t must

be $g(t) \in G$. Fig. 7 shows abstractly a fragment of the dynamic model, and the connection between the static and dynamic models with the tasks $u \rightarrow v$, $x \rightarrow y$ and events denoted by a and b .

The dynamic model must be implemented as a component that becomes a superclass for the class of the static model. In our example, this is the component *controller*. Fig. 8 shows a part of the dynamic GUI model for our example with events created by user commands *Run*, *Compute goal*, *Compute all*, *Scheme*. One can find meaning of these commands from the documentation on CoCoViLa. The respective tasks for the commands are

$$\begin{aligned} c &\rightarrow \textit{specification} \\ c, \textit{specification} &\rightarrow \textit{algorithm} \\ c, \textit{algorithm} &\rightarrow \textit{code} \\ c, \textit{specification}, \textit{goal} &\rightarrow \textit{algorithm} \\ c, \textit{specification} &\rightarrow \textit{results} \\ c, \textit{specification} &\rightarrow \textit{schemeMenuOpen}, \end{aligned}$$

where c is a control and context variable.

1

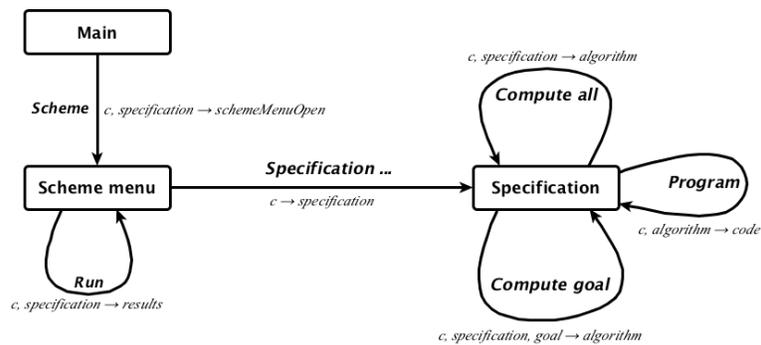


Fig. 8: A part of the dynamic model of CoCoSynth

7 Implementing user interface

The user interface is implemented as a component as described above. Its functionality is described by the dynamic model that is in the form of a statechart. Java technology can be used in full in this implementation. However, implementation of a connection between of the dynamic and the static model by means of events and tasks deserves a special attention, and here are some hints for this.

1. The user interface component (*controller* in our case) can be implemented as a superclass for the static model. This enables one to use the names of variables of the static model for representing the tasks to be solved on it.
2. A task is always invoked by a respective event (see the dynamic model). The event is handled by an event handler in Java. Hence a call of program for a task must be included in the event handler.
3. As a program for a task is synthesised automatically using SSP, each task must be described by an implication in an axiom whose implementation creates event handlers. In our example, the event handlers are created by the method `initGUI`. Its axiom, included in the metainterface of *controller*, is as follows:

```
[c -> specificatin], [c, specification -> algorithm],
[c, algorithm -> code], [c, specification, goal -> algorithm],
[c, specification -> results],
[c, specification -> schemeMenuOpen],
c -> doneinitGUI{initGUI};
```

8 Synthesising the software

When the static model is implemented including the dynamic model as a superclass, a new program can be synthesised automatically by giving command *Run* from *Scheme* menu of the CoCoViLa's main window. This means bootstrapping CoCoViLa in our CoCoSynt example.

The bootstrapping process and its results can be explained in Fig. 9. It shows the windows that open during the bootstrapping and running the bootstrapped tool. The two upper windows belong to CoCoViLa, they are a diagram of the static model of the new CoCoViLa (CoCoSynth) and the Java code of CoCoSynth synthesized in CoCoViLa. This completes the development of the new tool CoCoSynth.

The static model differs from the model in Fig. 6 by more compact presentation, where ports of the components are directly connected with each other without intermediate data components. Also an extra component *GUIactions* has been added to the model. It includes additional action listeners for the *controller*. When command *Run* is given from *Scheme* menu in this window, CoCoSynth is synthesized and started as well. The lower windows belong to the synthesized CoCoSynth. The leftmost window is the main window of CoCoSynth, it opens automatically after Run command given from CoCoViLa. It can be used for loading packages, drawing diagrams and performing computations. We see a package Gearbox loaded, and a diagram with several wheels, a motor and two visualizer components in it. After invoking *Specification...* command from *Scheme* menu of CoCoSynth, a new specification window opens according to the dynamic model.

This window can be used for textual editing of specification, for synthesizing application programs, for running these programs and for some other actions. After the command *ComputeAll* from the specification window, this window will show the synthesized Java program (partially visible in the second window

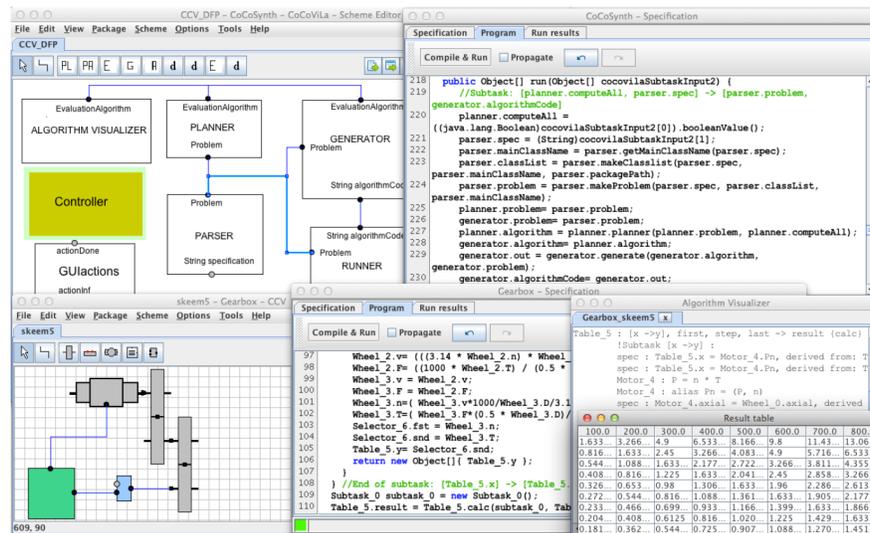


Fig. 9: Bootstrapping CoCoViLa

from left). After the command *Compile & Run* from this window, the synthesized program is compiled and executed and, in our example, a table of results is calculated and visualized (visible in the lower right window). Also a small part of algorithm for calculations on gearbox is visible from behind the result window. As the structural synthesis of programs used in CoCoViLa is fast, the whole process of bootstrapping and creating these windows takes less than 30 seconds (including interaction with a user, e.g. opening windows from GUI, etc) in the case when the diagrams are ready and loaded from some repository.

9 Related work and discussion

Model-based software development (MBSD) has been increasingly popular during several decades and a number of tools supporting implementation of model-driven principles have developed. The comprehensive study of achievements in the field can be found, for example, in the book [1]. Its most successful applications are in simulation software for automotive engineering, space technology and others, there are well known specialized products, mainly in simulation domain like Simulink [6] and more recently several systems like MetaEdit+[8], Modelica [3] etc.

UML² is de facto standard not to be ignored in model-based software development. Our technology uses two kinds of UML diagrams: statecharts and component diagrams. However, we have implemented the semantics of these diagrams in a way that guarantees automatic code generation from them for large

² www.uml.org

Java software including more than two hundred classes. This is a difference from the existing examples of code generation from UML models, e.g. [1], where only relatively small examples have been implemented.

Considerable amount of work is being done in improving the existing UML-based approaches with the aim of providing automated support to the software development[2] and language development [13]. One of the most successful approaches in this direction has been made by the Eclipse community. Eclipse Modelling Project (EMP) [5] that includes Eclipse Modelling Framework (EMF), Graphical Modelling Framework (GMF) and the Generative Modelling Tools (GMT) is a relatively new collection of technologies for building domain specific languages (DSLs). Generally speaking, EMP is a powerful set of tools, but it requires a lot of effort to develop a working DSL from scratch.

The system CoCoViLa used in this paper belongs to the another research direction in the MBSD – Model-Integrated Computing (MIC) that addresses the problems of designing, creating, and evolving information systems by providing rich, domain-specific modelling environments including model analysis and model-based program synthesis tools [7]. CoCoViLa, integrates tools for specification and implementation of visual DSMLs (both for abstract and concrete syntax together with some well-formedness constraints) and translators that implement mappings from syntactic form of models into formal theories in the semantic domain. The latter is a domain-independent framework for representing semantics (components and relations) of domain-specific models as sets of axioms in intuitionistic logic calculus equipped with specific inference rules (rules for structural synthesis of programs (SSP)) for generating of algorithms in a formal theory corresponding to the domain-specific model [12].

CoCoViLa has been applied mainly as a model-based simulation tool [10]³. The novelty of this paper is to apply this approach for design and development of software product, including its static and dynamic aspects as well as user interface.

A technology with automatic code generation from models has been developed by Steven Kelly and Juha-Pekka Tolvanen. Their technology and tool MetaEdit+ have been well described in literature [9]. Our software technology is similar to that. However, we use deductive program synthesis for code generation. This makes the implementation of components much faster, because no generator development is needed.

Bootstrapping of software tools has been popular since early days of computing. It was helpful in developing compilers for new hardware, when there was no tool support on hardware. It is a non-trivial test of the language compiled. A list of languages having self-hosting compilers today includes 41 names⁴. From this perspective, bootstrapping of CoCoViLa can also be considered a good test of our technology.

³ see also <http://cocovila.github.io/>

⁴ [https://en.wikipedia.org/wiki/Bootstrapping_\(compilers\)](https://en.wikipedia.org/wiki/Bootstrapping_(compilers))

10 Summary

The presented technology is summarised in Fig. 10. It shows the roles of different experts: domain expert, system engineer, graphic expert and code developer in a project developed in accordance with this technology. We see that after the requirements specification, developing the knowledge architecture and components specification, one can proceed by developing in parallel components metainterfaces and graphics as well as a dynamic model. Components code could be developed in parallel with graphics and metainterfaces as well, but one will need the dynamic model for writing the user interface component. The most important role has a system engineer, who performs six steps of the project. Also coordination of the project as a whole belongs to his role.

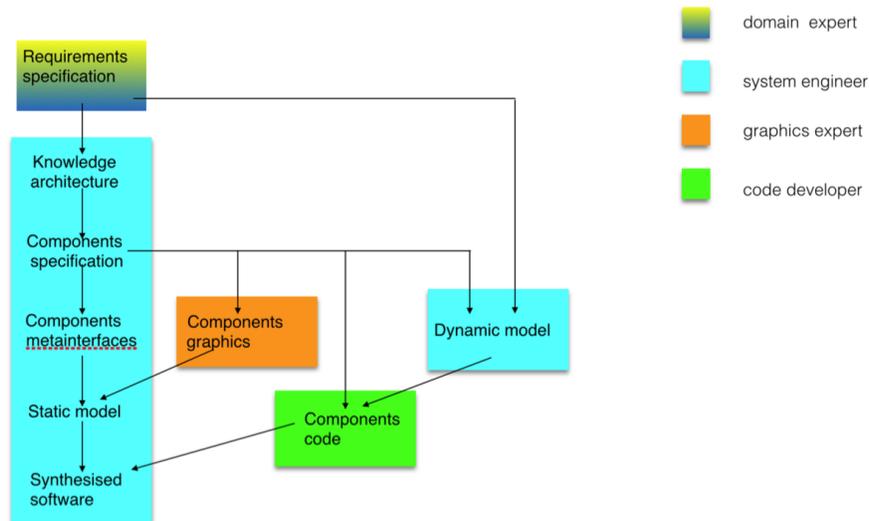


Fig. 10: Software technology step by step

Debugging and testing are not shown in Fig. 10. After developing the static model, its completeness can be tested on tasks prescribed by the dynamic model even before the code of components has been developed. After developing the code of user interface, one can test the interaction of dynamic and static model even without of other components codes. As usual, repetitions of some steps of the technology will be needed when errors are detected.

Acknowledgements

This research was supported by Estonian Research Council institutional research grant no. IUT33-13, and by the ERDF through the ITC project MBJSDT and Estonian national CoE project EXCS.

References

1. Brambilla, M., Cabot, J., Wimmer, M.: Model-Driven Software Engineering in Practice. Synthesis Lectures on Software Engineering, Morgan & Claypool Publishers (2012), <http://dx.doi.org/10.2200/S00441ED1V01Y201208SWE001>
2. Engels, G., Soltenborn, C., Wehrheim, H.: Analysis of UML activities using dynamic meta modeling. In: Bonsangue, M.M., Johnsen, E.B. (eds.) Formal Methods for Open Object-Based Distributed Systems, 9th IFIP WG 6.1 International Conference, FMOODS 2007, Paphos, Cyprus, June 6-8, 2007, Proceedings. Lecture Notes in Computer Science, vol. 4468, pp. 76–90. Springer (2007), http://dx.doi.org/10.1007/978-3-540-72952-5_5
3. Fritzson, P.: Introduction to Modeling and Simulation of Technical and Physical Systems with Modelica. Wiley (2011), https://books.google.ee/books?id=413e_S4DI-IC
4. Grigorenko, P., Saabas, A., Tyugu, E.: Cocovila – compiler-compiler for visual languages. Electron. Notes Theor. Comput. Sci. 141(4), 137–142 (Dec 2005), <http://dx.doi.org/10.1016/j.entcs.2005.05.009>
5. Gronback, R.C.: Eclipse Modeling Project: A Domain-Specific Language (DSL) Toolkit. Addison-Wesley Professional, 1 edn. (2009)
6. Jain, S.: Modeling & Simulation Using MATLAB Simulink (With CD). Wiley India Pvt. Limited (2011), <https://books.google.ee/books?id=qpV9ygAACAAJ>
7. Karsai, G.: Lessons learned from building a graph transformation system. In: Engels, G., Lewerentz, C., Schäfer, W., Schürr, A., Westfechtel, B. (eds.) Graph Transformations and Model-Driven Engineering - Essays Dedicated to Manfred Nagl on the Occasion of his 65th Birthday. Lecture Notes in Computer Science, vol. 5765, pp. 202–223. Springer (2010), http://dx.doi.org/10.1007/978-3-642-17322-6_10
8. Kelly, S., Lyytinen, K., Rossi, M., Tolvanen, J.: Metaedit+ at the age of 20. In: Jr., J.A.B., Krogstie, J., Pastor, O., Pernici, B., Rolland, C., Sølvberg, A. (eds.) Seminal Contributions to Information Systems Engineering, 25 Years of CAiSE, pp. 131–137. Springer (2013), http://dx.doi.org/10.1007/978-3-642-36926-1_10
9. Kelly, S., Tolvanen, J.: Domain-Specific Modeling - Enabling Full Code Generation. Wiley (2008), <http://eu.wiley.com/WileyCDA/WileyTitle/productCd-0470036664.html>
10. Kotkas, V., Ojamaa, A., Grigorenko, P., Maigre, R., Harf, M., Tyugu, E.: Cocovila as a multifunctional simulation platform. In: Liu, J., Quaglia, F., Eidenbenz, S., Gilmore, S. (eds.) 4th International ICST Conference on Simulation Tools and Techniques, SIMUTools '11, Barcelona, Spain, March 22 - 24, 2011. pp. 198–205. ICST/ACM (2011), <http://dx.doi.org/10.4108/icst.simutools.2011.245553>
11. Maslov, S.Y.: Theory of Deductive Systems and Its Applications (Foundations of Computing). MIT Press (1987)
12. Mints, G., Tyugu, E.: Propositional logic programming and priz system. J. Log. Program. 9(2&3), 179–193 (1990), [http://dx.doi.org/10.1016/0743-1066\(90\)90039-8](http://dx.doi.org/10.1016/0743-1066(90)90039-8)
13. Selic, B.: A systematic approach to domain-specific language design using UML. In: Tenth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2007), 7-9 May 2007, Santorini Island, Greece. pp. 2–9. IEEE Computer Society (2007), <http://dx.doi.org/10.1109/ISORC.2007.10>
14. Tyugu, E.: Understanding knowledge architectures. Knowledge-Based Systems 19(1), 50 – 56 (2006), <http://www.sciencedirect.com/science/article/pii/S0950705105000936>

Requirements management in GitHub with a lean approach

Risto Salo, Timo Poranen, and Zheyang Zhang

University of Tampere, School of Information Sciences, Tampere, Finland
risto.salo@gmail.com, {timo.t.poranen, zheyang.zhang}@uta.fi

Abstract. GitHub is an online platform for hosting projects that use the Git revision control system to manage code. Its lightweight issue tracker helps to maintain lists of issues identified during the development process, including bugs, features, or other software artifacts. Although issue tracking software has been practically used in software industry, studies on using it to manage requirements remain insufficient. This paper tackles the issue by presenting a semi-formal guideline for managing requirements in agile software development projects using GitHub. The guideline is evaluated on a theoretical level by analyzing how well it guides to manage requirements and fits in an agile software development setting. It is compared against lean principles. In addition, the guideline is put into use in a case study. The studies indicate that the guideline and GitHub are well-suited for requirements management in an agile environment.

Keywords: Requirements management; GitHub; Lean software development; guideline

1 Introduction

GitHub [6] is a rapidly growing Internet based service which describes itself as a social coding platform. In the beginning of 2013 GitHub had almost 5 million source code repositories, and in a year it doubled the figure to 10 million. An average of 10000 new users subscribe every weekday. GitHub is not a place just for individual users; its lightweight and easy to use features of managing source code and supporting collaboration with developers have attracted much attention, and been widely recognized by notable organizations, including Amazon, Twitter, Facebook, LinkedIn and even the White House. Although GitHub is mostly used for code, its issue tracker and wiki has been used for requesting and monitoring software features in ad hoc ways. How could these tools be used to manage requirements without a need to utilize another tool forms an interesting issue in software projects using GitHub. This paper tries to tackle it by introducing a guideline for requirements management (RM) using GitHub's native features. The guideline was put to use in a software development project and also evaluated based on the objectives of the requirements management and lean principles. This article is based on Salo's [13] thesis.

Lean ideology derives from Japanese car manufacturing from the middle of 1950s. In the beginning of the 21st century “being lean” has received a lot of hype in the software engineering field after Poppendiecks’ [12] converted its principles to a suitable format for software development. The enthusiasm towards lean started to cumulate a little after agile methodologies hit through. Agile methodologies are a response to the failure of coping with changes in the waterfall-like methods, and their goals are aligned with lean principles. In this study lean principles offer an insight how well the guideline for RM shapes into an agile environment that does not necessarily rely on a defined agile methodology like Scrum or Extreme Programming. Similar topics couldn’t be identified either within the scientific researches or from practitioners making the study novel. Although the problem domain and its solution are quite specific, the evaluation succeeds in combining lean principles to the objective of RM. The case study reveals that the whole project team values the guidelines defined approach. On the other hand the evaluation points out that the guideline and GitHub achieve also in a theoretical level. Overall the results prove that GitHub can be used successfully for agile requirements management when a systematic guide is applied.

The rest of this paper is organized as follows. Next section gives an introduction to GitHub. Section 3 addresses common RM practices in agile projects. Section 4 describes the guideline and its usage. In Section 5 the guideline is evaluated. The last section concludes the work.

2 GitHub

The central concept in GitHub is a repository. GitHub’s key aspect is its version control mechanism, so it is natural that other components are built upon this feature. The bulk of a repository’s main page consists of the files and folders inside the version control system. There are navigation links to the subcomponents of the repository, and free-text search function to search multitude of things inside GitHub’s repositories.

In GitHub all issues are created and tracked in an integrated issue tracker. An issue is a very vague concept; basically it is a task. However, issues can range from memos to requirements. In this study, the term issue is used as a higher level concept, and includes both requirements and tasks. A task is a concrete item that must be done in order to fulfill defined requirements. Requirements hold, in predefined format, the goals, business objectives and user requirements for the software. They are more abstract than tasks. Both can be divided to main and sublevel. Sublevels are to be used when a requirement or task is so large, that it is semantically wise to divide it to subcomponents. For example a task can have different subtasks for designing UI, creating it, designing architecture and implementation.

As shown in Figure 1, the issue tracker is composed of three components shown in three areas. The top of the screenshot is the issue tracker’s inner navigation. Provided options are the default view, milestone view, labels, filters and

a link to create a new issue. Below this area are exclusive general filters. In the bottom are the issues. Name, assigned labels, identification number, creator and elapsed time from the creation are displayed. Every issue has a checkbox, for carrying out actions like assigning a label without a need to open the issue.

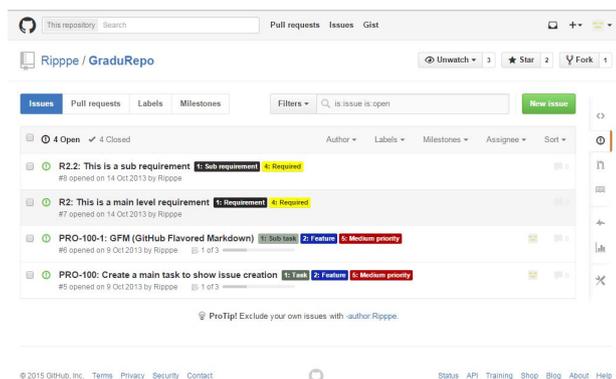


Fig. 1. The issue tracker view in GitHub.

The issue tracker provides a separate view for creating a new issue. User must give a name and a description of the new issue, but he can also assign a developer, desired milestone and labels. Labels are small tags that can be assigned to issues. Every label acts as a filter to the issues. Milestones are iterations and as such issues can be linked to them. An issue can always belong to a maximum of one milestone. User can observe and manage the milestones. User can toggle open and closed milestones and also milestones with or without any issues assigned to him. Every milestone can be edited, closed or deleted. Milestone's name, description and due date are also presented.

Filters refer to different filtering options available in the issue tracker. These filters can be divided to four categories: personal, milestone, label and state. Personal filters ("Assignee" and "Author" selections) are used to filter issues that are related with the current user. Milestone filter is for filtering issues associated to a specific milestone. The label filter limits the issues to those that have all the selected labels assigned to them. The state filter is for filtering open and closed issues. There is also a search box that provides a way to make more elaborated filter queries using a special syntax.

When an existing issue is opened, a detailed info view is displayed. All data in the view is editable. The state is notified below the name of the issue. Other relevant information includes subscription to the issue and participants of the issues. Subscribing to the issue means getting notifications about changes to the issue. The person creating the issue is automatically subscribed. Comments and references to the issue are visible below the description in a chronological order.

The last subcomponent of the repository is a wiki. Each repository has an own wiki collection that acts like any ordinary wiki instance. It consists of one or more pages that are created using one markup from the list of possibilities.

3 Requirements management

Paetsch and others [11] have noted that the upfront documentation of requirements and the inflexible change management process in a traditional software development process and as such, is not compatible with agile practices. Agile practices emphasize the communication and collaboration with customers, the working software and responding to changes. The questions of how requirements are managed in an agile environment don't have unambiguous answers. As different agile methods embrace different aspects of agile principles, so do researchers and requirements analysts. However some common practices can be declared.

Stakeholder involvement forms one of the core principles in agile methods, and a key factor for project success [2, 4, 8, 11]. Though the traditional requirements engineering also embraces customer contribution, it is valued even more in an agile environment. Communication barriers often exist in the interaction between developers and stakeholders, due to different working experience, mind sets and background knowledge. Different techniques provide support for communication and collaboration with stakeholders. Besides the traditional ones such as interviews, face-to-face conversations, brainstormings, observation, etc., techniques such as prototypes, working software [2] or the tests and review meetings offer varying ways for stakeholders to understand their product and propose changing requirements. It also helps developers to gain an in-depth understanding of the application domain [8]. Such an intense collaboration in agile software development processes has been reported to lower the need for major changes in delivered products [2].

Replacing heavyweight documentation with a lightweight alternative forms another common practice. Documentation should not be neglected because it is used for knowledge transfer, but it should be toned down to the minimum feasible level due to its cost ineffectiveness [11]. Instead of a full requirements document, story cards or user stories form the most common form of user requirements [11, 17]. They are usually either physical or electronic representations of cards that include a few sentences describing how the user completes a meaningful task in interaction with the software to be built [3, 15]. The backlog or feature list can then be used to keep the track of stories and their progress [15]. The details of user stories are elaborated just in time, before they are about to be implemented.

As many agile methods take advantage of iterative software development process, the same practice is applied in RM [5, 10, 14]. Working on the increments based on user stories involves interaction with end-users, where, in fact, changing requirements come in the form of users' feedback. A key aspect of the iterative and incremental process mentioned by multiple studies [4, 8, 11] is prioritization. Requirements are ranked according to their priority, and the ranking can be adjusted often before the next implementation iteration [11]. The update and

adjustment is based on users' feedback on the latest increment, and to ensure that the most important and urgent requirements are tracked and implemented first. [8, 2]

To summarize aforementioned three factors, the key aspect for agile requirements engineering is to manage the requirements to implement the most important ones first in order to produce the best possible business value for the customer [5, 11].

As with agile methods in general, it should be remembered that agile requirements engineering does not guarantee success, although correctly used can greatly enhance chances for it. Cao and Ramesh [2] have stated a wide array of possible problems with these practices. Communication is only as effective as its participants. It's hard to create cost and time estimates with iterative requirements. Documentation is easy to be neglected, same as non-functional requirements because they don't necessarily implement visible or otherwise concrete business value.

4 Managing requirements in GitHub - the guideline

4.1 A hierarchy between requirements and tasks

Tasks and requirements can be split to smaller parts. Smaller issues give a more transparent view into what is really wanted. The bigger the issue more likely it contains too wide a topic to fully grasp. This is why we recommend to use the subtasks and sub-requirements. For example the case study contained a requirement R2.2 "There are a total of 10-15 puzzles". A natural way of splitting R2.2 to tasks was to make a task for each puzzle. The tasks could be further split to subtasks representing implementing business logic, the UI and so on.

Splitting could be done endlessly so there must be a limit to how deep the hierarchy can go. For the guideline the maximum depth is four issues: a requirement, a sub-requirement, a task and a subtask. To revise the example, an alternative to creating just one subtask for a business logic would be to create multiple subtasks to depict individual components of the business logic. These subtasks would be direct descendants of the main task.

Subtasks have some drawbacks. Too small tasks cause more work with their creation and maintainability and offer minimal benefits. If developers are experienced, forcing them to create the subtasks might be a waste of time. On the other hand, in some cases it can help a developer further recognize the problem and its different areas. A good thing with subtasks is that the components they affect are easier to identify and trace. A big task could be clogged up with a long description and dozens of comments hindering the process of finding the relevant information.

There are benefits of not using the subtasks. When an issue is referenced, the reference shown in a comment area has the information of how many of issue's task list items are completed. With the subtasks such information cannot be displayed due to the lack of automated hierarchy handling. The second benefit is negating the overhead caused by creating and maintaining the subtasks.

Overall it is impossible to say when the benefits of the subtasks overcome the slight waste caused by their upkeep. The decision about their use should be made case by case. The main point is that all the necessary information is present, logically structured and findable with minor effort. It should be noted that to get the best out of the issues, the whole team must be committed to use them. In some cases developers may find subtasks to be just a nuisance and therefore the attitude towards the guideline might decrease causing a negligent usage. Overall it is hard to say when benefits from defining the subtasks overcome the slight waste caused by their upkeep. The decision about their use should be made case by case, although when information is separated to logical components, it is easier to find and interpret. The guideline emphasizes that the necessary information must be present, logically structured and findable with minor effort.

Issues should always follow a strictly specified hierarchy where the main level issues (requirements and tasks) must exist and the subissues (subtasks and -requirements) inherit from them. Every issue should always have only one parent, but the amount of children is not limited. Figure 2 depicts this hierarchy structure. To create this structure in GitHub issue references, labels and naming conventions are to be used.

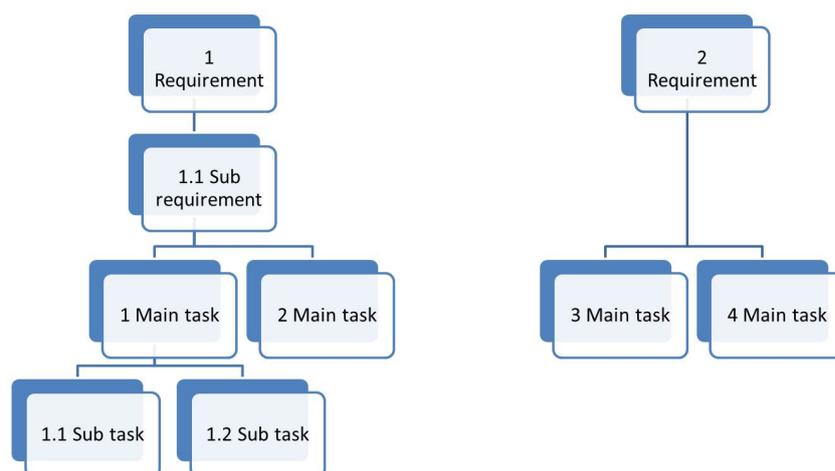


Fig. 2. Different hierarchy combinations for issues.

4.2 The steps of creating an issue

Creating issues should be an activity for the whole team. Requirements and sub-requirements may be composed with only a specified group of team members and customer representatives, but creating tasks should involve the whole team. This has many benefits. Seasoned developers may already have suitable

solutions or have such experience which is useful in some other way. Letting the developers contribute enhances the team spirit and takes them inside the decision making process. In the best scenario, the overview of the domain and its business objectives is broadened within the team, thus building a stronger foundation for making better decisions.

As the guideline is directed towards agile environments, issues can be created iteratively. The best approach is dictated by the size of the project and its application domain. As the requirements evolve, so do the issues: new ones are created, old ones closed, modified or rejected, tasks are started and completed.

The creation process consists of seven steps. **The first step** is deciding the title of an issue. The title should include some kind of a reference prefix that is different for requirements and tasks to help identifying the issue and its place in the issue hierarchy.

The second step involves writing the description of the issue. This is a very crucial step and concerns the question of how long should the description be? If too much information is given, the issue becomes cumbersome and it might be hard to find the core information. Too vague descriptions tend to cause guessing, misleading or a need to consult somebody with better knowledge thus wasting time. If the project team is highly experienced and proficient with the domain of the project, it might be a good idea to leave room for an individual thinking and implementation. This endorses the team and acts towards agilism which expects that experienced developers reach the best outcome without handholding.

The requirements on the other hand should be as unambiguous as possible to make sure that every stakeholder understands them the same way. The bottom line is that the issue should always have a description which is tangible. The guideline suggests that as much of the task specific information as possible should be directly in the description.

A large amount of pictures, documents or other relatively static material should be archived somewhere else to reduce the overhead of information. For requirements it might be a good practice to document them to the wiki. In such case a short description and a link are sufficient for a requirement issue.

The format of the issue's description is not predefined, but the guideline highly recommends that a logical structure is used throughout each issue taking advantage of GitHub Flavored Markdown (GFM). GFM is a special hypertext formatting syntax utilized by GitHub. It inherits from the Markdown [7] and is enhanced with additional convenient features, such as task lists and automated references.

The description shall comprise four parts: summary, information, task list and references. The summary explains the issue in few sentences. The information part consists of all the relevant information. If this data is stored somewhere else, links should be used to point out the source. The task list is utilized when a task involves steps, that are not split to own tasks yet need to be monitored. They should also be used as subtask replacements if the subtasks are not used.

Otherwise the format of the description should be kept as simple as possible. Developers should be aware of GFM's special features that are not included in

the basic Markdown. For example @-notation or “mentioning” can be used to notify a specific person or a team about the issue. Mentioning creates a filter into the issue tracker for the ones mentioned.

Because GitHub itself does not recognize any kind of hierarchical constraints between issues, such a behavior must be implemented manually. The way the guideline achieves this is by using labels, naming conventions for issues and issue references. An existing issue can be referenced from another issue establishing a link between them. In GitHub referencing an issue from another means that within the first issue’s view, there is a hyperlink pointing to the second issue. The link is accompanied by a comment. The issue referenced must exist to get the linking to work. This can cause inconvenience when a main task is created before its subtasks. The guideline recommends that the references are updated when new issues are created. Referencing issues follows the same rules as the hierarchy of them. The exception is that if an issue has a close relation with another issue which is not a descendant or an ancestor to the original issue, a relational reference can be established. Otherwise the reference should always be aimed at the direct parent or child of the issue.

Steps three to five consist of assigning additional information to the issues: first labels, then people and lastly milestones. The labels are an essential part of the guideline and they are the main solution for creating visibility and status tracking. The guideline does not explicitly state what exact labels should be used rather it states possible categories for them. The categories are to be chosen based on the needs of the project and they may vary. There are a total of six categories suggested by the guideline: the type of the issue, the subtype of the task, status, requirement level, priority and miscellaneous. The guideline recommends that every category has its distinctive color with a unique shading for every single label. A short prefix in the beginning of each label depicting the category it belongs to complements the color coding. Generally there should always be a maximum of one label per category assigned to an issue, the miscellaneous labels being an exception but the important thing is that the categories exist and are used appropriately. After the labels comes assigning people (assignees) who are responsible for the implementation of the issue. They are linked directly to the issue.

The use of milestones is recommended and they complement the status and priority categories of the labels. The exact way to use the milestones is up to the development team. The guideline suggests that they can be used to represent iterations or groups of tasks covering some wider topic or feature.

The sixth step is publishing the issue. **The seventh** instructs the creator of the issue to update the references of other issues. When an issue is created and references updated successfully, it starts its own lifecycle.

4.3 Status tracking and traceability of issues

Status tracking and tracing requirements are the core activities of RM. We have already created a base for them in the previous sections.

The traceability is mainly gained by the naming conventions and references, but also the labels from the type and subtype categories enhance it. The status tracking however relies on milestones and labels, like status, priority and miscellaneous.

The monitoring of these aspects requires filters. Even though they can not cover every scenario they are quite powerful, especially since the filters are a part of the URL, so bookmarking the most used filter combinations is possible. The required filter combinations is determined by what kind of information is relevant to different team members. Developers should be aware of all the issues that mention them and the issues directly assigned to them. Especially the priority and milestone information is relevant. Project managers should pay attention to every task currently worked with. This includes following milestone deadlines and task list progression if the issues contain them. Also issues needing special actions - like ones flagged “blocked” - must be dealt with without a delay to keep the workflow going.

4.4 Combining version control and wiki to the issues

The wiki has already been mentioned in the issue creation. Depending on how much information is attached directly to the issue, the wiki is an excellent alternative for lengthier data masses. As it is within the repository, the convenience is increased due to an easy access. Should the wiki be utilized, establishing a logic structure inside it is recommended. The structure itself does not concern the guideline as long as it is consistent and logical. A careful attention should be kept to avoid situations where the wiki’s data contradicts issues’.

The repository offers some interesting interactions like referencing an issue from a commit message is possible. This is special feature of GitHub. Therefore to enhance visibility, developers are encouraged to always reference the issues their commit affects. This creates an automatic comment to the issue’s comment section, making the link between the commit and the issue more visible and traceable.

4.5 Updating and maintaining issues

During the lifecycle of the issue, it is going to be updated in several ways: changing labels, assigning milestones, assigning people, commenting, referencing it from commit messages and closing it. Stakeholder communication regarding issues is a common update action. The challenge is that the communication can occur in multiple media. These interactions can cause changes to the issues’ content. It is imperative that in such cases, the issue is immediately updated to reflect the changes: the descriptions of the issues must always be up to date. This guarantees that the latest and best knowledge is easy to find. It is encouraged to use the comment area of an issue for discussion.

A crucial note is that users should be careful when deleting old information. By doing so the traceability is compromised. However the descriptions should not contain unnecessary information. The old information should be moved from the

issue to a suitable page in the wiki. As important as it is to keep the description updated, is to make sure that the labels are used and updated. The importance of the labels is to visualize issues and different aspects of them in one view. Should the labels be misused or not updated, an unnecessary waste is generated. Letting the information get old causes mistrust towards the guideline and may lead for rejecting its practices. It also interferes with the RM and its objectives.

5 The evaluation of the guideline

The guideline was evaluated in a student project given in the School of Information Sciences at the University of Tampere in 2013. The project lasted nine months, and had four developers and three project managers from whom one had to drop out during the project. None of the team had earlier experience with GitHub's issue tracker, though two of them had used GitHub. The team was multicultural and used English for the communication.

The goal of the project was to produce a mobile puzzle game which introduces the concept of computer science to school applicants. Due to a requirement of an open source development GitHub was chosen as a version control platform. At the beginning of the case study the team was accustomed to the platform. They also received the first version of the guideline. Couple of weeks later the team received a shorter document summarizing the key aspects of the guideline. The team was responsible of deploying the guideline. We observed the usage from three perspectives: watching the team's actions in GitHub, surveying their internal communication and attending the meetings the team had with the customer. In the end of the project an online survey was filled by the team with different sets of questions for developers and managers.

Several important observations were made from the case study. The managers were little lazy to bring the requirements to the issue tracker and convert them to tasks. When this was finally conducted the project had already gone a good way. This meant that some of the traceability and status tracking were inherently lost. As the team consisted of mainly inexperienced students, they put more emphasize on the actual implementation than on the RM. This was also reflected on how the team perceived the whole issue tracker. One of the developers commented "I think the management of tasks and requirements is the role of the managers, – Managers should just fiddle with the requirements." The rush with the deadlines also meant that the team skipped or neglected some parts of the guideline. For example issue descriptions were incomplete, issue references were neglected and the label categories were used too liberally. However the whole team felt that they had followed the guideline and it in fact did help them to achieve a more coherent RM process: "I believe it was very efficient", "Overall, the guideline was useful and logical, I did not find any inherent flaws in it."

The guideline was evolved from the first version published in October 2013 in a slideshow format. The purpose was to iteratively develop the guideline based on the feedback we received from the case study from October 2013 to May

2014. A few things were altered. The structure of the guideline was reformatted to make it more logical and readable. The discussion whether subtasks should be used was widened. We realized that there are valid situations where a main task coupled with a task list is enough. Therefore better arguments for and against of the subtasks were made, highlighting that the relevant information must be easily displayed whatever the chosen approach is. The first version leaned towards the expectation that there is a strict division between who creates and handles issues and the rest of the team. The guideline and RM are everyone's responsibility and everyone's contribution is valuable, so the guideline was changed to put more emphasis on the whole team. The guideline failed to note the powerfulness of the milestones. With added examples users should be able to identify additional purposes for them that could help monitoring issues. Maintaining the issues received a new note about preserving the old information, especially about disregarding it. Tracing requirements becomes hard should the initial situation get lost. A new recommendation was that the issues should be closed with an accompanying comment to clarify why the issue was closed. In the case study there were a lot of issues closed without an apparent reason. This uncertainty was also partially due to somewhat disorganized use of the labels. Therefore the up-to-date labels are even more highlighted. Creating the special issues like enhancement proposals and bug reports has been clarified to distinct how the guideline expects them to be used.

5.1 How to use the guideline

The purpose of the guideline was to offer a set of recommendations and practices for the RM in GitHub's environment. Individual sections from the guideline can be used as such but it is advisable to use it as a whole. Some room is left for customization to preserve the versatility. The guideline aims to complement the four areas of RM: change control, version control, requirements tracing and requirements status tracking [9, 11, 16] by utilizing mainly the issue tracker. As the issue tracker is lightweight to use, it is well suited for the agile RM. As argued before there does not exist as thorough a guideline for the RM in GitHub. There are blog posts covering the issue tracker but they are not systematic nor scientific, but rather experience reports. The lean software principles are used for assessing the guideline and its compatibility to the agile environment. These principles are abstract enough so that they don't restrict guideline's usefulness, yet still offer enough concreteness for assessment.

The guideline is a collection of practices and recommendations working together towards a consistent RM process. The guideline requires a knowledge of using GitHub, especially the repository, issue tracker and wiki. Using GFM is advised in the issues and wiki.

Before starting to work with issues, some preparations must be made. The labels, their categories and their color coding are to be decided. Overall naming conventions with the issues must be agreed on. Generally it is ideal to go through the guideline's practices and decide how they are applied into the project. Without a commitment from the whole team, the guideline loses its effectiveness.

The team should appoint somebody responsible for the RM. This does not exempt the team from the RM. The purpose of the appointment is to have a person who can attend questions regarding the issue tracker, guideline and issues.

The guideline does not take a stand when it comes to requirements engineering processes occurring before the RM. They can be conducted by whatever means necessary for the project's scope and type. As new requirements are identified they are gradually created into the issue tracker. The preferred way is that the requirements are immediately put into the issue tracker. Postponing this delays the implementation. As soon as the requirements are in the issue tracker the team can start splitting them into tasks. When the first tasks are ready the implementation can begin. The requirements are to be split in the order of priority to generate the best business value.

Creating requirements and tasks is usually an ongoing process and happens throughout the whole project. When the first tasks are under work, they must be monitored, maintained and updated.

5.2 Evaluating the guideline against the RM & lean principles

The following list will summarize the key aspects enhancing the objectives of the RM and the principles of the Lean as introduced by Poppendiecks' [12]. The first four cover the principle's of the RM, the rest focuses on the Lean.

Change control. Working and successful change control requires that there is a way to get an overview of how different requirements and their implementations and components relate to each other. This greatly helps solving the impacts of the proposed change. A special issue type the enhancement proposal gives a convenient way for formally proposing changes and tracking their life span: does the proposed change get implemented, is it further evaluated or even completely rejected. Of course, in an agile environment, changes don't always follow a formal path. The guideline enforces that information contained in the issues is always up to date.

Version control. The issue tracker itself holds all the requirements together. Every issue is given an unambiguous identification number by GitHub that can be used for referencing the issue. Versioning itself is not directly supported so the guideline suggests that the old information is not erased, but preserved somewhere else. The naming conventions and type labels are there for identifying purposes.

Requirements tracing. Separating requirements and concrete tasks and establishing a hierarchy between each issue creates a clear structure for tracing requirements and links they have. This strongly requires that the references in the descriptions are used and maintained properly. The aforementioned naming conventions also help identifying the relations between the issues.

Status tracking. The labels are the best way to accomplish the status tracking. The suggested categories are crafted so that tracking requirements and their tasks is as straightforward as possible. The drawback is that the labels

are mainly toggled with tasks, not with requirements so the status tracking of requirements must be carried out through tasks.

Eliminating waste. Descriptions with enough information and an encouragement for a free speech in comments aim to generate more knowledge for the whole team. This contributes towards learning how to produce value more efficiently. Avoiding unnecessary waiting sharing knowledge and using labels are suggested. The processes of the guideline are narrowed down and are inherently simple. They are also flexible, which makes them more suitable for various situations. Splitting requirements to small pieces increases the possibility that unrequired extra features are detected and rejected.

Optimizing the whole. This is more like an attitude of the team and cannot be created by this kind of guideline. Sharing knowledge and discussing it increases the insight to the customer's mind and thus makes it easier to comprehend the whole picture.

Building quality in. The customer's needs must be understood and openly collaborated and communicated. Information sharing, keeping everything visible and actively using commenting are ways the guideline proposes.

Learning constantly. Sharing and absorbing both information and knowledge is best way to fulfill this principle. The splitting of requirements in small pieces forces the team to truly get a better understanding of the domain.

Delivering fast & deferring commitment. Small tasks take less time to implement making the fast delivery possible. Feedback and communication should occur all the time further improving the delivery and its quality. Deferring commitment requires the best available information. Keeping issues up to date and openly discussing them enhances this.

Respecting people. The guideline aims for empowering the whole team. They are the ones to use the issue tracker, and they all should have an interest on what is happening there. The guideline does not restrict who can do what. How well the empowerment succeeds is based on whether the parent organization supports giving the decision power to the development teams.

6 Conclusions

The aim of this article was to present a semi-formal guideline for conducting the RM process using only the tools provided by GitHub. To prove that the guideline is applicable and working, a case study was conducted. The guideline was evaluated on how well it supports the RM objectives and the lean principles.

As far as the research shows, this is a unique topic: similar guidelines for GitHub platform with a properly surveyed case study could not be identified. There exist several blog posts about thoughts and suggestions of using the issue tracker, but none of them approached the topic as systematically. These blog posts also lacked the evaluation of their usefulness [1]. Our conclusions are based on the case study and observations, not just vague opinions.

For the science, the guideline is interesting because it combines the lean principles into the RM. This is also an area that is covered by very few studies.

The evaluation of the guideline gives more insight on how the RM can be coupled with the lean principles.

For the practitioners the guideline provides considerable, systematic and well documented instructions for using the issue tracker to handle the RM in a project that uses GitHub. The guideline is also evaluated by the case study, and though the study consisted of only a student project, its feedback was very positive. The case study project group felt that the guideline helped them achieve a more consistent RM process. By comparing the guideline against the lean principles we have showed that the guideline is well suited for the agile approach.

The guideline was created to be usable in a wide array of projects, though it is required that the project utilizes some agile approach. Therefore the biggest factor for not using the guideline is the chosen development process of a project. The practices and suggestions of the guideline are designed to be easily understood and adopted for establishing a good base for the RM. This is especially beneficial for agile projects that tend to bury the RM below an iterative development. As GitHub could be considered a programmer-friendly platform, handling the RM in the same place can lower the threshold of developers to participate and take more active role in it.

Like with any agile practices, the users are left with the responsibility to follow the instructions as they see fit. This can cause a problem, if users try to cherry pick the concepts and leave out others. A selective use can greatly diminish the usability and results achieved by the guideline. The guideline leaves room for customization to preserve versatility but this means that instruction contain some vagueness.

There are some limitations concerning this article. The topic is very specific which makes it unique but at the same time difficult to generalize. The RM has certain objectives and accomplishing them depends on the tools at disposal. With GitHub we have a limited set of features but on some other platform the tools and their usage may be completely different. Although the evaluation combines the RM and lean principles, it wraps around the guideline's practices and tools offered by GitHub.

The case study was relatively limited in scope. The project was a student project that lasted nine months. The problem with students is that they tend to lack the experience present in professional software projects. As the project was carried out during the university semester the students also had other course to attend to, leaving relatively small amount of time to work with the project. The team was multicultural and this posed a minor problem with communication: English was used as the communication language but it was not native to any of the project members.

The guideline would greatly benefit from a larger case study involving projects with different settings and sizes. A wider case study would make it easier to further evaluate the core ideas of the guideline and how well they achieve their objectives. It could also highlight the situational contexts the guideline manages the best. Projects with experienced team members could give better facts to support different kind of approaches.

Overall the guideline achieved its goal: creating instructions for an agile requirements management process utilizing only GitHub's own tools. This is backed up by the case study and critical assertion of guideline's practices and GitHub's features.

References

1. Bicking, I. (2014). How We Use GitHub Issues To Organize a Project, <http://www.ianbicking.org/blog/2014/03/use-github-issues-to-organize-a-project.html>
2. Cao, L., & Ramesh, B. (2008). Agile Requirements Engineering Practices: An Empirical Study. *IEEE Software*, 25(1), 60–67.
3. Cockburn, A. (2006). *Agile Software Development: The Cooperative Game* (2nd edition.). Upper Saddle River, NJ: Addison-Wesley Professional.
4. Ernst, N. A., Murphy, G. C.: Case studies in just-in-time requirements analysis. *IEEE Second International Workshop on Empirical Requirements Engineering*, 25–32 (2012)
5. Favaro, J. (2002). Managing requirements for business value. *IEEE Software*, 19(2), 15–17.
6. GitHub - Web-based Git repository hosting service, <http://www.github.com>
7. Gruber, J. (2004). Markdown, <http://daringfireball.net/projects/markdown/>
8. Hofmann, H. F., & Lehner, F. (2001). Requirements engineering as a success factor in software projects. *IEEE Software*, 18(4), 58–66.
9. Li, J., Zhang, H., Zhu, L., Jeffery, R., Wang, Q., & Li, M. (2012). Preliminary results of a systematic review on requirements evolution. In *16th International Conference on Evaluation Assessment in Software Engineering (EASE 2012)* (pp. 12–21).
10. Miller, G. G. (2001). The Characteristics of Agile Software Processes. In *Proceedings of the 39th International Conference and Exhibition on Technology of Object-Oriented Languages and Systems (TOOLS39)* (p. 385).
11. Paetsch, F., Eberlein, A., & Maurer, F. (2003). Requirements engineering and agile software development. In *Twelfth IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises, 2003. WET ICE 2003. Proceedings* (pp. 308–313).
12. Poppendieck, M., & Poppendieck, T. (2003). *Lean Software Development: An Agile Toolkit*. Boston, Mass.: Addison-Wesley Professional.
13. Salo, R. (2014). A guideline for requirements management in GitHub with lean approach, University of Tampere, School of Information Sciences, master's thesis, 71 pages.
14. Sommerville, I. (2007). *Software engineering*. Harlow, England; New York: Addison-Wesley.
15. Waldmann, B. (2011). There's never enough time: Doing requirements under resource constraints, and what requirements engineering can learn from agile development. In *Requirements Engineering Conference (RE), 2011 19th IEEE International* (pp. 301–305).
16. Wiegers, K. E. (2009). *Software Requirements*. Microsoft Press.
17. Zhang, Z., Arvela, M., Berki, E., Muhonen, M., Nummenmaa, J., & Poranen, T. (2010). Towards Lightweight Requirements Documentation. *Journal of Software Engineering and Applications*, 03(09), 882–889.

Priority Queue Classes with Priority Update

Matti Rintala¹ and Antti Valmari²

Department of Pervasive Systems¹, Department of Mathematics²
Tampere University of Technology
Tampere, Finland
{Matti.Rintala,Antti.Valmari}@tut.fi

Abstract. A limitation in the design of the interface of C++ standard priority queues is addressed. The use of priority queues in Dijkstra's shortest path search algorithm is used as an example. Priority queues are often implemented using heaps. There is a problem, as it may be necessary to change the priority of an element while it is in the queue, but finding the element from within a heap is costly. The problem may be solved by keeping track, in a variable that is outside the heap, of the position of the element in the heap. Unfortunately, this is impossible with the template class interface used by the C++ standard library priority queue. In this research, the problem is analysed in detail. Four interface designs and corresponding implementations are suggested. They are compared experimentally to each other and the C++ design.

Keywords: data structure, interface, C++ standard library, priority queue, priority update

1 Introduction

Some modern programming languages such as C++ contain high-quality implementations of fundamental data structures in the form of *containers*. This has obviously reduced the need for programmers to implement fundamental data structures on their own.

Unfortunately, the shift from self-made data structures to ready-made containers prevents the exploitation of the data structures to their full potential. This is because the interfaces of containers reflect some idea of how the data structures will be used, making some other uses difficult or impossible. An obvious example is the dynamic order statistics data structure (DOSDS) [5, Ch. 14.1], which consists of a red-black tree with an additional *size* attribute in each node. Although the C++ standard map is typically implemented as a red-black tree [8, p. 315] and it seems easy to extend it to the DOSDS at the source code level, it seems impossible to implement the DOSDS only using the features offered by the interface of the C++ standard map.

This study focuses on the priority queue. The C++ standard priority queue is implemented as a heap. It has operations for pushing an element to the queue, and finding and removing an element with the highest priority from the queue.

It does not have an operation for changing the priority of an element that is in the queue, although the heap facilitates an efficient implementation of it. The problem is that the efficient implementation requires such co-operation between the queue and its user that is difficult to support in an interface. To solve the problem, we introduce four designs and test them experimentally.

As a use case of the priority queue, Dijkstra's shortest path algorithm is used. Because it is simple and widely known, it serves well for illustrating the problem and the interface designs. We emphasize that the goal of this study is not to make Dijkstra's algorithm as fast as possible (for that purpose, please see [4]). The purpose of the measurements in this study is to demonstrate that our alternative priority queue interface designs are not unrealistically slow.

There are C++ libraries that provide priority queues which are not based on binary heaps, and some of these also allow changing the priorities of queue elements. One such library is Boost.Heap [2], which contains priority queues based on d -ary, binomial, Fibonacci, pairing, and skew heaps. Comparing the performance of these queues to the ones presented in this study is a potential subject for further study. In this study, "heap" always refers to a binary heap, unless stated otherwise.

The use of priority queues in Dijkstra's algorithm is discussed in Section 2. Section 3 presents the alternative designs, and the experiments are reported in Section 4. Section 5 concludes the presentation.

2 Dijkstra's algorithm

Dijkstra's algorithm finds shortest paths in a directed multigraph whose edge weights are non-negative. It uses one vertex as the start vertex. It finds shortest paths from the start vertex to each vertex in increasing order of the length of the path. It may be terminated when a designated target vertex is found, or it may be continued until all vertices that are reachable from the start vertex have been found. We use the latter termination criterion.

Two versions of Dijkstra's algorithm are shown in Figure 1 in pseudocode. Each vertex v has an attribute `edges` that lists the outgoing edges of v ; `dist` that contains the shortest distance so far known from *start* to v ; and `prev` that tells the previous vertex in a shortest path so far known. Initially each `dist` attribute contains a special value ∞ that indicates that the vertex has not yet been reached. The `edges` attributes represent the multigraph and thus, together with *start*, contain the input to the algorithm. The initial values of `prev` do not matter.

The algorithm on the left in Figure 1 uses a priority queue Q that stores pointers, indices, iterators, or other kind of handles to vertices. For simplicity, we will talk about pushing, popping, and the presence of vertices in Q , although in reality handles to vertices are in question. The operation `push(v)` pushes v (that is, a handle to v) to Q , and `is_empty` returns a truth value with the obvious meaning. The operation `pop` returns a vertex whose `dist` value is the smallest among the vertices in Q . It also removes the vertex from Q . The operation

```

start.prev := nil; start.dist := 0
Q.push(start)
while ¬Q.is_empty do
  u := Q.pop
  for e ∈ u.edges do
    v := e.head; d := u.dist + e.length
    if d < v.dist then
      v.prev := u
      if v.dist = ∞ then
        v.dist := d; Q.push(v)
      else
        v.dist := d; Q.decrease(v)

```

```

start.prev := nil; start.dist := 0
Q.push(start, 0)
while ¬Q.is_empty do
  (u, d) := Q.pop
  if u.dist = d then
    for e ∈ u.edges do
      v := e.head; d := u.dist + e.length
      if d < v.dist then
        v.prev := u; v.dist := d
        Q.push(v, d)

```

Fig. 1. Dijkstra's algorithm assuming that the priority queue decrease operation is (left) and is not (right) available

`decrease(v)` informs Q that the `dist` value of v has become smaller. This causes Q to re-organize its internal data structure.

The queue does not store (copies of) the `dist` values. The queue operations get the `dist` values from the vertex data structure. This means either that the code for Q depends on the vertex data structure or that the interface of Q contains features via which Q can access the vertex data that it needs. This issue is discussed extensively in the next sections.

Let us say that the algorithm *processes* a vertex when it is the u of the **for**-loop. The algorithm processes the reachable vertices in the order of their true shortest distance from $start$. It goes through the outgoing edges of u , to see whether a shortest path to u followed by the edge would yield a shorter path to the head state v of the edge than the shortest so far known path to v . If yes, the path and distance information on v is updated accordingly. Furthermore, v is either pushed to Q or its entry in Q is updated, depending on whether v is in Q already.

An example is shown in Figure 2. In it, the outgoing edges of vertices are investigated from top to bottom. Vertex A is used as $start$. Initially, the algorithm marks A found with distance 0 and no predecessor vertex, and pushes it to Q . Then it pops A from Q and processes it. It finds B with distance $0 + 1 = 1$ and predecessor A , and pushes it to Q . Then it does the same to E with distance $0 + 6 = 6$, and then to C with distance $0 + 2 = 2$.

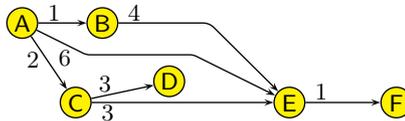


Fig. 2. An example graph for Dijkstra's algorithm

Next it picks B, because its distance 1 is the smallest among those currently in Q . It finds E with distance $1 + 4 = 5$ and predecessor B. It updates the entry for E in Q because of the distance decreasing from 6 to 5. Next the algorithm processes C, finding D with distance 5 which it pushes to Q . Then it finds E anew, but does nothing to it, because the distance does not improve. Now Q contains D and E. Both have distance 5. The algorithm processes them one after the other in an order that depends on implementation details. When processing E, it finds F. The predecessor of F is E, its predecessor is B, and then A, listing a shortest path from A to F backwards.

The C++ standard priority queue is based on a heap. A heap is an array A such that for $1 \leq i < k$, the priority of $A[\lfloor \frac{i-1}{2} \rfloor]$ is at least as high as the priority of $A[i]$, where k is the number of elements in the array. Thus $A[0]$ has the highest priority. We say that $A[\lfloor \frac{i-1}{2} \rfloor]$ is the *parent* of $A[i]$ and $A[i]$ is a *child* of $A[\lfloor \frac{i-1}{2} \rfloor]$. An element has typically and at most two children and one parent. A new element is added by assigning it to $A[k]$, incrementing k , and then swapping it with its parent, the parent of the parent, and so on until it is in the right place. The `pop` operation returns $A[0]$, moves $A[k-1]$ to its place, decrements k , and then swaps the new $A[0]$ with its higher-priority child until it is in the right place. These require $O(\log k)$ operations.

The fact that the priority queue operations obtain the `dist` values from the vertex data structure does not prevent the use of the C++ standard priority queue as Q . The `dist` values are only needed when comparing queue elements to each other. The C++ standard priority queue can be given a special comparison operator that fetches the values that it compares from the vertex data structure. Alternatively, one can define a special type for the queue elements such that its data consists of the handle, and its comparison operator works as described above. These are common practice.

On the other hand, the C++ standard heap and priority queue do not have any `decrease(v)` operation. Furthermore, `decrease(v)` cannot be implemented efficiently using the features that they offer (except with exorbitant trickery, see Section 3.5). The main problem is that there is no fast way of finding the handle to v from within A . The handle is not necessarily where `push(v)` left it, because later `push` and `pop` operations may have moved it. As a consequence, in practice, the C++ standard priority queue cannot be used in the algorithm in Figure 1 left.

This problem can be solved by using a self-made heap and by adding an attribute called `loc` to each vertex. The value $v.\text{loc}$ tells the location of the handle to v in A , that is, $A[v.\text{loc}]$ contains a handle to v . When swapping handles in the heap, the heap operations also update the `loc` values of the vertices to which the handles lead.

Let n denote the number of vertices and m the number of edges in the multigraph. The running time of this implementation is $O(m \log n)$. If also the initialization of the `dist` attributes to ∞ is counted, then the running time is $O(n + m \log n)$. This formula is slightly different from the corresponding formula in [5, Ch. 24.3], because there all vertices are put initially to the queue, while only

reachable vertices ever enter Q in Figure 1. The `edges`, `head`, `length`, `prev`, and `dist` attributes contain the input data and the answer, so they are considered part of the problem and can be left out when comparing the memory consumption of alternative solutions. In addition to them, this solution uses n `loc`-attributes and at most n handles in the queue, which is $\Theta(n)$ bytes of memory. Assuming that each elementary data item consumes the same amount of memory, the increase in memory consumption is at most 40%. This occurs if there is an edge from `start` to each other vertex, no other edges, and no other information in vertices and edges than mentioned above. If $m \gg n$, the increase is approximately 0%.

With the C++ standard priority queue, Dijkstra's algorithm can be implemented as is shown in Figure 1 right. The elements of Q now consist of two components: a handle to the vertex and the `dist` value of the vertex at the time when it was pushed to Q . A vertex is pushed to Q each time it is found with a shorter distance than before. Therefore, it may be many times in Q . However, each instance has a different distance. The test `u.dist = d` recognizes the first time when u is popped from Q . It prevents the processing of the same vertex more than once.

In the case of Figure 2, the algorithm pushes (A, 0); pops (A, 0); pushes (B, 1), (E, 6) and (C, 2); pops (B, 1); pushes (E, 5); pops (C, 2); and pushes (D, 5). It does not push (E, 5) again, because 5 is not smaller than the `dist` value that E already had. At this point Q contains (E, 6), (E, 5), and (D, 5). The value of `E.dist` is 5. Next the algorithm pops (D, 5) and (E, 5) in this or the opposite order, processing D and E. Then it pops (E, 6). This time it does not process E, because the popped value 6 is different from `E.dist` which is 5.

With this implementation, the running time is $O(m \log m)$. This is asymptotically worse than $O(m \log n)$. (With a multigraph, it is not necessarily the case that $m \leq n^2$.) However, with such applications as road maps the difference is insignificant. This is because very few roadcrosses have more than 6 outgoing roads. So $m \leq 6n$, and the number of swappings of heap elements per each push or pop (or decrease) operation is at most approximately $\log_2 6n = \log_2 6 + \log_2 n \approx 2.6 + \log_2 n$, while the corresponding number with Figure 1 left is at most approximately $\log_2 n$. At the level of constant factors, Figure 1 right wins in simpler access of the distances but loses in two data items being exchanged in each swapping instead of one. As a consequence, which one is faster in practice is likely to depend on implementation details and the properties of the input multigraphs.

The additional memory consumption consists now of at most m handles and at most m distances in the queue, which is $\Omega(1)$ and $O(m)$ bytes of memory. If $m \gg n$, this may almost double the memory consumption.

3 Priority queue implementations discussed in this study

Based on the need for a priority queue with updateable priorities, several solutions were designed by the authors, each providing an increased level of encapsulation, modularity, and genericity. These solutions were implemented and

performance tested against each other. Figure 3 shows the structures of the five solutions used in this study (of which the first lacks updateable priorities). Sections 3.1–3.5 discuss these in detail.

3.1 Using STL priority queue with duplicated queue elements

Figure 3(a) shows the structure of a priority queue using STL priority queue. In this approach the lack of priority updates is circumvented by adding the same vertex into the queue several times, if necessary (as described earlier). Each queue element consists of a handle (a pointer, for example) to the graph vertex, as well as the shortest distance to the vertex at the time it was pushed to the queue. The priority queue uses this distance as the priority of the element. In addition to other graph data, each graph vertex contains the calculated shortest distance (originally initialized to infinity or other special value).

Data that is only needed for priority changes is marked in red. In this version that data consists of the queue’s internal priority field. In the 64-bit test setup this caused the size of graph vertices to be 24 bytes. The size of the distance field was 4 bytes, the remaining 16 bytes were used for pointers needed by the graph itself and the resulting shortest path, with 4 bytes added by the compiler for memory alignment purposes [1, Ch. 6]. The size of queue elements is 16 bytes (4 bytes for the internal priority field, 8 bytes for the vertex pointer, and additional 4 bytes again for memory alignment).

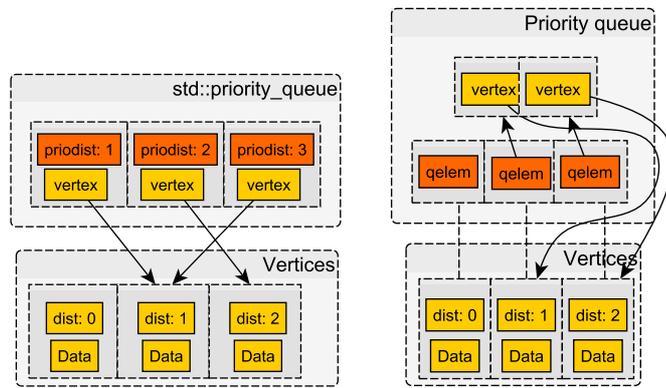
3.2 Allowing priority updates in a heap using additional location data

Figure 3(b) depicts a solution using a self-made priority queue based on a heap and a separate array for tracking the location of graph vertices in the heap (the location information is drawn as an arrow in the figure, but it can be implemented as an integer index). The additional location array has a corresponding element for each graph vertex in the graph array. This implementation requires that graph vertices are stored in an array so that indices can be used.

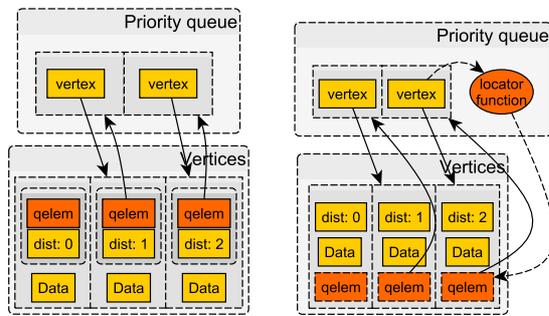
Each time a graph vertex is added to the heap, its location index in the heap is stored in the location array. Similarly each time an element is moved in the heap, the heap algorithms update the location index in the array. Moving an element may be caused by adding or removing an element or changing its priority. When the priority of a graph vertex has to be changed, its location in the heap can be found from the location array, so that only appropriate elements of the heap have to be updated.

Handling of the location tracking is encapsulated inside the priority queue, which is an improvement to the previous version, where the user of the queue had to take care of duplicate queue elements and the extra distance fields.

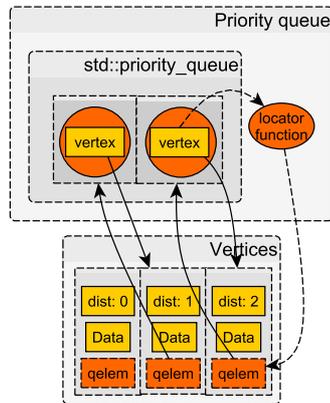
In this implementation the only extra data needed for priority changes consists of the location array. In the test setup this adds extra 4 bytes for each graph vertex. The size of the graph vertex itself was again 24 bytes, including 4 bytes for memory alignment. The size of the queue elements in the heap was 8 bytes.



(a) `std::priority_queue` and duplicates (b) separate index array



(c) inheritance-based (d) locator function -based



(e) `std::priority_queue` and locator -based

Fig. 3. Priority queue variations used in this study

3.3 Using inheritance to store location information in the graph vertices

The previous algorithm used an internal array for tracking the location of queue elements. This forces the actual graph vertices to be stored in an indexable data structure as well (so that the index of the vertex in the queue can be used to find the location in the location array). It also means that the number of graph vertices has to be known beforehand, so that a location array of suitable size can be allocated in the priority queue.

This situation can be improved by moving the location index inside the graph vertices themselves. However, to preserve modularity it should still be the responsibility of the priority queue to take care of this information. One solution is shown in Figure 3(c). The priority queue provides a base class from which graph vertices should be derived (using object-oriented inheritance). The base class contains both the distance field used as a priority and the location index in the heap. This way the priority queue does not have to know the exact type of the graph vertices, but it can access the base class data to provide a priority queue with updateable priorities. Similarly, the implementor of the graph vertex does not have to know the internal implementation of the queue, just inherit the vertex from the given base class. The priority of a graph vertex is set and changed using the interface of the priority queue, and the value of the priority (i.e., distance) can be queried directly from the base class.

The size of the base class became 8 bytes in the test setup (4 bytes for the distance, 4 bytes for location index), making the size of the graph vertex again 24 bytes. Queue elements were 8 bytes each.

3.4 Adding modularity: externalizing priority and location data

The inheritance-based implementation abstracts the priority and location index into a separate base class, allowing vertex data structures to be created without having to pay attention to data needed by the priority queue. This adds modularity and helps in separation of concerns.

However, even the inheritance-based implementation requires that the programmer implementing the vertex data structure is aware of the priority queue and its special needs. The vertices have to be inherited from the base class provided by the priority queue, and this means that the choice to use the priority queue described in this study has to be made when the vertex data structure is defined. This rules out using the modifiable priority queue with third-party data structures, whose definition cannot be changed. The fact that priority changes have to be made using the base class's interface also forces the code that changes the priority to be aware of the chosen priority queue algorithm.

In order to solve those problems, a fourth implementation of the priority queue is shown in Figure 3(d). It uses additional functions to calculate the location of the priority data and storage needed for the location index. When a priority queue is created, it is given three parameters: the type of data used as queue elements and two functions, one to compare the priorities of two vertices,

and one to calculate the location of the vertex's location index. The priority queue implementation uses these functions each time it has to compare priorities or access the location index.

This approach removes the dependency between the graph vertex data type and the priority queue algorithm. Any data type can be used as a graph vertex as long as it is possible to calculate the priority and the location of the location index based on a handle to the vertex. This makes it possible to use a third-party data type as a graph vertex, and code the name and the location of the priority field into the priority function. Similarly the storage needed for the location index can be embedded inside the graph vertex or in an external data structure.

In fact, this approach is general enough to be used to produce implementations described in Sections 3.2 and 3.3. The first one can be achieved by providing a location index function that calculates the index of a graph vertex and uses that to access an external location index array. Similarly for the inheritance-based approach, the location index function would return a handle to the location index stored inside the base class of the queue element.

The downside of this approach is that using external functions causes additional performance overhead everywhere where the priority or location index is used. Also, using this approach is a little bit more complicated for programmers, since they have to write two small extra functions in addition to the queue element data type.

In this approach, the size of the graph vertex is 24 bytes (4 bytes are again needed for the priority, 4 bytes for the location index). Queue elements are 8 bytes long.

3.5 Test of reuse: Using STL priority queue to create a modifiable queue

The standard C++ already provides a priority queue (`std::priority_queue`). Therefore, it is logical to find out if that queue could be used for creating a priority queue capable of modifying priorities of its elements. As mentioned earlier, the semantics of the standard C++ priority queue assumes that the priority of each element stays constant after the element has been added to the queue.

The benefits of this approach are that the standard C++ priority queue is likely to have been implemented quite optimally (considering the compiler), and this way further development of the standard priority queue benefits the modifiable queue also. Since this implementation is based on the standard C++ priority queue just like the one using duplicate elements, this version can also be used to get an estimate of how much updating the position of each element in the queue costs compared to keeping duplicate queue elements.

The C++ standard guarantees that its priority queue is on a heap, and uses library heap algorithms `std::make_heap`, `std::push_heap`, and `std::pop_heap` to maintain the heap [8, Ch. 12.3, Ch. 11.9.4]. The standard priority queue allows the programmer to specify the underlying data structure for the heap, but by default a vector `std::vector` is used. To make element priorities modifiable,

there have to be mechanisms to check whether an element already exists in the queue, find the element to be updated in the heap, and to update the heap structure based on the new priority.

The first obstacle in this approach is that standard C++ heap algorithms only allow removal of the top element (`pop_heap`) and addition of a new element (`push_heap`). No standard algorithm for updating the position of an existing heap element exists. Fortunately, the implementation provided by the GCC compiler has such operations internally, called `__push_heap` and `__adjust_heap`. The former moves an element up in the heap to its correct position, and the latter moves an element down, if necessary. Using these GCC-specific functions it is possible to readjust the heap after the priority of an element has been changed.

Figure 3(e) shows the structure of this approach. Just like the previous one, this approach is based on a locator function which calculates the location of the location index based on a handle to a graph vertex (making it possible to either embed the location index inside the vertex itself, use a separate location vector, or something else). However, updating the location index when the heap is modified is more difficult. The standard C++ heap algorithms used by the C++ priority queue do not provide means for knowing when the location of a heap element is changed.

In order to keep track of element locations in the heap, this approach uses an auxiliary `HeapElement` class, and a C++ priority queue is created to store these `HeapElement`s instead of regular data elements. Each `HeapElement` object contains a handle to a graph vertex. In addition to performing the assignment itself, the assignment operators of `HeapElement` update the location of the element using the locator function. This way, when a `HeapElement` object is moved inside the heap by C++ heap algorithms, the `HeapElement` assignment operators update the location index of each element. To be specific, C++11 now provides two kinds of assignment, *copy assignment* and *move assignment* [7, Ch. 12.8]. Copy assignment copies data and keeps the original intact, whereas move assignment may transfer parts of the original data to its target and reset the original data to an empty state. All C++11 library algorithms prefer move assignment, if available. The `HeapElement` class only provides move assignment, making sure that multiple copies of the same element cannot be temporarily created during heap algorithms.

After this, implementation of a `std::priority_queue`-based modifiable priority queue is quite straightforward. Standard priority queue operations can be used to add and remove elements from the queue. New operations are provided for modifying priorities of existing elements. In order to change the priority of an element, first the priority data inside the element is modified by the programmer. Then the programmer informs the queue about the priority update, and the queue updates the heap. This of course introduces the risk that the programmer updates the priority but forgets to inform the queue, causing the invariants of the heap to be violated with unpredictable results.

The sizes of the graph vertices and queue elements are the same as in the previous version, 24 bytes and 8 bytes, respectively.

4 Testing the algorithms

In order to test the algorithms, they were used to implement Dijkstra's shortest path algorithm and run through several graphs. This section presents the test results and discussion.

4.1 The test graphs

Figure 4 shows the graph types used for testing the algorithms.

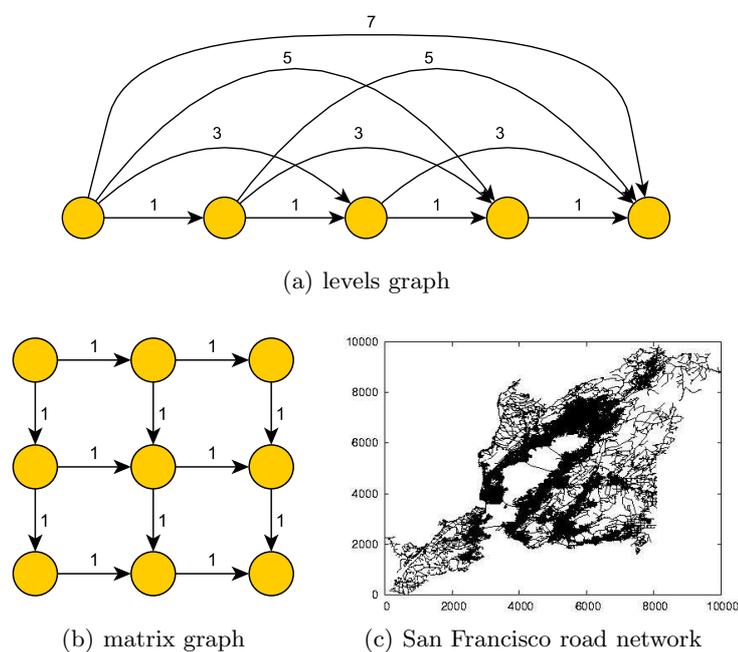


Fig. 4. Graphs used for testing

Graph (a) in Figure 4 depicts a family of graphs that was designed to be increasingly beneficial to the mutable priority queue. In the simplest case (level 1), the graph is simply a series of vertices connected to each other with an edge of length 1 (edges marked “1” in the figure). In level 2 new edges are added. They skip over one vertex and have length 3 (edges marked “3”). Similarly, a level k graph contains for all $i = 1, \dots, k$ edges of length $2i - 1$ that leave from every possible vertex skipping $i - 1$ vertices.

In Dijkstra's algorithm, this type of graph causes the priority of most vertices to be changed $k - 1$ times for a level k graph. For example, when $k = 2$, each vertex (other than the first two) is first pushed to the priority queue with distance

$d + 3$ (where d is the shortest distance to the previous vertex). Then in the next step a shorter path of length $d + 1 + 1$ is found and the priority must be changed. Similarly, for $k = 3$, most vertices get distances $d + 5$, $d + 1 + 3$, and $d + 1 + 1 + 1$, respectively. Thus, the higher the level, the more the basic STL priority queue-based algorithm has to pay for extraneous elements in the queue. However, the total size of the priority queue stays small (at most k vertices at any time) and does not depend on the total number of vertices.

Graph (b) is designed to work in the opposite direction. The graph is a matrix where each vertex (other than the rightmost and bottommost) has edges of length 1 to its neighbour vertices on the right and below. This means that during Dijkstra's algorithm, the shortest distance to a vertex never has to be updated. On the other hand, the size of the priority queue can grow to k for an $k \times k$ graph. This type of graph forces the mutable priority queue algorithms to pay for updating the location index in the queue, but without benefiting from priority changes.

Finally, graph (c) represents a "real world" example without any intentional bias towards any algorithm. It is the road network of San Francisco with 174,956 vertices and 446,002 edges (each road in the map data was assumed to be two-way). The map data and graph image were obtained from [6].

4.2 The test setup

All tests were run under 64-bit OpenSuse 13.2 Linux with Intel Core i5-3570K processor. GCC 5.1.0 compiler was used with `-O3` optimization. Each test was compiled as one compilation unit so that the compiler was able to fully optimize the code.

For increasing the accuracy of the time measurements the operating system was in single-user mode with networking disabled. In each test, the test graph was first created and a dry-run of Dijkstra's algorithm was run to fill CPU caches, etc. Then calculated distances in the graph were zeroed out, and a high-precision time-stamp was acquired from the operating system (using `clock_gettime()`) and a raw hardware-based system clock with a claimed resolution of 1 ns). After this, Dijkstra's algorithm was run again, and a new timestamp was acquired at the end of it. The difference between these two timestamps was recorded as the duration of the algorithm. Then distances in the graph were zeroed out again as preparation for repeating the test.

Hardware interrupts and operating system services can slow down the execution of the program, even in the single-user mode. To eliminate extra overhead, each test run of the algorithm was executed at least 10 times or the test was repeated for 30 seconds, whichever took longer. For tests with small graphs, this meant running the tests for millions of times, whereas with large graphs even the 10 test runs could take several minutes to run. The algorithms themselves are fully deterministic, so any fluctuation in the results is caused by outside overhead, which can only increase the measured time durations. Because of this, only the *minimum* running time for each test was chosen, because it has to contain

the least overhead [3]. In practice, the average and minimum running times only differed on average 1.8 % in the tests.

4.3 Test results

For the levels graphs, tests were run for graph sizes of 10^m , for $m = 1, \dots, 8$ and for levels $k = 1, \dots, 5$. The graph size had no measured effect on the relative results, which is logical since the length of the priority queue does not depend on the graph size in this test. Therefore, only the results for the largest graph of size 100,000,000 vertices are shown. Figure 5 contains the results of the tests. For each algorithm, the table shows both the absolute time for running Dijkstra's algorithm, as well as the difference percentage compared to the basic algorithm using STL priority queue and duplicate queue elements.

k	(a) <code>std::p-q</code>	(b) sep. index	(c) inheritance	(d) locator	(e) <code>std::p-q&loc</code>					
1	0.895s	0.00%	0.734s	-18.1%	0.827s	-7.60%	0.827s	-7.60%	0.693s	-22.6%
2	3.23s	0.00%	1.05s	-67.4%	1.40s	-56.7%	1.38s	-57.2%	1.52s	-53.1%
3	5.64s	0.00%	1.54s	-72.6%	2.01s	-64.3%	2.08s	-63.2%	2.13s	-62.3%
4	8.85s	0.00%	2.22s	-74.9%	2.51s	-71.7%	2.56s	-71.1%	2.51s	-71.6%
5	12.4s	0.00%	2.80s	-77.3%	3.06s	-75.2%	3.20s	-74.1%	3.02s	-75.6%

Fig. 5. Results of performance tests with levels graphs of size 100,000,000

The results show that mutable priority queue algorithms presented in this study are faster than the basic STL priority queue algorithm for all test graphs, and the speed difference grows for graphs with more levels. This is as expected, since the priority of a vertex in the queue is changed an increasing number of times when new levels are added to the graph, forcing STL priority queue to pay for duplicate elements in the queue.

It is interesting to notice that mutable queues are faster than the basic STL priority queue also for level 1, where the graph is a simple string of vertices and no priority changes occur. This also holds for the case where a mutable queue is implemented using STL's own priority queue, ruling out differences in the queue's internal implementation. A possible explanation for this difference is that the basic STL-based approach requires an extra distance field in the queue elements, making the queue elements larger.

Figure 6 shows the test results for matrix graphs. Tests were run for graphs of size 10×10 , 100×100 , 1000×1000 , and 10000×10000 . Again the table shows both the absolute time for running Dijkstra's algorithm, as well as the difference percentage compared to the algorithm using `std::priority_queue` and duplicate queue elements.

The matrix graph tests show that when the size of the graph (and thus the size of the priority queue) grows, the mutable algorithms become slower compared to the basic STL priority queue. Again this is as expected because in

size	(a) std::p_q		(b) sep. index		(c) inheritance		(d) locator		(e) std::p_q&loc	
10 ²	2.05μs	0.00%	1.15μs	-44.0%	1.62μs	-20.8%	1.65μs	-19.4%	1.84μs	-10.1%
100 ²	390μs	0.00%	324μs	-17.0%	342μs	-12.3%	350μs	-10.4%	401μs	2.72%
1000 ²	63.8ms	0.00%	70.3ms	10.3%	63.4ms	-0.52%	62.1ms	-2.62%	61.0ms	-4.26%
10000 ²	10.0s	0.00%	18.7s	86.9%	11.9s	19.0%	11.6s	15.6%	15.0s	50.3%

Fig. 6. Results of performance tests with matrix graphs

this test priority changes do not occur, but the mutable queues have to pay for updating the location index of queue elements. It is interesting to see, however, that the mutable algorithms are still somewhat faster for matrix graphs smaller than 1000×1000 . This is again probably caused by the additional distance field needed in the unmutable duplicate-based version. It can also be seen that the mutable version using a separate internal location array is the fastest for small graphs, but becomes the slowest for large ones. No obvious reason for this behaviour was found, but differences in memory locality are one possible explanation.

Finally Figure 7 shows the results when the algorithms were used on the San Francisco road network. The figures are averages of 10 test runs with random starting points (the same starting points were used for all algorithms). Individual runs produced results that differed at most 1.7 % from the average, so they are not shown.

size	(a) std::p_q		(b) sep. index		(c) inheritance		(d) locator		(e) std::p_q&loc	
174956	18.7ms	0.00%	20.0ms	7.34%	19.8ms	6.00%	19.7ms	5.47%	18.3ms	-2.14%

Fig. 7. Results of performance tests with SF road map graph

In this “real” test differences between the algorithms were relatively small. Most of the mutable algorithms were 5–7 % slower than the basic STL priority queue -based one. The results suggest that this graph did not contain enough priority changes for mutable versions to win, and it did not cause the priority queue to grow large enough for the basic version to be clearly faster. Perhaps surprisingly, the modified STL priority queue -based algorithm was 2 % faster than the others. This can probably be attributed to more optimized queue/heap algorithms in the STL priority queue, which can tip the balance a bit to the other direction.

5 Conclusion

The measurements in Section 4 demonstrate that either the simple implementation or our designs are better, depending on the nature of the input graph. With

graphs where the same vertex is found several times with decreasing distance, the performance benefit provided by our designs is significant. On the other hand, with graphs that resemble road maps (Figure 6 and 7), the new designs were slower, sometimes significantly.

Adding the priority change operation to the interface of the priority queue proved to be a trade-off between simplicity and generality. Fortunately, the most general of our designs, the locator function -based, was never so much slower in our experiments that it should be rejected on that basis. Therefore, the locator function -based design seems suitable for a general purpose library.

Acknowledgement. We thank the anonymous reviewers for helpful comments.

References

1. Alexander, R. & Bensley, G.: *C++ Footprint and Performance Optimization*. SAMS Publishing, 2000
2. Blechmann, T.: *Boost.Heap*. http://www.boost.org/doc/libs/1_59_0/doc/html/heap.html, page last modified 4th August 2015, contents checked 20th August 2015
3. Bryant, R. E. & O'Hallaron, D. R.: *Computer Systems: A Programmer's Perspective*, Chapter 9. Measuring Program Execution Time. Prentice Hall 2003
4. Chen, M. & Chowdhury, R. A. & Ramachandran, V. & Lan Roche, D. & Tong, L.: *Priority Queues and Dijkstra's Algorithm*. UTCS Technical Report TR-07-54, The University of Texas at Austin, Department of Computer Science, October 12, 2007
5. Cormen, T. H. & Leiserson, C. E. & Rivest, R. L. & Stein, C.: *Introduction to Algorithms, 3rd edition*. The MIT Press 2009
6. Feifei, L.: *Real Datasets for Spatial Databases: Road Networks and Points of Interest*. <http://www.cs.fsu.edu/~lifeifei/SpatialDataset.htm>, page last modified 9th September 2005, contents checked 11th August 2015
7. ISO/IEC: *Standard for Programming Language C++*. ISO/IEC 2011
8. Josuttis, N.: *The C++ Standard Library, 2nd. Edition*. Addison-Wesley 2012

Two Set-based Implementations of Quotients in Type Theory

Niccolò Veltri

Institute of Cybernetics, Tallinn University of Technology
 Akadeemia tee 21, 12618 Tallinn, Estonia,
 niccolo@cs.ioc.ee

Abstract. We present and compare two different implementations of quotient types in Intensional Type Theory. We first introduce quotients as particular inductive-like types following Martin Hofmann’s extension of Calculus of Constructions with quotient types [6]. Then we give an impredicative encoding of quotients. This implementation is reminiscent of Church numerals and more generally of encodings of inductive types in Calculus of Constructions.

1 Introduction

In mathematics, given a set X and an equivalence relation R on X , the quotient set X/R is the set of equivalence classes of X with respect to R , i.e. $X/R = \{[x] \mid x \in X\}$, where $[x] = \{y \in X \mid x R y\}$. An important example is the set of integer numbers, constructed as the quotient set $(\mathbb{N} \times \mathbb{N})/\text{SameDiff}$, where $\text{SameDiff}(n_1, m_1)(n_2, m_2)$ if and only if $n_1 + m_2 = n_2 + m_1$. Another example is the set of real numbers, constructed as the quotient set $\text{Cauchy}_{\mathbb{Q}}/\text{Diff}_{\rightarrow 0}$, where $\text{Cauchy}_{\mathbb{Q}}$ is the set of Cauchy sequences of rational numbers and $\text{Diff}_{\rightarrow 0} \{x_n\} \{y_n\}$ if and only if the sequence $\{x_n - y_n\}$ converges to 0. A fundamental usage of quotients in programming is the construction of finite multisubsets of a given type X as “lists modulo permutations”, and of finite subsets of X as “lists modulo permutations and multiplicity”.

In Martin-Löf type theory (MLTT) [8] and in Calculus of Inductive Constructions (CIC) [1] quotients are typically represented by setoids. A setoid is a pair (A, R) where A is a set and R is an equivalence relation on A . A map between setoids (A, R) and (B, S) is a map $f : A \rightarrow B$ compatible with the relations, i.e. if $a R b$ then $(fa) S (fb)$. Every set A can be represented as the setoid (A, \equiv) , where \equiv is propositional equality on A . Given an equivalence relation R on A the quotient A/R is represented as the setoid (A, R) . There is a canonical setoid map $\text{abs} : (A, \equiv) \rightarrow (A, R)$, $\text{abs} = \text{id}$, that is clearly compatible, and every setoid map $f : (A, \equiv) \rightarrow (B, S)$ such that $(fa) S (fb)$ whenever $a R b$ extends to a setoid map $\text{lift } f : (A, R) \rightarrow (B, S)$.

The implementation of quotients as setoids forces us to lift every type former to setoids. For example the type formers of products, function spaces, lists and trees must become setoid transformers. Moreover in several applications it is preferable to work with sets instead of setoids.

In this paper we present two different frameworks for reasoning about set-based quotients, i.e. quotients as types. We first introduce in Section 2 quotients as particular inductive-like types. The presentation is inspired by quotient types in Martin Hofmann’s PhD thesis [6], and works fine both in MLTT and in CIC. Our presentation is settled in MLTT. In Section 3 we show an alternative encoding of quotients in a small extension of Calculus of Constructions (CC). The two implementations are pretty different in flavor. We highlight their main features and show some examples. In Section 4 we present integer numbers as the quotient of $\mathbb{N} \times \mathbb{N}$ mentioned in the introduction, and in Section 5 we present finite multisubsets of a given type X as the quotient of $\text{List } X$ also mentioned above. The presentations work fine both in MLTT and in our extension of CC.

Note that integer numbers are already definable in type theory without the need of quotient types. In MLTT, for example, integers are implemented as two distinct copies of natural numbers $\mathbb{N} + \mathbb{N}$, interpreted as the negative and non-negative numbers. Note that in order to avoid the presence of two zeros, the elements of the first copy of \mathbb{N} have to be considered as “shifted by one”, i.e. $\text{inl } n$ has to be read as $-(n + 1)$. Another possibility is to introduce integers as the type $\top + \mathbb{N} + \mathbb{N}$, specifying zero explicitly and “shifting by one” both copies. Using such implementations, defining operations on integers and proving that such operations satisfy the laws of arithmetic (e.g. \mathbb{Z} is a integral domain) become tedious due to the number of cases involved in the definitions. In Section 4 we want to show that our implementation is more elegant and less tedious to work with than the other two presented above.

We have fully formalized the results of this paper in the dependently typed programming language Agda [9]. The formalization is available at <http://cs.ioc.ee/~niccolo/quotients/>. In order to be consistent with the formalization, in this paper we use the notation of MLTT.

2 Inductive-Like Quotients

In this section, we introduce quotient types as particular inductive-like types introduced by M. Hofmann [6]. First we briefly describe the type theory under consideration.

2.1 The Type Theory under Consideration

We consider Martin-Löf type theory (MLTT) with inductive types and a cumulative hierarchy of universes \mathcal{U}_k . We allow dependent functions to have implicit arguments and indicated implicit argument positions with curly brackets (as in Agda). We write \equiv for propositional equality (identity types) and $=$ for judgmental (definitional) equality. Reflexivity, symmetry, transitivity and substitutivity of \equiv are named *refl*, *sym*, *trans* and *subst*, respectively.

We assume *uniqueness of identity proofs* for all types, i.e., an inhabitant for

$$\text{UIP} = \prod_{\{X:\mathcal{U}\}} \prod_{\{x_1, x_2:X\}} \prod_{p_1, p_2:x_1 \equiv x_2} p_1 \equiv p_2.$$

A type X is said to be a *proposition*, if it has at most one inhabitant, i.e., if the type

$$\text{isProp } X = \prod_{x_1, x_2 : X} x_1 \equiv x_2$$

is inhabited.

Uniqueness of identity proofs is needed only to prove that the propositional truncation of a type is a proposition (Subsection 2.4), which in turn is needed in the proof of Proposition 1.

2.2 The Implementation

We now describe quotient types à la Hofmann. We call them “inductive-like quotients” because they are given a dependent elimination principle (sometimes also called induction principle). Let X be a type and R an equivalence relation on X . For any type Y and function $f : X \rightarrow Y$, we say that f is *R -compatible* (or simply *compatible*, when the intended equivalence relation is clear from the context), if the type

$$\text{compat } f = \prod_{\{x_1, x_2 : X\}} x_1 R x_2 \rightarrow f x_1 \equiv f x_2$$

is inhabited. The quotient of X by the relation R is described by the following data:

- (i) a carrier type X/R ;
- (ii) a constructor $\text{abs} : X \rightarrow X/R$ together with a proof $\text{sound} : \text{compat } \text{abs}$;
- (iii) a dependent eliminator: for every family of types $Y : X/R \rightarrow \mathcal{U}_k$ and function $f : \prod_{x : X} Y (\text{abs } x)$ with $p : \text{dcompat } f$, there exists a function $\text{lift } f p : \prod_{q : X/R} Y q$;
- (iv) a computation rule: for every family of types $Y : X/R \rightarrow \mathcal{U}_k$, function $f : \prod_{x : X} Y (\text{abs } x)$ with $p : \text{dcompat } f$ and $x : X$ we have

$$\text{lift}_\beta f p x : \text{lift } f p (\text{abs } x) \equiv f x$$

The predicate dcompat represents compatibility for dependent functions $f : \prod_{x : X} Y (\text{abs } x)$:

$$\text{dcompat } f = \prod_{\{x_1, x_2 : X\}} \prod_{r : x_1 R x_2} \text{subst } Y (\text{sound } r) (f x_1) \equiv f x_2.$$

We postulate the existence of data (i)–(iv) for all types X and equivalence relations R on X . Notice that the predicate dcompat depends on the availability of sound . Also notice that, in (iii), we allow elimination on every universe \mathcal{U}_k . In our development, we actually eliminate only on \mathcal{U} and once on \mathcal{U}_1 (Proposition 1).

We now take a look at some derived results and examples.

2.3 Classical Quotients

Classically every equivalence class in a quotient X/R has a representative element in the original set, i.e. a map $\text{rep} : X/R \rightarrow X$ that satisfies the following conditions:

$$\begin{aligned} \text{complete} &: \prod_{x:X} (\text{rep}(\text{abs } x)) R x \\ \text{stable} &: \prod_{q:X/R} \text{abs}(\text{rep } q) \equiv q \end{aligned}$$

If we postulate the existence of such quotients for all sets and equivalence relations it is possible to derive the law of excluded middle [2].

In general in constructive mathematics, for a given equivalence class there is no canonical choice of a representative. This idea is reflected in the implementation of quotients we presented in the previous section. Every map of type $X/R \rightarrow X$ is of the form $\text{lift } f p$ for a certain R -compatible map $f : X \rightarrow X$. But for a general type X and equivalence relation R strictly weaker than equality, there is no such canonical f .

2.4 Propositional Truncation

The *propositional truncation* (or *squash*) $\|X\|$ of a type X is the quotient of X by the total relation $\lambda x_1 x_2. \top$. Intuitively $\|X\|$ is the unit type \top if X is inhabited and it is empty otherwise. In other words, $\|X\|$ is the proposition associated with the type X . Indeed:

$$\begin{aligned} \text{isProp}_{\parallel} &: \text{isProp } \|X\| \\ \text{isProp}_{\parallel} x_1 x_2 &= \text{lift } (\lambda y_1. \text{lift } (\lambda y_2. \text{sound } \star) p_1 x_2) p_2 x_1 \end{aligned}$$

where $\star : \top$ is the constructor of the unit type, while p_1 and p_2 are simple compatibility proofs. Note that in these compatibility proofs we need to show that two equality proofs are equal, and we do it by using the uniqueness of identity proofs.

Note that the propositional truncation operation defines a monad: the unit is $|_|$ and multiplication $\mu_{\parallel} : \|\|X\|\| \rightarrow \|X\|$ is defined as $\mu_{\parallel} = \text{lift id } p$, where p is the easy proof of compatibility that follows from the fact that $\|X\|$ is a proposition. In general, for a given family of equivalence relations $R_X : X \rightarrow X \rightarrow \mathcal{U}$, indexed by $X : \mathcal{U}$, the functor $F X = X/R_X$ is not a monad, since there is no way of constructing a multiplication $\mu : (X/R_X)/R_{X/R_X} \rightarrow X/R_X$.

2.5 Function Extensionality

Let X and Y be types. Extensional equality of functions is an equivalence relation on $X \rightarrow Y$:

$$\begin{aligned} \text{FunExt}_{\equiv} &: (X \rightarrow Y) \rightarrow (X \rightarrow Y) \rightarrow \mathcal{U} \\ \text{FunExt}_{\equiv} f g &= \prod_{x:X} f x \equiv g x \end{aligned}$$

For the quotient $(X \rightarrow Y)/\text{FunExt}_{\equiv}$ there exists a map that associates a representative function to each equivalence class.

$$\begin{aligned} \text{rep} &: (X \rightarrow Y)/\text{FunExt}_{\equiv} \rightarrow (X \rightarrow Y) \\ \text{rep } q x &= \text{lift } (\lambda f. f x) (\lambda p. p x) q \end{aligned}$$

Using the computation rule lift_{β} of quotients we obtain $\text{rep } (\text{abs } f) x \equiv f x$, for all $f : X \rightarrow Y$ and $x : X$. The computation rule holds only up to propositional equality. If equality in lift_{β} were definitional, one could prove, using rep , the principle of function extensionality. Indeed, consider $f, g : X \rightarrow Y$ with $\text{FunExt}_{\equiv} f g$. Then the following sequence of equations holds:

$$\begin{aligned} f &= \lambda x. f x = \lambda x. \text{rep } (\text{abs } f) x = \text{rep } (\text{abs } f) \\ &\equiv \text{rep } (\text{abs } g) = \lambda x. \text{rep } (\text{abs } g) x = \lambda x. g x = g \end{aligned}$$

2.6 Effectiveness

A quotient X/R is said to be *effective*, if the type $\prod_{x_1, x_2 : X} \text{abs } x_1 \equiv \text{abs } x_2 \rightarrow x_1 R x_2$ is inhabited. In general, effectiveness does not hold for all quotients. Moreover, postulating effectiveness for all quotients implies the law of excluded middle [7]. Clearly classical quotients, discussed in Subsection 2.3, are effective. Indeed, if for $x_1, x_2 : X$ we have $\text{abs } x_1 \equiv \text{abs } x_2$ then, using complete we are done, since $\text{rep } (\text{abs } x_1) R x_1$, $\text{rep } (\text{abs } x_2) R x_2$ and $\text{rep } (\text{abs } x_1) \equiv \text{rep } (\text{abs } x_2)$.

For a general type X and a general equivalence relation R on X , we can only prove that, under the assumption of proposition extensionality, the quotient X/R satisfies a weaker property. The principle of *proposition extensionality* states that logically equivalent propositions are equal:¹

$$\text{PropExt} = \prod_{\{X, Y : \mathcal{U}\}} \text{isProp } X \rightarrow \text{isProp } Y \rightarrow X \leftrightarrow Y \rightarrow X \equiv Y$$

where $X \leftrightarrow Y = (X \rightarrow Y) \times (Y \rightarrow X)$. We say that a quotient X/R is *weakly effective*, if the type $\prod_{x_1, x_2 : X} \text{abs } x_1 \equiv \text{abs } x_2 \rightarrow \|x_1 R x_2\|$ is inhabited.

If we extend our type theory with PropExt , we can prove that all quotients are weakly effective.

Proposition 1. *Under the hypothesis of proposition extensionality, all quotients are weakly effective.*

Proof. In fact, let X be a type, R an equivalence relation on X and $x : X$. Consider the function $\|x R _ \| : X \rightarrow \mathcal{U}$, $\|x R _ \| = \lambda x'. \|x R x'\|$. We show that $\|x R _ \|$ is R -compatible. Let $x_1, x_2 : X$ with $x_1 R x_2$. We have $x R x_1 \leftrightarrow x R x_2$ and therefore $\|x R x_1\| \leftrightarrow \|x R x_2\|$. Since propositional truncations are propositions

¹ Note that proposition extensionality is accepted in homotopy type theory [12]. Propositions are (-1)-types and proposition extensionality is univalence for (-1)-types.

(proof is `Prop` in Subsection 2.4), using proposition extensionality, we conclude $\|x R x_1\| \equiv \|x R x_2\|$. We have constructed a term $p_x : \mathbf{compat} \|x R _ \|$, and therefore a function $\mathbf{lift} \|x R _ \| p_x : X/R \rightarrow \mathcal{U}$ (large elimination is fundamental in order to apply `lift`, since $\|x R _ \| : X \rightarrow \mathcal{U}$ and $X \rightarrow \mathcal{U} : \mathcal{U}$). Moreover, $\mathbf{lift} \|x R _ \| p_x (\mathbf{abs} y) \equiv \|x R y\|$ by its computation rule.

Let $\mathbf{abs} x_1 \equiv \mathbf{abs} x_2$ for some $x_1, x_2 : X$. We have:

$$\|x_1 R x_2\| \equiv \mathbf{lift} \|x_1 R _ \| p_{x_1} (\mathbf{abs} x_2) \equiv \mathbf{lift} \|x_1 R _ \| p_{x_1} (\mathbf{abs} x_1) \equiv \|x_1 R x_1\|$$

and $x_1 R x_1$ holds, since R is reflexive. \square

3 Impredicative Encoding of Quotients

In this section, we present an implementation of quotients in Calculus of Constructions (CC). The implementation is different in flavor from the one discussed in Section 2.

3.1 The Type Theory under Consideration

Remember that our presentation is done using the language of MLTT. Our Agda formalization makes use of type-in-type instead of Agda's current implementation of universe polymorphism. This means that we are working in a type theory with only one universe \mathcal{U} and $\mathcal{U} : \mathcal{U}$. Type-in-type is known to be inconsistent [5, 3], but we are using it only to simulate in Agda the impredicativity of CC, which is consistent.

In Subsection 3.3 we need the existence of dependent sums and identity types. Both are definable in CC. Consider $X : \mathcal{U}$ and $P : X \rightarrow \mathcal{U}$. The dependent sum $\sum_{x:X} P x$ can be defined as follows:

$$\sum_{x:X} P x = \prod_{Y:\mathcal{U}} \left(\prod_{x:X} P x \rightarrow Y \right) \rightarrow Y$$

Consider $X : \mathcal{U}$ and $x_1, x_2 : X$. We can define (*Leibniz*) equality $x_1 \equiv x_2$ as follows:

$$x_1 \equiv x_2 = \prod_{P:X \rightarrow \mathcal{U}} P x_1 \rightarrow P x_2$$

One can easily define the constructor and the first projection map of dependent sums.

$$\mathbf{pair} : \prod_{x:X} \left(P x \rightarrow \sum_{x:X} P x \right)$$

$$\mathbf{pair} x p = \lambda Y f. f x p$$

$$\mathbf{fst} : \sum_{x:X} P x \rightarrow X$$

$$\mathbf{fst} c = c X (\lambda x p. x)$$

It is also possible to prove that Leibniz equality is a substitutive equivalence relation. But it is not possible to construct the second projection map $\text{snd} : \prod_{c:\sum_{x:X} P x} P(\text{fst } c)$, showing that the type $\sum_{x:X} P x$ defined above is a *weak* dependent sum. Leibniz equality is also weak, since it is not possible to prove “dependent substitutivity”, i.e. given a type X , a family of types $Y : X \rightarrow \mathcal{U}$ and a predicate $P : \prod_{x:X} Y x \rightarrow \mathcal{U}$, we cannot construct a term subst_2 of type

$$\prod_{p:x_1 \equiv x_2} \text{subst } Y p y_1 \equiv y_2 \rightarrow P x_1 y_1 \rightarrow P x_2 y_2$$

for all $x_1, x_2 : X$, $y_1 : Y x_1$ and $y_2 : Y x_2$.

The results of Subsection 3.3 rely on the existence of terms snd and subst_2 . Therefore we extend CC with identity types and dependent sums as primitives. As a consequence we obtain that the terms snd and subst_2 are easily definable. An instance of subst_2 gives us sufficient conditions for proving equality of pairs. Let X be a type and $P : X \rightarrow \mathcal{U}$ a family of types. Then for all $x_1, x_2 : X$, $p_1 : P x_1$ and $p_2 : P x_2$:

$$\begin{aligned} \text{pair}_{\equiv} &: \prod_{r:x_1 \equiv x_2} \text{subst } P r p_1 \equiv p_2 \rightarrow \text{pair } x_1 p_1 \equiv \text{pair } x_2 p_2 \\ \text{pair}_{\equiv} r s &= \text{subst}_2 (\lambda x p. \text{pair } x_1 p_1 \equiv \text{pair } x p) r s \text{ refl} \end{aligned}$$

We also assume the dependent version of the principle of function extensionality, i.e. there is a term dfunext that inhabits the type

$$\text{DFunExt} = \prod_{\{X:\mathcal{U}\}} \prod_{\{Y:X \rightarrow \mathcal{U}\}} \prod_{\{f_1 f_2:\prod_{x:X} Y x\}} \left(\prod_{x:X} f_1 x \equiv f_2 x \right) \rightarrow f_1 \equiv f_2$$

3.2 The Implementation

We now describe our impredicative implementation of quotients. Let X be a type and R an equivalence relation on X . We define the quotient of X over R as the following type:

$$X/R = \prod_{Y:\mathcal{U}} \prod_{f:X \rightarrow Y} \text{compat } f \rightarrow Y$$

In other words, X/R is a polymorphic function which assigns, to every type Y equipped with a compatible function $f : X \rightarrow Y$, an element of Y . One can then define the constructor abs :

$$\begin{aligned} \text{abs} &: X \rightarrow X/R \\ \text{abs } x &= \lambda Y f r. f x \end{aligned}$$

Using the principle of function extensionality one proves that abs is an R -compatible map. Notice that the dependent version of the principle of function

extensionality is needed here, since elements of type X/R are dependent maps.

`sound` : `compat abs`
`sound r` = `dfunext (λY. dfunext (λf. dfunext (λp. p r)))`

One can then define the non-dependent elimination principle, which turns out to be just function application. Crucially the computation rule holds definitionally, as witnessed below in the observation that `refl` proves the corresponding propositional equality.

$$\begin{aligned} \text{lift} &: \prod_{\{Y:\mathcal{U}\}} \prod_{f:X \rightarrow Y} \text{compat } f \rightarrow X/R \rightarrow Y \\ \text{lift } \{Y\} f r q &= q Y f p \\ \\ \text{lift}_\beta &: \prod_{\{Y:\mathcal{U}\}} \prod_{f:X \rightarrow Y} \prod_{r:\text{compat } f} \prod_{x:X} \text{lift } f p (\text{abs } x) \equiv f x \\ \text{lift}_\beta f r x &= \text{refl} \end{aligned}$$

Note the similarity with Church numerals and the implementation of dependent sums given above, and more generally the similarity with the impredicative encoding of inductive types in CC [10]. Moreover this representation is inspired by the impredicative encoding of higher inductive types [12, Ch. 6] in CIC [11].

3.3 Dependent Elimination

While in practice having a definitional computation rule is convenient, it is impossible to derive a dependent elimination principle. Implementations of inductive types in CC in general suffer from this problem [4].

In this subsection we assume the *uniqueness property* of `lift` i.e. the fact that, for every type Y and R -compatible function $f : X \rightarrow Y$, `lift f r` is the only map that makes the following diagram commute:

$$\begin{array}{ccc} X & \xrightarrow{f} & Y \\ \text{abs} \downarrow & \dashrightarrow & \uparrow \text{lift } f r \\ X/R & & \end{array}$$

From the uniqueness property we derive the dependent elimination principle. Let $Y : X/R \rightarrow \mathcal{U}$ be a type family and $f : \prod_{x:X} Y(\text{abs } x)$ a map with compatibility proof $r : \text{dcompat } f$. Using the non-dependent eliminator we define a map of type $X/R \rightarrow \sum_{q:X/R} Y q$.

$$\begin{aligned} \text{dlift}' &: \prod_{\{Y:X \rightarrow \mathcal{U}\}} \prod_{f:\prod_{x:X} Y(\text{abs } x)} \text{dcompat } f \rightarrow X/R \rightarrow \sum_{q:X/R} Y q \\ \text{dlift}' f r &= \text{lift } (\lambda x. \text{pair } (\text{abs } x) (f x)) (\lambda p. \text{pair}_\equiv (\text{sound } p) (r p)) \end{aligned}$$

Notice that, for all $x : X$, $\text{fst}(\text{dlift}' f r (\text{abs } x)) = \text{abs } x = \text{id}(\text{abs } x)$. Therefore, by the uniqueness property, we obtain a term $s_q : \text{fst}(\text{dlift}' f r q) \equiv q$ for all $q : X/R$. This allows us to derive the dependent elimination principle:

$$\begin{aligned} \text{dlift} &: \prod_{\{Y : X \rightarrow \mathcal{U}\}} \prod_{f : \prod_{x : X} Y(\text{abs } x)} \text{dcompat } f \rightarrow \prod_{q : X/R} Y q \\ \text{dlift } \{Y\} f r q &= \text{subst } Y s_q (\text{snd}(\text{dlift}' f r q)) \end{aligned}$$

4 Integer Numbers

As an example we present integer numbers. In order to do that we need to have natural numbers in our system (defined as Church numerals in CC or defined inductively in MLTT, it does not matter). We introduce a synonym for pairs of natural numbers, $\text{Diff} = \mathbb{N} \times \mathbb{N}$, and we use the notation $_-_$ for the constructor of Diff . Elements of Diff represent differences of natural numbers. We define an equivalence relation SameDiff on Diff relating pairs with the same difference:

$$\begin{aligned} \text{SameDiff} &: \text{Diff} \rightarrow \text{Diff} \rightarrow \mathcal{U} \\ \text{SameDiff } (n_1 - m_1) (n_2 - m_2) &= \text{plus } n_1 m_2 \equiv \text{plus } n_2 m_1 \end{aligned}$$

where plus is addition on \mathbb{N} . We define $\mathbb{Z} = \text{Diff}/\text{SameDiff}$. We show formally that \mathbb{Z} is a commutative monoid. The unit $\text{zero}_{\mathbb{Z}}$ is the equivalence class of $\text{zero}_{\text{Diff}} = \text{zero} - \text{zero}$, where zero is the unit of \mathbb{N} . Addition is defined in two steps. First we introduce an addition operation on Diff .

$$\begin{aligned} \text{plus}_{\text{Diff}} &: \text{Diff} \rightarrow \text{Diff} \rightarrow \text{Diff} \\ \text{plus}_{\text{Diff}} (n_1 - m_1) (n_2 - m_2) &= \text{plus } n_1 n_2 - \text{plus } m_1 m_2 \end{aligned}$$

Before lifting addition to \mathbb{Z} , we introduce a useful variant of compat_2 , the compatibility predicate for two-argument functions. Let X, Y and Z be types and R, S and T equivalence relations on X, Y and Z respectively. The predicate compat'_2 on $X \rightarrow Y \rightarrow Z$ is defined as follows:

$$\text{compat}'_2 f = \prod_{\{x_1, x_2 : X\}} \prod_{\{y_1, y_2 : Y\}} x_1 R x_2 \rightarrow y_1 S y_2 \rightarrow (f x_1 y_1) T (f x_2 y_2)$$

A function f satisfies compat'_2 if it sends R -related and S -related inputs to T -related outputs. It is easy to construct a proof $p : \text{compat}'_2 \text{plus}_{\text{Diff}}$. We are ready to lift the addition $\text{plus}_{\text{Diff}}$ to \mathbb{Z} :

$$\begin{aligned} \text{plus}_{\mathbb{Z}} &: \mathbb{Z} \rightarrow \mathbb{Z} \rightarrow \mathbb{Z} \\ \text{plus}_{\mathbb{Z}} &= \text{lift}_2 (\lambda d e. \text{abs}(\text{plus}_{\text{Diff}} d e)) (\lambda r s. \text{sound}(p r s)) \end{aligned}$$

where lift_2 is the two-argument version of lift . We prove the right unit law. First notice that the law holds in Diff up to SameDiff , i.e. for all $d : \text{Diff}$, we have a

proof $s_d : \text{SameDiff} (\text{plus}_{\text{Diff}} d (\text{zero} - \text{zero})) d$. We lift this proof to \mathbb{Z} :

$$\begin{aligned} \text{rightUnit}_{\mathbb{Z}} &: \prod_{z:\mathbb{Z}} \text{plus}_{\mathbb{Z}} z \text{zero}_{\mathbb{Z}} \equiv z \\ \text{rightUnit}_{\mathbb{Z}} &= \text{absEpi} (\lambda d. \text{sound } s_d) \end{aligned}$$

where absEpi is a proof that the map $\text{abs} : X \rightarrow X/R$ is an epimorphism, for all types X and equivalence relations R on X , i.e. for all types Y and maps $f_1, f_2 : X/R \rightarrow Y$, if $f_1 (\text{abs } x) \equiv f_2 (\text{abs } x)$ for all $x : X$, then for all $q : X/R$ we have $f_1 q \equiv f_2 q$. This is an easy consequence of the uniqueness property.

We observe that working with impredicative quotients facilitates proofs, since the computation rule holds definitionally.

5 Finite Multisubsets

Another example we present is finite multisubsets of a given type X . In this section we work in MLTT. Let X be a type with decidable equality, i.e. there exists a function $\text{dec}_{\equiv} : X \rightarrow X \rightarrow \text{Bool}$ such that $\text{dec}_{\equiv} x_1 x_2 = \text{true}$ if and only if $x_1 \equiv x_2$. We introduce the binary relation Perm on $\text{List } X$, inductively defined by the rules:

$$\begin{array}{c} \frac{}{\text{Perm } [] []} \qquad \frac{\text{Perm } xs \ ys}{\text{Perm } (x :: xs) (x :: ys)} \\ \frac{\text{Perm } xs \ ys}{\text{Perm } (x :: y :: xs) (y :: x :: ys)} \qquad \frac{\text{Perm } xs \ ys \quad \text{Perm } ys \ zs}{\text{Perm } xs \ zs} \end{array}$$

Two lists xs and ys are in the relation Perm if xs is a permutation of ys . The relation is transitive by construction, and it is easily provable reflexive and symmetric. Therefore we form the quotient $\text{Multisubset } X = \text{List } X / \text{Perm}$, i.e. a finite multisubset of X is a list modulo permutations.

We introduce a function counting the multiplicity of an element x in a list xs . If the element does not belong to the list, then its multiplicity is zero. Note that decidable equality on X is fundamental in order to count the number of occurrences of x in xs .

$$\begin{aligned} \text{multiplicity} &: X \rightarrow \text{List } X \rightarrow \mathbb{N} \\ \text{multiplicity } x &[] = \text{zero} \\ \text{multiplicity } x (y :: xs) &\text{ with } \text{dec}_{\equiv} x y \\ \text{multiplicity } x (y :: xs) &| \text{true} = \text{suc } (\text{multiplicity } x xs) \\ \text{multiplicity } x (y :: xs) &| \text{false} = \text{multiplicity } x xs \end{aligned}$$

The function multiplicity can be proved compatible with the relation Perm . This is true since permuting a list does not alter the number of occurrences of an element in it. The proof is easily done by induction on the structure of Perm . Therefore the function multiplicity lifts to $\text{Multisubset } X$.

We conclude this section by noting that there are other possible definitions of “equality” on finite multisubsets of X . For example one could define a relation Perm' on $\text{List } X$ as $\text{Perm}' xs ys = \prod_{x:X} (x \in xs) \cong (x \in ys)$, where \cong is type isomorphism and \in is list membership. The definition of Perm' is more concise than the definition of Perm . The two relations are logically equivalent, but proving multiplicity compatible with Perm' is much more complicated than proving multiplicity compatible with Perm .

6 Conclusions

In this paper we showed two different implementation of quotient types. Both are set-based and therefore different from the setoid-based approach.

In Section 2 we presented inductive-like quotients in Martin-Löf type theory. They do not need impredicativity in order to be introduced, but their existence has to be postulated. Moreover the computation rule only holds up to propositional equality. Hofmann’s extension of Calculus of Constructions [6] is consistent, therefore the same holds for our implementation in MLTT.

In Section 3 we presented an impredicative encoding of quotients in Calculus of Constructions. In order to derive the dependent elimination principle from the uniqueness property we need to extend CC with dependent sums and identity types. Our implementation shows that, at the cost of impredicativity, quotient types are definable. However they are “weak”, similarly to Leibniz equality or the impredicative encoding of dependent sums given in Subsection 3.2. To get “strong” quotients, one needs to introduce postulates, such as the uniqueness property. Geuvers [4] showed that postulating dependent elimination for impredicative encodings of inductive types is safe. Similarly this can be extended to our quotient types. The uniqueness property of quotients is logically equivalent to the dependent elimination principle, therefore assuming the uniqueness property is also safe. There are other ways of deriving the dependent elimination principle for inductive types in impredicative systems such as CC, most notably parametricity [13].

Acknowledgement This research was supported by the ERDF funded ICT national programme project ”Coinduction”, the Estonian Science Foundation grant no. 9219 and the Estonian Ministry of Education and Research institutional research grant no. PUT33-13.

References

1. Y. Bertot and P. Castéran. *Interactive theorem proving and program development: Coq’Art: the calculus of inductive constructions*. Springer, 2004.
2. L. Chicli, L. Pottier, and C. Simpson. Mathematical quotients and quotient types in Coq. In H. Geuvers and F. Wiedijk, editors, *Types for Proofs and Programs*, volume 2646 of *Lecture Notes in Computer Science*, pages 95–107. Springer, 2003.
3. T. Coquand. An analysis of Girard’s paradox. In *Symposium on Logic in Computer Science*, pages 227–236. IEEE Computer Society, 1986.

4. H. Geuvers. Induction is not derivable in second order dependent type theory. In S. Abramsky, editor, *Typed Lambda Calculi and Applications*, volume 2044 of *Lecture Notes in Computer Science*, pages 166–181. Springer, 2001.
5. J.-Y. Girard. Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur, Ph.D. thesis, Université Paris VII, 1972.
6. M. Hofmann. Extensional concepts in intensional type theory, Ph.D. thesis, University of Edinburgh, 1995.
7. M. Maietti. About effective quotients in constructive type theory. In T. Altenkirch, B. Reus, and W. Naraschewski, editors, *Types for Proofs and Programs*, volume 1657 of *Lecture Notes in Computer Science*, pages 166–178. Springer, 1999.
8. B. Nordström, K. Petersson, and J. M. Smith. *Programming in Martin-Löf's type theory*. Oxford University Press Oxford, 1990.
9. U. Norell. Dependently typed programming in Agda. In P. Koopman, R. Plasmeijer, and S. D. Swierstra, editors, *Advanced Functional Programming*, volume 5832 of *Lecture Notes in Computer Science*, pages 230–266. Springer, 2009.
10. F. Pfenning and C. Paulin-Mohring. Inductively defined types in the calculus of constructions. In M. G. Main, A. Melton, M. W. Mislove, and D. A. Schmidt, editors, *Mathematical Foundations of Programming Semantics*, volume 442 of *Lecture Notes in Computer Science*, pages 209–228. Springer, 1989.
11. M. Shulman. Higher inductive types via impredicative polymorphism. Blog post, 2011. <http://homotopytypetheory.org/2011/04/25/higher-inductive-types-via-impredicative-polymorphism>.
12. The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. Institute for Advanced Study, 2013. <http://homotopytypetheory.org/book>.
13. P. Wadler. The Girard–Reynolds isomorphism. *Theoretical Computer Science*, 375(1):201–226, 2007.

Preventing malicious attacks by diversifying Linux shell commands^{*}

Joni Uitto, Sampsa Rauti, Jari-Matti Mäkelä, and Ville Leppänen

University of Turku, 20014 Turku, Finland
 {jjuitt, sjprau, jmjmak, ville.leppanen}@utu.fi

Abstract. In instruction set diversification, a "language" used in a system is uniquely diversified in order to protect software against malicious attacks. In this paper, we apply diversification to Linux shell commands in order to prevent malware from taking advantage of the functionality they provide. When the Linux shell commands are diversified, malware no longer knows the correct commands and cannot use the shell to achieve its goals. We demonstrate this by using Shellshock as an example. This paper presents a scheme that diversifies the commands of Bash, the most widely used Linux shell and all the scripts in the system. The feasibility of our scheme is tested with a proof-of-concept implementation. We also present a study on the extent of changes required to make all the trusted scripts and applications in the system use the new diversified shell commands.

Keywords: software security, instruction set diversification, Linux command shell, Bash

1 Introduction

In this paper, we present a diversification scheme which prevents the execution of undiversified command shell scripts in order to protect the system from malware. While the focus of our discussion is on injection attacks such as Shellshock, our scheme also generally prevents attacks in many situations where the attacker tries to execute a malicious script in the target system.

Among security bugs, vulnerabilities allowing code injection attacks are probably most commonly exploited by malware [3]. In these attacks, the code is inserted in the vulnerable program, enabling the attacker to use the program's privileges to launch an attack. Code injection attacks are known to be popular with binaries compiled from weakly typed languages like C, but are also often used to execute arbitrary code on other environments like SQL [19] or Unix command shell [13]. In this paper, we concentrate on preventing attacks in command shell environment.

Malware and attackers make use of the fact that the set of commands interpreted by the command shell is identical on each computer. Because of this

^{*} The authors gratefully acknowledge Tekes – the Finnish Funding Agency for Innovation, DIGILE Oy and Cyber Trust research program for their support.

software monoculture, an adversary can design a single program that is able to successfully attack millions of vulnerable computers and devices. To defeat these kinds of attacks, we employ a method based on instruction set diversification.

The command sets of command shells on different computers, servers and devices can be uniquely diversified so that a piece of malware no longer knows the correct shell commands to perform a specific operation in order to access resources on a computer. As the malware is unfamiliar with the language used by the command shell, attempts to attack are rendered useless. Even if a piece of malware were to find out the secret diversified commands for one shell script, the same secret commands do not work for other scripts or systems. This diversification scheme can also be seen as proactive countermeasure against code injection attacks: The exact type of injection does not have to be known beforehand in order to thwart it.

It is also worth noting that diversification does not affect the software development process. The general idea in diversification (be it targeted at a command language or an API interface) is that software development is done against the ordinary reference language or API interface, and software artefacts are diversified machine-wisely after the development phase.

The contributions of this paper are as follows. We propose a scheme for diversifying Unix shell commands. Portokalidis et al. have briefly mentioned this idea in [15] among other possible applications of instruction set diversification. However, they do not go into much detail or provide any implementation for a diversified command shell. Our work can be seen as a continuation of this work, taking a more detailed and concrete approach on this issue. Our approach also significantly improves the security of their previous idea.

We present a proof-of-concept implementation of a diversified command shell, Bash, in order to demonstrate the feasibility of our approach in practice. We also show how our solution prevents code injection attacks, using different popular cases of Shellshock attack as examples. Additionally, we provide a brief study on the extent of changes required to make all the script files in two real life Linux distributions use the new diversified shell commands.

The rest of the paper is structured as follows. Section 2 describes the attack scenario. As an example, we describe Shellshock, an attack exploiting vulnerabilities in the Bash command shell and explain how our approach prevents this threat. In Section 3, we present our solution first as a general conceptual model and then as a practical implementation. Section 4 discusses the feasibility of shell diversification and presents some results on the number of the script files in two popular Linux distributions. The limitations of shell diversification are also covered. Section 5 contains the related work and Section 6 concludes the paper.

2 Attack scenario

Our solution aims to prevent the attacks where the attackers succeed to run malicious shell scripts or shell code fragments in the system. Code injection attacks are one typical way for adversaries to achieve this. Code injection usually

happens against interfaces where the target system requests data from the user. If the system doesn't properly handle this data, it may become susceptible to code injection attack. Malicious user has an opportunity to offer the system data containing code instructions that could get executed.

For example, in C programming language, the function `int system(const char *command)` from `stdlib.h` runs the given command string as a shell command:

```
char command[100] = "ls -l ";
char *user_input;

/* ask a file name from the user here and put it in user_input */

strcat(command, user_input); /* add a file name to the input */
system(command); /* execute as a shell command*/
```

Now, if the user would give the string `"; cat /etc/passwd"` as an input, contents of the password file would be printed.

As another example of a possible attack scenario, we discuss Shellshock [5], a family of security bugs found in the widely used Unix Bash shell, first discovered on 24 September 2014. While the vulnerabilities making this attack possible have been patched, similar attacks are possible in future. Shellshock would have easily been defeated by our approach. Bugs like Shellshock are very critical, because many services on Internet, like several web servers, use Bash to process certain requests.

The Shellshock attacks made use of vulnerabilities in Bash, a program that several Unix-based operating systems utilize to run command scripts. Bash is often installed as the operating system's standard command-line interface.

In Unix-based systems, every running program possesses a list of environment variables, which are basically name-value pairs. When a running program invokes another program, it gives an initial environment variable list to this new process. In addition, Bash also internally stores a list of functions that can be run from within the program. When Bash invokes itself as a child process, the original instance can pass the environment variables and function definitions on to the new subshell. More specifically, the function definitions reside in the environment variable list as encoded variables, the values of which start with parentheses followed up by a function definition. When the subshell starts, it changes these values back into internal functions. The piece of code in the value is executed and a function is created dynamically on the fly.

The problem is that the Bash version vulnerable to the attack does not perform any check to make sure that the code fragment is a valid function definition. Attacker therefore has a chance to run Bash with a freely chosen value in its environment variable list. This means the adversary can execute any commands of his or her choice. Naturally, this arbitrary code execution would not

be possible in the situation where the interpreter only accepts scripts conforming to the diversified script language.

As an example, Shellshock can be used to take control of a server. The following remote control attack attempts to use two programs – `wget` and `curl` – to connect to the attacker’s server and download a program that the attacker can then use to control the targeted server [5]:

```
() { :}; /bin/bash -c \"cd /tmp;wget http://213.x.x.x/ji;curl -O /tmp/ji http://213.x.x.x/ji ; perl /tmp/ji;rm -rf /tmp/ji\"
```

The downloaded Perl program is run immediately and remote access for the attacker is established.

Attacks like Shellshock can potentially compromise millions of servers and other systems. However, if our implementation is in place, a successful attack requires knowing the diversified shell commands, that is, the secret used to diversify the original commands. Without this knowledge, many security vulnerabilities become useless. It follows that our solution is also proactive in the sense that it does not depend on the exact attack vector as long as the adversary tries to use a shell language to perform the attack. In what follows, we will provide a detailed description of our scheme.

3 Our solution

3.1 The conceptual diversification scheme

In our conceptual diversification scheme, a diversifier tool is used to produce uniquely diversified script files. These scripts can then be run only by an interpreter that supports diversified scripts. The interpreter executes the script by making use of the secret that has been used to diversify the script file. As the malicious adversaries do not possess the diversification secret, they cannot diversify their malicious code fragments correctly and their attacks are thwarted.

Our diversification scheme is shown in Figure 1. Each diversified script has its own secret that is used to generate the diversified script file with a diversifier tool and execute it with a diversified interpreter. The tokens in the scripts are diversified by combining the semantic value (that is, the string presentation) of the original token and a unique tag. In this context, “token” means a collection of characters that is assigned a token identifier by the interpreter’s lexer. The tag is calculated using the secret and the semantic value of the token under diversification (see the circles in Figure 1). Simple concatenation can be used but it is also possible to use some cryptographic function to combine these parts. For example, our implementation appends a hash value to the original token.

Our method only diversifies the tokens that occur in the script, so the adversary has no way of knowing the diversified forms of other tokens even if he or she somehow get access to the script’s source code. It is worth noting that each token receives its own unique diversification. In this sense, we improve the

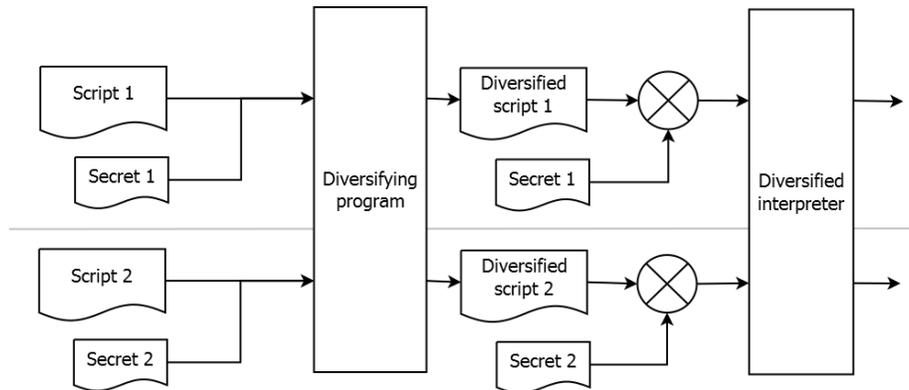


Fig. 1. Our conceptual diversification scheme.

solution suggested by Portokalidis et al. [15] where each token in a script file is diversified by appending the same secret tag to each token in a script file. Our scheme makes the diversification more secure by making the diversification of different tokens independent of each other. Taking this approach a step further, we can also vary the diversification of a token depending on the context it appears in. For example, the diversified form of a token can depend on preceding tokens or the location of the token in the script file. This makes it even harder for an attacker to guess the diversified forms of the tokens and inject anything into the script.

3.2 The practical implementation

Our proof-of-concept implementation of the diversified Bash shell was implemented by extending and modifying GNU Bash version 4.3.39. The implementation is written in C. The implementation and testing was performed using Ubuntu GNU/Linux 3.16.0-45-generic on 32-bit architecture. Our implementation itself is provided as an additional tool library, keeping direct modifications to the actual Bash interpreter minimal.

In [15], Portokalidis et al. implemented a proof-of-concept version of a Perl interpreter that executed Perl scripts with randomized instruction sets. The interpreter's lexical analyzer was modified to append a 9-number tag to each token recognized by the lexical analyzer. Our approach for diversifying Bash follows a similar design: we append a diversifying tag after each recognizable token's semantic value.

In Bash, these tokens can be keywords like `while`, `for`, `if` or more complex constructs like assignments such as `k=1`. As mentioned previously, the diversifying tag depends on the semantic value of these tokens. In Bash, the semantic value of the token `if` is "if" but for example `k=1` is understood as token of the

type `ASSIGNMENT_WORD`, `k=1` being its semantic value. Hence, the string `k=1` receives a different diversifying tag from `k=2` despite both being of the same token type.

The diversification process is shown in Figure 2. The diversification library provides an interface that the Bash interpreter uses during the tokenizing and execution phases. Each hash value is separated from a token with a distinct string of characters. This separator string is used to strip hashes from the input stream before it is passed to the lexical analyzer. For example, with the separator and a hash value, the `echo` command could become

```
echo~~~B2D21E771D9F86865C5EFF193663574DD1796C8F
```

After the lexical analyzer has determined which token it is currently handling, a hash is calculated for that token and compared with the collected hash. If the hashes match, execution is allowed. Otherwise, the diversified token is considered erroneous and execution of the script is halted.

The current implementation uses two different separators for the hashes. The first separator is meant for language specific reserved words and other tokens. The second separator informs the diversification library that the word before the separator should be a command word, that is, built-in utility function, a function call, or a program or script in the `PATH` variable. Before the command gets executed, it is parsed for a hash and it is compared to a hash calculated from the command word. As with other tokens, if the comparison is successful, execution is allowed, otherwise the script execution is halted.

In our proof-of-concept implementation of our diversified Bash interpreter, the hashes are generated using SHA-1. The hash is calculated by concatenating the original token and a token-specific secret. In [15] Portokalidis et al. included the secret in the beginning of the diversified script file or provided it as a command line argument to the interpreter. The secret was omitted from the executable script before parsing. Our solution currently uses the same approach but different methods of storing and handling the secret are quite easy to add.

As an example of script diversification, consider the following script that calculates a few first digits of the Fibonacci sequence:

```
Num=5
f1=1
f2=1
echo "The Fibonacci sequence for the number $Num is : "

for (( i=0;i<=Num;i++ ))
do
    /bin/echo -n "$f1 "
    fn=$((f1+f2))
    f1=$f2
    f2=$fn
done
```

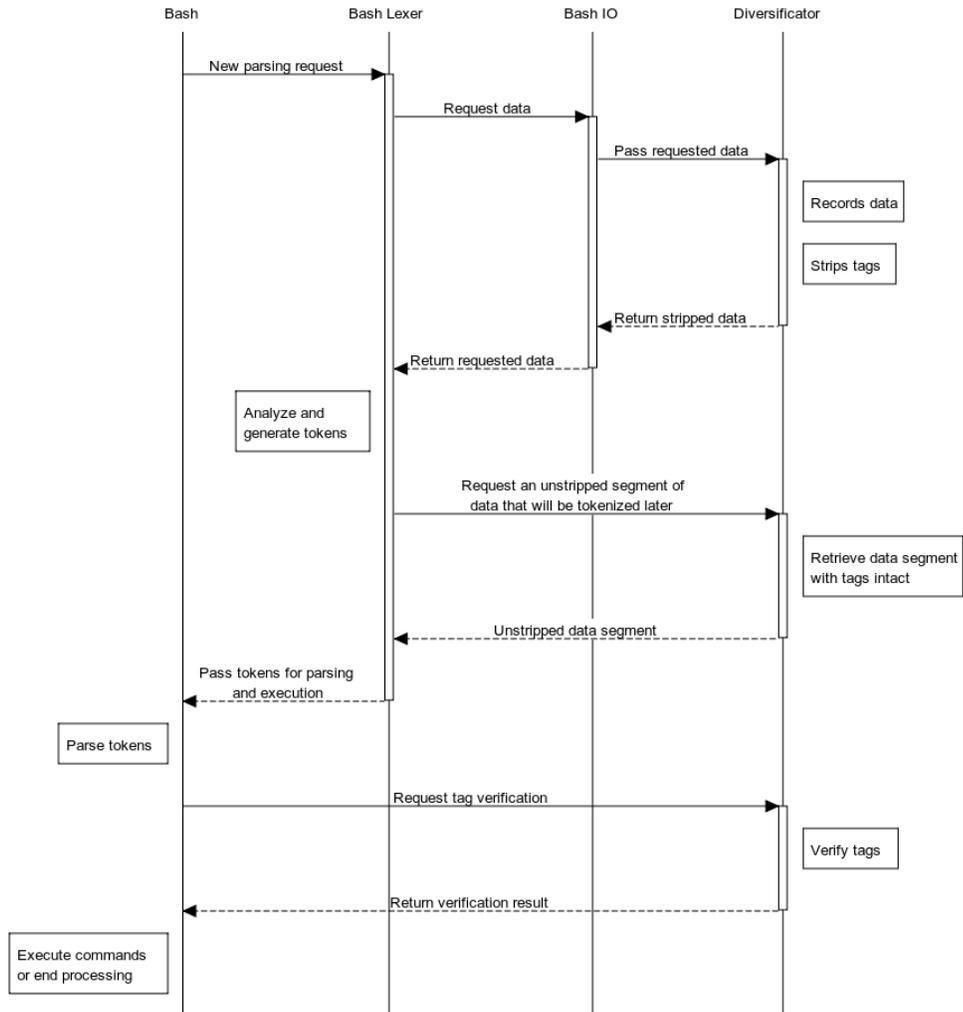


Fig. 2. The diversification process.

The diversified version of the script would look like the following:

```

Num=5^^^9D4D7FB947AFB1BA187FAEFB20533E918EE04212

f1=1^^^D0EE7568D8FE56441EA4BA60CEB119526C12CA06
f2=1^^^BB8630463671DBC49124A08566D6211B5BB90A6B

echo~~~B2D21E771D9F86865C5EFF193663574DD1796C8F
"The Fibonacci sequence for the number $Num is : "

for^^^D9000A6E1DBA2A95B2DDB13E74B220354B5B63AC
(( i=0;i<=Num;i++ ))^^^A04BDD7E8B4AB852FDC07FAF54E0107B12913976
do^^^23CF80A1D6201DAEA7112F6EA161DBA32A055BD2
  /bin/echo~~~BCD981E6B112655886C12639214C366EF6961F03 -n "$f1 "
  fn=$((f1+f2))^A52A61459E705054790329809CA21970B2999E77
  f1=$f2^^^451DBA3B0289063BCA2F6B7319D9F37F944C1BA6
  f2=$fn^^^7ECA3DF4236A6E384DE9ABABD46C4D53BEA2528A
done^^^14D13C75E6A9348DDD5561AD7F1155609175F38A

```

The hashes in this example, generated using SHA-1 function, are rather long and result into a considerable increase in source script file sizes. However, the module responsible for generating and validating the hashes can be easily extended to facilitate alternative methods of hash generation. For the sake of clarity, the hashes are encoded in hexadecimals in the previous example script. The test run performed on this diversified script and other similar examples executed without errors.

The purpose of our diversification library is to provide integrable diversification functionality with minimal changes required to the original interpreter. This would enable a multitude of Bash-like and other interpretable languages to be diversified relatively easily and lessen the burden of maintaining vastly different versions of diversifying script interpreters.

Integrating the diversification library into the existing Bash interpreter required fairly minimal changes to the Bash source code. Most changes were required in `parse.y`, the input file for the Bison parser generator. As mentioned before, the diversification module operates between Bash's I/O handlers and lexical analyzer. As Bash analyses the source code, the diversification module collects recognizable hashes for future comparison. When Bash's lexical analyzer identifies a token, diversification module catches these tokens and calculates hashes for them and compares them with the previously collected hashes. To make sure that code does not get executed before the tokens have been verified, the file `execute_cmd.c` was modified to ask permission from the diversification module to execute the parsed code.

3.3 Further notes on our approach

A big benefit of our approach is that it does not change the software development process. The programmer can write scripts as usual and the diversification of the script is performed by an automatic tool after the code has been written. The user experience is also not affected because the semantics of the scripts remain the same.

Diversification resembles encryption, and one might wonder why we do not encrypt the script files wholly in our scheme. There is a clear benefit in diversifying script languages instead of simply encrypting those script files. When executing an encrypted script, the file first needs to be decrypted. Once this step has been completed, the file is fed to the interpreter. Were an attacker to utilize an attack vector that would bypass the decryption phase entirely, such as a code injection attack, the system would remain vulnerable. In a code injection attack, the malicious code is placed inside the running program or script. This would circumvent the encryption-decryption process. Diversification prevents this scenario by renaming the language interface. Even if malicious code is injected in the running software, it will no longer match the language of the interpreter.

Moreover, unlike with completely encrypted code, with diversified code it is possible to use a renaming scheme in which the original command names are part of the diversified names. This way the code remains easily readable and also maintainable to some extent. The script could also be only partially diversified so that some parts of the code remain open to manual or automatic changes. In any case, it is worth noting that diversification and encryption can be used as separate layers of protection.

Security of our approach could also be further improved with several methods. For example, the original language interface of the Bash command shell could be left in the system as a honeypot that catches malicious programs trying to use it. This is possible because no trusted program should use this original interface anymore. Other way to increase the resilience of our scheme is to make the diversification change dynamically over time. This way, the adversary will have much less time to figure out the diversification that keeps varying.

We also performed preliminary performance tests on our diversified interpreter. The test file consists of 5000 lines of randomized assignment operations. This file was then diversified in order to run experiments with our implementation. While the code example itself is naïve, it requires the diversified interpreter to undiversify each command. This represents the worst case performance scenario for our implementation. Many more complex command structures, such as loops, could be undiversified just once, even though their code is executed several times. The Both files were executed 100 times for both original and diversified Bash interpreter. The times were measured using Bash's built-in time-command. The standard Bash interpreter performed each execution at an average of 0.0164 seconds, while the diversified Bash performed at 0.0443 seconds. Hence, our diversified interpreter takes around 2.7 times longer to execute. Because we have not yet fully optimized our diversified interpreter and because the experiment

was run using the worst case scenario, we do not consider this a large performance penalty.

4 Feasibility of shell diversification

In this section, we present a study of presence of script files in two Linux distributions and discuss some limitations of our diversification scheme.

4.1 A study of presence of script files in two Linux distributions

Our data was collected on Fedora 22 Server distribution and an older, minimal Gentoo distribution. More specifically, on Fedora, the command `uname -a` yields `Linux 4.0.4-301.fc22.x86_64 #1 SMP Thu May 13:10:33 UTC 2015 x86_64 GNU/Linux`. Respectively, Gentoo's `uname -a` is `Linux Gentoo 3.14.4 #1 Tue May 20 11:04:51 EEST 2014 x86_64 GNU/Linux`. During Fedora's installation process a few extra selections were made. The installation type Web Server was chosen and add-ons Tomcat, PHP and MariaDB were added to the installation to provide a touch of real-life server environment.

The process of cataloguing script files was performed using simple tools provided with the installation. First, qualifying files were aggregated using the `find` command and then filtered using a simple `grep` command. All commands were run with root privileges and only script files with execute permissions were searched for. A guard file was created in order to avoid files that are being actively updated. Finally, the `sed` command was used to remove a few pure binary files from the results:

```
# touch guard
# find / -type f -perm /a+x ! -newer guard > files
# xargs grep '^#[/a-z]*/bin/[a-z0-9]*'
  < files > grepmatches 2> /dev/null
# sed -i '/Binary/d' grepmatches
```

The results of this process were processed with a simple Python script. Interpreter paths of the form `#!/usr/bin/env X` were shortened to either `#!/usr/bin/X` or `#!/bin/X` where appropriate. The first column of Tables 1 and 2 shows the interpreter referenced by the script on the shebang (`#!`) line and the second column has the number of such references. Due to the grepping procedure, some files were listed twice. Those files were eliminated in post-processing.

In addition to executable scripts, we also aggregated non-executable library scripts by first listing all files in the system using

```
# find / -type f
```

These files were then filtered using the command

```
# grep grep '\.py[co]*$|\.sh$|\.p[lm]$\$' files > libraryfiles
```

The filtering process relies upon file extensions used by library developers. In Unix-based systems there is no guarantee that file names contain a file extension. However, most well maintained libraries adhere to the convention of using file extensions and thus the numbers give an accurate enough estimation on the quantity of scripts in a fresh system installation.

The data we collected on executable and non-executable scripts were combined using a simple Python script. This script ensured that every file would be calculated only once (having a file extension and a shebang would qualify the file for both executable and non-executable categories). Some libraries have multiple versions of the same file, for example, python library might include `script_file.py`, `script_file.pyo` and `script_file.pyc` where the first file is the source file and the two latter files are byte-code files. In this case `script_file` would only be added to the sum once.

The script files – both executable and non-executable – found in Fedora and Gentoo are shown in Table 1 and Table 2, respectively. Comparing these two tables, we see Fedora has 2319 script files more, but it is also a bit more service-oriented distribution. The biggest difference seems to be in the number of library scripts, Gentoo has more Perl, Python and shell libraries than Fedora. Other than that, the number of scripts is quite similar. The installations themselves are also fairly similar in size (about 80 MiB).

What can be deduced from this data, then? There are quite many script files in both distributions we studied. Still, diversifying them all would not be a huge work for an automated diversifier. It is also worth noting that most of the scripts are Bash or sh scripts that can be handled (sh is a subset of Bash and diversified sh scripts can therefore be run using our implementation). Perl and Python scripts also seem to make up a significant proportion of all the scripts in the system, so covering the interpreters of these script languages would be important for a comprehensive script diversification system. Also, some of the scripts can be rewritten to use a different interpreter to achieve a completely diversified solution.

4.2 Limitations of shell diversification

Although our diversification scheme provides many advantages from the security point of view, it also has some limitations and drawbacks. Obviously, diversifying all scripts in the system introduces a problem for users who want to use the command shell manually. After all, it would be too laborious for the users to write diversified keywords and scripts. We could provide users with a separate terminal for inputting shell commands, but this solution can be a security risk as the malware may find a way to use this interface as well. On the other hand, many normal users are not able or do not need to use the command shell. In some remote systems, the need for an interactive local shell could be replaced by remote administration tools.

Another challenge is the problem of diversifying all the scripts and programs that may dynamically create new scripts at runtime. Still, an automatic diversifier program that programmers can use when adding scripts to their programs

Table 1. Script files found in Fedora.

Interpreter path	Number of files
#!/bin/bash	154
#!/bin/sh	349
#!/usr/bin/bash	3
#!/usr/bin/lua	1
#!/usr/bin/perl	50
#!/usr/bin/python	75
#!/usr/bin/python2	2
#!/usr/bin/python2.7	1
#!/usr/bin/python3	10
#!/usr/bin/python3.4	2
perl libraries	1085
python libraries	3705
shell libraries	31
Total	5469

Table 2. Script files found in Gentoo.

Interpreter path	Number of files
#!/bin/bash	149
#!/bin/csh	1
#!/bin/sh	237
#!/usr/bin/awk	3
#!/usr/bin/jimsh	1
#!/usr/bin/lua	1
#!/usr/bin/perl	107
#!/usr/bin/perl5.16.3	1
#!/usr/bin/python	40
#!/usr/bin/python2	2
#!/usr/bin/python2.7	15
#!/usr/bin/python3	6
#!/usr/bin/python3.3	13
#!/usr/bin/bash	3
perl libraries	1972
python libraries	4991
shell libraries	251
Total	7788

can be created for this purpose. This way, the programmer does not have to manually diversify any scripts that might be included in his or her program code. Also, minimal systems – for instance the operating systems for IoT devices – contain much smaller amounts of script files and scripts included in program code and are thus easier to handle with regard to our approach.

Installing new programs and scripts into the system can also introduce some problems. In order to work correctly, new programs and scripts also need to be diversified using a diversificator tool. In some systems – such as small-scale systems on IoT devices – the issue could be mitigated by preferring image based full-system updates over in-place updates of system files run by scripts. Therefore, at least for minimal and restricted IoT environments, our solution can be expected to work well. In the IoT context, the size of the full system may only be a few megabytes, which makes it quite easy to apply the diversification and fix possible issues.

In instruction set diversification schemes in general, storing the secret diversification key or keys securely is also an issue. In our scheme, however, we assume the attacker does not have an access to the file system of the computer he or she is targeting; this is the case in Shellshock and similar attacks in which the adversary is trying to gain an access to the system. Therefore, for the purposes of this attack scenario the file system can be seen as a safe place to store the secret key. Of course, some stronger cryptographical storing schemes could also be considered.

5 Related work

Instruction set randomization has been applied to several different software layers and many different application areas. Portokalidis et. al present the model closest to our work in [15]. The authors apply instruction set randomization to a Perl interpreter, enabling execution of diversified scripts. Perl code injected by an adversary will fail to run because it is not correctly diversified and is not recognized by the system. They also briefly mention the idea of diversifying shell scripts but do not provide any details or implementation. In this sense, our work can be seen as continuation for their paper.

Boyd and Keromytis have studied the SQL language [4]. Their intermediate proxy, SQLrand, translates the diversified queries into the original SQL language and passes them on to the database. We present an improved scheme and implementation for SQL randomization in [19]. Many papers [3, 15] and books [9, 10] also study the idea of system-wide, global instruction set diversification. Building diverse operating systems and software systems in general has been suggested by Cohen already in the nineties [6]. Forrest [8] also discusses diversified software systems as a security measure.

Barrantes et al. have studied instruction set randomization on binary level to defend against code injection attacks [1, 2]. There has also been some interest in randomizing the system call numbers to render malicious code useless: Jiang et al. [12] and Liang et al. [14] have studied this issue. We have also presented

a tool for system call randomization [16]. Using similar ideas, randomization of memory addresses has been used to prevent memory exploits [7]. The common factor for all of these approaches is that the basic idea is to change the language of the system in order to prevent malicious programs from using some kind of interface that provides access to a resource.

Identifier obfuscation scrambles identifier names on source code level. This is conceptually somewhat similar to our diversification scheme and has been discussed in many papers. For example, there are diversifying tools for Java [20] and JavaScript [11]. We have studied this topic and built a tool that scrambles identifiers and function signatures in web applications written in JavaScript and HTML [17, 18].

6 Conclusions

We presented an instruction set randomization based scheme for preventing code injection attacks in Linux shells. By diversifying the tokens of the Bash scripts uniquely, we prevent the attacker from possessing the knowledge about the correct script language beforehand. We have also discussed the practical implementation for our scheme and explained the effectiveness of our scheme against Linux shell code injection attacks such as Shellshock. We also discussed how our solution improves security over a previous diversification approach.

A study of the presence of script files in two popular Linux distributions was also presented. Based on this, it seems that Perl and Python interpreters should also be covered in a practical and comprehensive script diversification scheme. Therefore, possible future work includes developing our diversificator library to more general direction in order to handle other script languages like Perl and Python.

One limitation of our approach is that all scripts in a system need to be diversified. However, this is quite possible at least in many restricted server environments and small IoT environments with a limited number of scripts and infrequent updates. Also, the diversification could be performed automatically for the most part.

References

1. E.G. Barrantes, D.H. Ackley, S. Forrest, and D. Stefanovic. Randomized Instruction Set Emulation. *ACM Trans. Inf. Syst. Secur.*, 8(1):3–40, 2005.
2. E.G. Barrantes, D.H. Ackley, T.S. Palmer, D. Stefanovic, and D.D. Zovi. Randomized Instruction Set Emulation to Disrupt Binary Code Injection Attacks. In *Proceedings of the 10th ACM Conference on Computer and Communications Security, CCS '03*, pages 281–289, 2003.
3. S.W. Boyd, G.S. Kc, M.E. Locasto, and A.D. Keromytis. On the General Applicability of Instruction-Set Randomization. *IEEE Transactions on Dependable and Secure Computing*, 7(3), 2008.

4. S.W. Boyd and A.D. Keromytis. SQLrand: Preventing SQL Injection Attacks. In *Applied Cryptography and Network Security*, Lecture Notes in Computer Science Volume 3089, pages 292–302, 2004.
5. CloudFlare. Inside Shellshock: How hackers are using it to exploit systems. Available at: <https://blog.cloudflare.com/inside-shellshock/>, 2014.
6. F.B. Cohen. Operating System Protection through Program Evolution. *Comput. Secur.*, 12(6):565–584, 1993.
7. D.C. DuVarney, V.N. Venkatakrishnan, and S. Bhatkar. SELF: A Transparent Security Extension for ELF Binaries. In *Proceedings of New Security Paradigms Workshop*, 2003.
8. S. Forrest, A. Somayaji, and D. Ackley. Building Diverse Computer Systems. In *Proceedings of the 6th Workshop on Hot Topics in Operating Systems (HotOS-VI)*, HOTOS '97, 1997.
9. S. Jajodia, A.K. Ghosh, V.S. Subrahmanian, V. Swarup, C. Wang, and X.S. Wang. *Moving Target Defense II, Advances in Information Security 100*. Springer, 2013.
10. S. Jajodia, A.K. Ghosh, V. Swarup, C. Wang, and X.S. Wang. *Moving Target Defense, Creating Asymmetric Uncertainty for Cyber Threats, Advances in Information Security 54*. Springer, 2011.
11. Q. Jiancheng, B. Zhongying, and B. Yuan. Polymorphic Algorithm of JavaScript Code Protection. In *Proceedings of International Symposium on Computer Science and Computational Technology, ISCCT '08*, pages 451–454, 2008.
12. X. Jiang, H.J. Wang, D. Xu, and Y-M. Wang. RandSys: Thwarting Code Injection Attacks with System Service Interface Randomization. In *IEEE International Symposium on Reliable Distributed Systems, SRDS 2007*, pages 209–218, 2007.
13. G.S. Kc, A.D. Keromytis, and V. Prevelakis. Countering Code-injection Attacks with Instruction-set Randomization. In *Proceedings of the 10th ACM Conference on Computer and Communications Security, CCS '03*, pages 272–280, 2003.
14. Z. Liang, B. Liang, and L. Li. A System Call Randomization Based Method for Countering Code Injection Attacks. In *International Conference on Networks Security, Wireless Communications and Trusted Computing, NSWCTC 2009*, pages 584–587, 2009.
15. G. Portokalidis and A.D. Keromytis. Global ISR: Toward a Comprehensive Defense Against Unauthorized Code Execution. In *Moving Target Defense, Creating Asymmetric Uncertainty for Cyber Threats, Advances in Information Security 54*, 2014.
16. S. Rauti, S. Laurén, S. Hosseinzadeh, J. Mäkelä, S. Hyrynsalmi, and V. Leppänen. Diversification of System Calls in Linux Binaries. In *To be published in proceedings of the 6th International Conference on Trustworthy Systems (InTrust 2014)*, 2014.
17. S. Rauti and V. Leppänen. A Proxy-Like Obfuscator for Web Application Protection. *International Journal on Information Technologies & Security*, 5(1), 2014.
18. S. Rauti and V. Leppänen. Man-in-the-Browser Attacks in Modern Web Browsers. In *Emerging Trends in ICT Security*, 2014.
19. S. Rauti, J. Teuhola, and V. Leppänen. Diversifying SQL to Prevent Injection Attacks. To be published in proceedings of International Conference on Trust, Security and Privacy in Computing and Communications, 2015.
20. T. Zhangyong, C. Xiaojiang, F. Dingyi, and C. Feng. Research on Java Software Protection with the Obfuscation in Identifier Renaming. In *Fourth International Conference on Innovative Computing, Information and Control (ICICIC)*, pages 1067–1071, 2009.

Phishing Knowledge based User Modelling in Software Design

Linfeng Li¹, Timo Nummenmaa², Eleni Berki², Marko Helenius³

¹Beijing Institute of Petrochemical Technology, Beijing, China, 48222692@qq.com

²University of Tampere, Tampere, Finland, {firstname.lastname}@uta.fi

³Tampere University of Technology, Tampere, Finland, marko.t.helenius@tut.fi

Abstract. Due to the limitations of anti-phishing software and limitations in creating such software, we propose the usage of metamodelling frameworks and software tools for implementing software systems where phishing prevention is already designed as a part of the system itself. An expressive computational, verifiable and validatable metamodel is created that captures user behaviour. Next it is shown through examples that the metamodel follows and describes reported phishing scams accurately. The model is then used to create specification in an executable formal specification tool. The formal specification, which can be executed to observe user behaviour, can be used as a building block in the specification of a larger software system, resulting in an inherently phishing-resilient software system design in the form of a formal specification.

Keywords: Phishing, Metamodelling, Formal Methods, Software Design

1 Introduction

Though a variety of anti-phishing technologies have been used against online identity theft (commonly known as phishing), phishers (online identity thieves) had never been discouraged. On the contrary, the phishing attacks become more advanced and more sophisticating [1], utilizing knowledge and expertise from socio-technical and cognitive domains. Technical anti-phishing solutions e.g. static phishing preventions, based on a pre-defined white list or black list [1,2,3] and dynamic anti-phishing appli-

cations follow reactive or preventive maintenance principles [4,5,6] that are not adequate for users' protection. These prevention methods are not adaptable enough to meet users' personal anti-phishing/spam demands and software designers' requirements [1]. To discover the user requirements for anti-phishing software, many user-centered research studies have been carried out, by Zhang et al. [7], Wu et al.[8], Jakobsson and Ratkiewicz [9], Li [1] and other, considering usability and security together. These studies did not, however, result in a reliable, abstract and general model, practical for the designers of anti-phishing software. In this paper, we approach the problem by not creating anti-phishing software, but by creating a metamodel of phishing. This metamodel can help to design software systems using a formal specification system. In so doing, phishing avoidance is already considered in the design of the software. Thus our research question is: *How could user behavior in phishing context be computationally modelled so that the resulted metamodel can be used as a basis for a formal design model of a software system?*

Traditional metamodeling frameworks and tools [10,11] cannot be useful here, because the nature of this research requires to model and simplify the various users' requirements and SW needs, instead of constructing metamodels based on the collected requirements. Hence, in this research work, the authors apply an enriched version of the finite state machine (FSM) model, which is a straightforward and exact computational modelling methodology to describe system processes. The rest of the paper is organized as follows: Related research approaches and studies in anti-phishing and phishing modelling are briefly reviewed. Next, two real phishing scams are presented. An FSM model of user behavior is presented and exemplified through the real phishing cases. The FSM model is then used as the basis for the model of the user in a specification created in the DisCo [12] formal specification language, showing how the FSM can be utilized in software system design. Concluding, the authors summarise the strengths and future potential of this computational metamodeling of user behaviour from the software quality point of view.

2 Research Rationale and Related Work

There are no formal models or metamodels for understanding phishing activities and phishers' and corresponding victims' behaviours online; such model's construction is not a simple task. From the software quality point of view, such a model should be adequately expressive and rich in modeling human activity in detail on the

one hand and sufficiently generic and abstract on the other hand. The latter are needed in order to communicate and test specialization and/or generalization details when needed by and for the various interested groups in the software design of anti-phishing technologies. Moreover, the correctness and consistency of the phishing actions will certainly influence the quality of the resulted anti-phishing prevention software. In any case, the phishing situations are hard for software designers to model, understand and grasp in details unless a modeling solution fits to their cognitive needs. Hence, it is imperative to search for a more abstract, formal point of view and a higher level of modeling for all software development stakeholder groups and especially for those of software users and software designers. Our assumption, which we aim to prove through our ongoing research, is that developers wishing to create phishing-proof software, should employ formal metamodelling techniques to construct dynamic models (like the dynamic nature of phishing) that assist in design.

The following models and notations employ formality and other quality criteria. They are related to but also very different approaches from our work.

Chandrasekaran et al. [13] conducted a pilot project to detect phishing emails based on a set of characteristics of phishing emails. The results showed that the authors' methodology performed with 95% accuracy rate and low false positive rate [13]. However, phishing scams launched from other communication channels, e.g. online social media are still not adequately addressed in the use of this methodology.

Shahriar and Zulkernine [14] introduced a method to model and examine phishing web pages by using FSM theory. By testing the behaviour of given web pages' response, the method brought to them the way to identify phishing pages. However, phishers can easily circumvent it by using embedded objects and malformed web pages, continuing playing games with different roles and versatile game rules.

Akhawe and other [15] presented a formal model of web security based on an abstraction of the web platform and used this model to analyze the security of several sample web mechanisms and applications. Nonetheless, this model is still lacking in considering human factors. For example, how well users understand web communications and the risks of misusing phishing preventions are still questionable.

Kumaraguru et al. [16] attempted to model online trust in their research on trust and security with six components. The aim of the research was to create tools and training modules to help online users make correct decisions about trust.

Dong et al. [17] elaborated on why users form false perceptions and fail to discover mismatches between authentic and phishing web services. They mentioned four rea-

sons: insufficient information, misinterpretation, inaccurate/incomplete expectation and perception ability drop and the users' carelessness when taking the next planned action. Despite of adequate modeling of the user-phishing interactions, this modeling approach is still too general to assist in the design of efficient phishing prevention.

3 Example Reported Phishing Attack and User Behaviors in the Phishing Attack

Most research projects define phishing attacks as cybercrime. This type of crime aims at deceiving victims through fraud web pages in order to steal their personal information. Jacobs describes phishing as a marriage between technology and social engineering [9]. In his definition, the extent of phishing is (i) widened and (ii) not Internet-restricted. Thus more scenarios could be included, where phishers take advantage of other means than the Internet, web browsers or emails communication channels, e.g. surface mail. In order to build a comprehensive and reliable model, the authors collected phishing reports based on the analysis of abused by phishers communication channels and tools. Next, we expose two real-life examples from our data collection.

3.1 Reported phishing attack examples

Attack1: Man-in-the-middle attack

In September 2011 two banks in Finland, Osuuspankki and Nordea experienced a man-in-the-middle attack [18] in which there was no need to install malware to victims' computers. In this attack victims received an email that contained a link to a malicious, but real-looking web site. There was no certificate and the web site address was false, resembling though the original address. If credentials, including a one-time password, were given to the fraudulent site the victim was presented with a fake message, which instructed to wait for two minutes. During that time the server had time to use the credentials and login to the real service for preparing the money transfer. Before this could be accomplished a final one-time verification code was needed. That was asked after the delay and the money could be, thereafter, directly transferred [19].

Attack2: A sophisticated Trojan horse program

There were three banks in Finland attacked: Nordea, Sampo Bank and OP-Pohjola in January 2012. Losses were reported at least from OP-Pohjola. In Nordea e-Bank

money transfers were cancelled because verifications were sent to the victims by SMS about suspicious money transfers [19].

In this case, a sophisticated Trojan horse program (later on called as “Trojan”) was installed on the users’ computers. The Trojan was created by a Zeus toolkit program [20]. There are several possibilities on how a user might have gotten the Trojan installed on his computer:

1. Email that contains the Trojan or link to the Trojan.
2. A malicious or non-malicious source (e.g. web site, memory stick or other media) from which the user installed the Trojan without knowing its true nature.
3. A malicious or non-malicious source (e.g. web site, memory stick or other media) that the user used and vulnerability in the user’s computer was used for infection.
4. Other vulnerabilities that left the user’s computer open to Trojan injection.

It should be noticed that possibilities 1, 3 and 4 have two basic possibilities: either a user does something actively that causes infection or the infection occurs automatically. However, active doing is more probable in possibilities 1 and 3 than in 4.

After the Trojan is installed it activates when a user logs into her/his e-bank. This time a user just needs to pay a bill and at that exact moment the Trojan intercepts the session by changing the bank account and the sum from the transfer. Because a user gives all the credentials authorizing money transfer, the Trojan hiding in the computer has access to the credential information and can change the transfer accordingly. What makes this even more difficult to observe is that the Trojan is able to hijack the browser session and show incorrect account balance.

The authors dare to ask the question: Are the anti-phishing technologies unreasonably ineffective? Why did not we become more secure from all the security technologies that have been deployed against phishing attacks? The authors are also tempted to answer that in terms of metacognitive design of secure software, the right computational metamodels had never been completely realised. As computers and the Internet pervaded all areas of social life, communication and production, computationally and cognitively effective metamodels are increasingly required.

4 Computational Modelling of User Behavior

A finite state machine or finite automaton is a formal, computational model to describe the states in a process and the transition functions among the states. In formal

definition, a FSM consists of 5-tuple elements $(Q, \Sigma, \delta, q_0, F)$, which accordingly represent the state set (Q) , transition alphabet of inputs (Σ) , transition function (δ) , the start state (q_0) , and the set of final (accept) states (F) [10, 11, 21].

The Labelled Transition System Analyser (LTSA) [22] is a verification tool for concurrent systems. It is scriptable and able to generate FSMs based on the scripts. It is also possible to automatically analyze the model and detect errors and deadlocks. In this section, we present our LTSA script, and introduce our model complying with the formal definition of FSM theory.

According to the examples of phishing attacks presented earlier, we have designed the LTSA script in Table 1. In the FSM notation, the model starts from the composing initial state, which means end users' behaviours begin to take place. That is, the user behaviour is to unfold or process according to the state-transition (processing) functions defined in the FSM when a phishing or non-phishing message is composed. When a user receives a message, the user starts processing the message, which means a user needs to either learn from the message or take further actions upon receiving the message.

Table 1. LTSA script to model user behavior in phishing context

COMPOSING = (messageReceived->INFORMATION), INFORMATION = (messageRead->PROCESSING), PROCESSING = (learnFromMessage ->ACQUIRING takeActionUponMessage -> DECISIONMAKING), ACQUIRING = (understandMessage->COMPOSING misunderstandMessage->COMPOSING), KNOWLEDGE = (knowledgeMatchinformation.normalInfo ->BELIEVE knowledgeNotmatchinformation.detectedFraud ->NOTBELIEVE knowledgeNotmatchinformation.checkingMoreinfo ->INFORMATION	knowledgeNotmatchinformation.misled ->DECISIONMAKING), MISKNOWLEDGE = (misknowledgeMatchinformation.undetectedFraud ->BELIEVE misknowledgeNotmatchinformation.obstinacy ->NOTBELIEVE misknowledgeNotmatchinformation. misknowledgeCorrected->DECISIONMAKING), DECISIONMAKING = (informedDecision->KNOWLEDGE uninformedDecision->MISKNOWLEDGE nothingDecided->ACQUIRING), BELIEVE = (decisionMade->COMPOSING), NOTBELIEVE = (decisionMade->COMPOSING).
---	---

When the user learns from the message, the information in the message could be correctly understood or misunderstood. Correct understandings result in the user's advanced knowledge, while any misunderstandings finally end up with mis-knowledge in mind. Both knowledge and misknowledge affect the user's decisions later on. If, in the message, the user needs to take further actions upon the message, s/he will make an informed decision, uninformed decision, or nothing will be decided. As soon as a user makes a final decision on the given message or processes/learns the

information in the received message, the FSM is reset to the initial state to wait for the next piece of composed message either from innocent senders or phishers.

Informed decision represents that the user applies the knowledge to check against the information mentioned in the received message. When the user makes an uninformed decision, misknowledge is used; and 'nothing decided' means there is no knowledge or misknowledge in the mind to take action upon the information in the received message. In the nothing-decided case, the user has to look for more related information from other sources in order to make a proper decision later on.

When knowledge is applied, it does not necessarily mean that the user made a correct final decision. For example, the phishing information in the received message could be so persuasive that the user is misled by the phishing content to make another decision, even though there is an explicit conflict between the user's knowledge and the phishing information. At this point, it is highly likely that the user will be misled and deceived by the phishing information. When the user is determined on the knowledge, the phishing information can be detected, and no fraud can succeed.

When misknowledge is taken into use, the phishing scam may succeed, especially when the phishing information in the received message matches the mis-knowledge in the user's mind. In the FSM model, this transition is called *misknowledgeMatchinformation.undetectedFraud*, and this transition goes to the state named BELIEVE; that is the user believes in the phishing information. When a piece of innocent information showing in the received message mismatches the user's mis-knowledge, the user could either be persistent to stick on the misknowledge or be sensible enough to actively correct the misknowledge and re-make the decision. The transition alphabet is listed in the Table 2 and the state-transition functions and inputs among the states are listed in the Table 3.

Table 2. States of the FSM

States	Explanation	States	Explanation
q ₀	A message is composed	q ₅	Corresponding uninformed decision is made
q ₁	A message is received	q ₆	Not believing the given information
q ₂	Processing information	q ₇	Believing the given information
q ₃	Making decision	q ₈	Corresponding informed decision is made
q ₄	Acquiring information		

Table 3. The state transition and the inputs of the FSM

States transitions	Input	States transitions	Input
->q ₀	A message is composed	q ₈ ->q ₃	knowledgeNotmatchinformation.misled
q ₀ ->q ₁	ReceivedMessage	q ₅ ->q ₆	misknowledgeNotmatchinformation.obstinacy
q ₁ ->q ₂	messageRead	q ₅ ->q ₇	misknowledgeMatchinformation.undetectedFraud

q ₂ >q ₃	takeActionUponMessage	q ₆ >q ₀	decisionMade
q ₂ >q ₄	learnFromMessage	q ₇ >q ₀	decisionMade
q ₃ >q ₄	nothingDecided	q ₈ >q ₇	knowledgeMatchinformation.normalInfo
q ₃ >q ₅	uninformedDecision	q ₈ >q ₆	knowledgeNotmatchinformation.detectedFraud
q ₃ >q ₈	informedDecision	q ₈ >q ₁	knowledgeNotmatchinformation.checkingMoreinfo
q ₄ >q ₀	understandMessage, misunderstandMessage	q ₅ >q ₃	misknowledgeNotmatchinformation .misknowledgeCorrected

In order to verify the FSM model that we elaborated earlier, we explain the phishing example in the FSM. Taking advantage of the animation feature in LTSA, we were able to produce a sequence of interactions in each phishing context. Examples of these traces are presented next.

Attack1: When a victim receives an email containing a link to a malicious, but real-looking web site, and decides to give credentials to the phishing web site (Verification 1-1):

*messageReceived ->messageRead ->takeActionUponMessage->
uninformedDecision-> misknowledgeMatchinformation.undetectedFraud
->decisionMade*

In this case, the misknowledge is that the given link in the email is mis-understood as an authentic and trustworthy web site. Therefore, the victim decides to hand out the credentials to the web site.

When a victim receives an email containing a link to a malicious real-looking web site, the victim may suspect the authenticity of the visited web site (Verification 1-2):

*messageReceived -> messageRead ->takeActionUponMessage->informedDecision
->knowledgeNotmatchinformation.checkingMoreinfo*

In this case, the victim does not use the same misknowledge as the one above. Instead, the victim checks more information to verify the authenticity of the displayed web page. If verifying the authenticity of the web page, the possible victim's behavior could be e.g. acquiring new information from various sources.

Attack 2: The four possibilities of users' behaviors to get Trojan horse program installed could be abstracted into two types, installation with users' awareness and permission and installation with no users' awareness. The two types of installation decisions could be described as below:

1. Installation with users' awareness and permission (Verification 2-1):

*messageReceived -> messageRead -> takeActionUponMessage
->uninformedDecision ->misknowledgeMatchinformation.undetectedFraud
-> decisionMade*

2. Installation with no users' awareness (Verification 2-2):

messageReceived -> messageRead -> takeActionUponMessage

-> nothingDecided -> misunderstandMessage

After the Trojan horse program is installed, the following user behaviors are committed under the mis-understanding that the communication traffic is secure and no one else has the access to the credentials or monitor the traffics. In this case, the victim's behavior could be explained like this (Verification 2-3):

messageReceived -> messageRead -> takeActionUponMessage

->uninformedDecision ->misknowledgeMatchinformation.undetectedFraud

->decisionMade

5 Modeling Software: Usage of the FSM Model

The presented FSM model depicts the behavior of a user in a phishing situation. This kind of knowledge can be utilized in various ways, as long as it is understood. The FSM model developed with LTSA is aimed to be easy to understand and use because of its simplicity, focus on state transitions instead of states and the possibility for animating those transitions to help in understanding the model.

To demonstrate how the FSM model can be used in software development, the authors created an executable specification of dynamic users in a dynamic environment in the DisCo language. DisCo is an action based executable formal specification software package for modelling reactive and distributed systems [23]. It has been updated to include support for probabilistic modelling [24,25,26]. Models created with the system can be instantiated with different system states and can be animated in a graphical animator. The goal of specifications created with the system is that they can later be implemented as software systems.

While DisCo is a modelling system itself (as LTSA is), it is also different from LTSA because DisCo models stay manageable in larger sizes than LTSA models. DisCo models also facilitate larger system views and not just state transitions. On the other hand, DisCo does not support similar automatic analysis of models as LTSA does. Because of the way the LTSA model has been written in the LTSA language, that is to focus on state transitions and not actual states, the specification can easily be converted into an action based DisCo specification. In fact, the conversion could even, at least partially, be done automatically. The converted information can be used as part of a larger specification.

Models created with DisCo can support multiple independent objects that can all have states of their own. Thus, it is possible to model larger multi-user phishing at-

tacks and other scenarios that resemble actual real world scenarios. DisCo can be used to analyze how objects following the state transitions given in an LTSA model act in those scenarios.

The model depicts a world view where several persons and information sources exist. Each person acts based on the user behavior model in our FSM, but their decisions at the points where multiple choices are possible, are influenced by both a knowledge level variable and a misknowledge variable that are updated when the persons learn correct or incorrect information. Each information source has a different value for how influential it is, which results in users earning different knowledge and misknowledge depending on the information sources they access.

The specification is implemented by specifying classes and actions. The classes of the specification are presented in Table 4 and the actions are presented in Table 5. The classes are instantiated to be objects in the specification system. The actions are largely based on transitions in the LTSA model and are what alters the state of the system. Each action has a guard that determines whether the action is enabled and can be executed. The guard is a boolean expression that must evaluate true for the action to be enabled. Because DisCo is an action-based system, objects take part in actions but the action chooses its participants through the guard expression.

Table 4. DisCo specification classes

Class	Explanation
InformationSource	A source of information for a person.
Person	A person who can be the target of phishing.

Table 5. DisCo Specification actions

Action	Explanation	Action	Explanation
receiveMessage	A source of information for a person.	takeAction	A person starts to take action.
learn	A person starts learning.	canNotDecide	A person cannot make a decision.
understand	A person understands and its knowledge value is increased.	Misunderstand	A person understands and its misknowledge value is increased.
makeUninformed Decision	A person starts making an uninformed decision.	makeInformed Decision	A person starts making an informed decision.
checkMoreInfo	A person starts reading more information.	Correct-Misknowledge	A person starts taking action and its misknowledge is lowered.
mislead	A person is misled and starts taking action.	detectFraudDecision	A person detects a fraud decision.
normalInfo-Decision	A person makes a decision.	undetectedFraudDecision	A person makes a decision but cannot detect fraud.
obstinacyDecision	Makes a stubborn decision.	updateSkip-Values	Update values that dictate the probability of a person understanding, misunderstanding, and making informed and uninformed decisions

An example action can be seen in Table 6. The class *Person* (Table 7) is a representation of a person, who can access information that might include phishing mes-

sages. *Person* has a state variable, which is based on the states in the LTSA model. The state variable is used to control the state of each individual *Person* instance. The person has a variable for mis-knowledge and knowledge, so it is possible to control how knowledgeable that *Person* object is. These values are changed through certain actions such as ‘mis-understand’. *Person* also has skip variables which enable instances of *Person* to decline taking part in actions. For example, an instance of *Person* may decline to take part in ‘understand’ if that instance’s misknowledge is much higher than knowledge. The *informationSource* class (Table 8), is a generic information source that a person may access. It only has one variable, *informationValue*, which means how influential the specific source of information is.

To test the DisCo model, a creation was instantiated. The creation is a view of the state of a system in DisCo and is instantiated based on the specified classes. In this case, two instances of *Person* were instantiated (*Person_1* and *Person_2*) along with two instances of *informationSource* (*InformationSource_1* and *InformationSource_2*). *Person_1* started with knowledge and misknowledge values of 60, while *Person_2* started with both values at 30. The *informationValue* of *InformationSource_1* is 20, while the *informationValue* of *InformationSource_2* is 10.

Table 6. receiveMessage action

```
action receiveMessage (p : Person; isource: InformationSource) is
when (p.updateSkips = false and p.personState='idle')
do
    p.personState->reading()||
    p.currentIs:=isource; end;
```

Table 8. InformationSource class

```
class InformationSource is
informationValue : integer;end;
```

Table 7. Person class

```
class Person is personState :
(idle,reading,read,learning,taking_action,indecisive,
making_informed_decision,making_uninformed_decision);
skipPossibilitymisunderstand : integer;
skipPossibilityunderstand : integer;
skipPossibilitymakeUninformedDecision : integer;
skipPossibilitymakeInformedDecision : integer;
skipPossibilitycanNotDecide : integer;
updateSkips : boolean; knowledge : integer;
misknowledge : integer;
currentIs : reference InformationSource; end;
```

There were 100 execution steps, resulting in a state where the knowledge of *Person_1* was 130 while mis-knowledge was 90. The knowledge of *Person_2* was 150 and misknowledge 10. In this execution, *Person_2* had become more knowledgeable and shed off some misknowledge, while *Person_1* still had a lot of misknowledge, even while learning new knowledge.

Table 9 shows a trace of the first 20 actions that were executed. Traces of executed actions can be used to analyze behavior within a scenario. Table 9 divides each action into the name of the action and the participant objects that took part in the action to give a clear picture of what has happened in each step. Among the things we can dis-

cover from this trace, is when a Person is part of an undesirable action and why that happened.

Table 9. Execution trace of the first 20 actions executed

Step	Action	Participants	Step	Action	Participants
1	receiveMessage	Person_2, InformationSource_1	11	makeInformedDecision	Person_2
2	takeAction	Person_2	12	detectFraudDecision	Person_2
3	makeUninformedDecision	Person_2	13	receiveMessage	Person_2, InformationSource_2
4	undetectedFraudDecision	Person_2	14	takeAction	Person_2
5	receiveMessage	Person_1, InformationSource_2	15	makeUninformedDecision	Person_2
6	takeAction	Person_1	16	receiveMessage	Person_1, InformationSource_1
7	makeUninformedDecision	Person_1	17	obstinacyDecision	Person_2
8	receiveMessage	Person_2, InformationSource_2	18	receiveMessage	Person_2, InformationSource_1
9	undetectedFraudDecision	Person_1	19	takeAction	Person_1
10	takeAction	Person_2	20	takeAction	Person_2

We can see that in step 4, Person_2 was a participant in the *undetectedFraudDecision* action which is considered the worst case scenario in which a phishing attempt is successful. Person_2 became a part of that action by first receiving a message from InformationSource_1, then by taking action and making an uninformed decision. Person_1 also takes part in the *undetectedFraudDecision* action in step 9.

As DisCo is built to support incremental development of specifications through a layer system, the presented model can easily be used as a building block of a software system. A specification created in this way would enable the designer to verify that the software design takes into account how a user behaves in phishing situations. Also, as a DisCo specification can include outside influences, phishers could be modeled separately as classes in a specification to gain a better understanding of possible phishing situations regarding the system in design.

6 Conclusions and Future Research

In contrast to specific reactive anti-phishing solutions, the method proposed in this paper focuses on the analysis and modelling of user behaviour to facilitate the proactive design of phishing-resistant systems. First, user behaviour in a phishing context is modelled with FSM notation. The FSM-based model provides a general and abstract view to describe user behavioural patterns in phishing context. This model can be a helpful guide to evaluate the design quality of anti-phishing and phishing-resistant systems at early design stages. The model can be verified, both with automatic meth-

ods, and through real cases testing with it. The FSM model can be used as a basis for a specification of a full software system using a tool such as DisCo. That model can take into account important security aspects found in the abstract FSM model. Our method is a series of transitions, first transitioning from the FSM model, to a DisCo implementation, then to a more refined DisCo implementation of a software system through a layer system, and finally to a software implementation. The refined DisCo implementation and software implementation are left as future research.

Rich and multipurposed, testable, computational and easily implementable meta-models that are expressible and extendable (yet, by no means complete) can assist: i) software designers and developers to realize the dynamic design of security auditing systems by considering a great amount of static and dynamic situations; ii) towards an improved understanding and empowerment with knowledge of the digitally competent person to protect his/her own personal data and take appropriate security decisions; iii) software sponsors to decide on cost-effectiveness by checking and testing for security in early design steps. This research concentrates on improving security through preventive, and not corrective or adaptive, systems maintenance. For this reason cost-effectiveness is a quality property that can also be achieved with our suggested phishing prevention solutions.

Although the advantages of this metacognitive and computational FSM-based metamodel demonstrate that the model is an efficient analytical tool with great design and implementation potential, it still requires certain efforts on metamodelling. Arguably, limitations may hinder the usage of the enriched FSM metamodel, but this necessitates the investment of efforts and resources on model-driven testing and corresponding automated tools at the early stages of the anti-phishing software lifecycle.

Naturally, general considerations that will be tackled in future research are: i) the scalability of the approach, since phishing typically affects large groups of people. DisCo simulations can simulate groups of people, but it really should not be important to model a million individuals in an abstract simulation; the focus is elsewhere. ii) The FSM described in the paper consists of very general transitions. Nonetheless the knowledge we hope to obtain from analysing its usage is that we can see that the transitions conform to the reality and then we can integrate those transitions to a realistic SW design. We can, then, analyse that design which includes the transitions. iii) In the case of phishing, and due to its dynamic (not static) nature, unavoidably people become more or less informed over time; there should always be ways to distinguish

and model between the user making an informed or an uninformed decision. However, this is, by and large, a socio-cognitive issue.

References

1. Li, L.: A Contingency Framework to Assure the User-Centered Quality and to Support the Design of Anti-Phishing Software, PhD. Dissertation, University of Tampere (2013).
2. SpamAssassin: The Apache SpamAssassin Project, <http://spamassassin.apache.org/>, retrieved on 12th Aug. 2015.
3. Sun, B., Wen, Q., Liang, X.: A DNS Based Anti-phishing Approach. In: 2010 Second International Conference on Networks Security Wireless Communications and Trusted Computing (NSWCTC), pp. 262-265 (2010).
4. Atighetchi, M., Pal, P.: Attribute-Based Prevention of Phishing Attacks. In: Eighth IEEE International Symposium on Network Computing and Applications, pp. 266-269 (2009).
5. GMAIL: How is spam handled, <http://support.google.com/mail/bin/answer.py?hl=en&answer=78759>, retrieved on 11th Dec., 2014
6. SpamAssassin: DNS Blocklists, <http://wiki.apache.org/spamassassin/DnsBlocklists>, retrieved on 2nd May 2015.
7. Zhang, Y., Egelman, S., Cranor, L., Hong, J.: Phinding Phish: Evaluating Anti-Phishing Tools. In: the 14th Annual Network & Distributed System Security Symposium, San Diego, CA, USA (2007).
8. Wu, M., Miller, R. C., Garfinkel, S. L.: Do Security Toolbars Actually Prevent Phishing Attacks? In Proceedings of Conference on Human Factors in Computing Systems (2006).
9. Jakobsson, M., Ratkiewicz, J.: Designing Ethical Phishing Experiments: A study of (ROT13) rOnl auction query features. In: the 15th annual World Wide Web Conference, pp. 513-522 (2006).
10. Berki, E.: Establishing a Scientific Discipline for Capturing the Entropy of Systems Process Models. CDM-FILTERS. A Computational and Dynamic Metamodel as a Flexible and Integrated Language for Testing, Expression and Re-engineering of Systems. Ph.D. Thesis. Faculty of Science, Computing and Engineering. University of North London (2001).

11. Eilenberg, S.: Automata, Languages and Machines. Academic Press. (1974).
12. DisCo: DisCo Home Page, <http://disco.cs.tut.fi/>, retrieved on 22nd Dec., 2014.
13. Chandrasekaran, M., Narayanan, K., Upadhyaya, S.: Phishing email detection based on structural properties, In: the NYS Cyber Security Conference (2006).
14. Shahriar, H., Zulkernine, M.: PhishTester: Automatic Testing of Phishing Attacks, In: 2010 Fourth international Conference on Secure Software Integration and Reliability Improvement, pp. 198-207 (2010).
15. Akhawe, D., Barth, A., Lam, P. E., Mitchell, J., Song D.: Towards a Formal Foundation of Web Security. In: 23rd IEEE Computer Security Foundations Symposium, pp. 290-304 (2010).
16. Kumaraguru, P., Acquisti, A., Cranor, L.: Trust modeling for online transactions: A phishing scenario. In: Privacy Security Trust, Ontario, Canada (2006).
17. Dong, X., Clark, J. A., Jacob, J.: Modelling user-phishing interaction. In: Human-System Interaction, Kraków, Poland (2008).
18. F-Secure: Man-in-the-Middle Attacks on Multiple Finnish Banks, <http://www.f-secure.com/weblog/archives/00002235.html>, retrieved on 30.1.2015.
19. YLE: Finnish bank accounts breached, http://www.yle.fi/uutiset/news/2012/01/finnish_bank_accounts_breached_3191383.html, retrieved on 10.4. 2015.
20. Wyke, J.: What is Zeus, technical paper, <http://www.sophos.com/en-us/why-sophos/our-people/technical-papers/what-is-zeus.aspx>, retrieved on 30.4.2015.
21. Sipser, M.: Introduction to the Theory of Computation, PWS Publishing Company (1997).
22. LTSA: Labelled Transition System Analyser, <http://www.doc.ic.ac.uk/ltsa/>, retrieved on 10th November, 2015.
23. Aaltonen, T., Katara, M., Pitkänen, R.: DisCo toolset - the new generation. Journal of Universal Computer Science, 7, 1, 3–18 (2001).
24. Nummenmaa, T.: A method for modeling probabilistic object behaviour for simulations of formal specifications. In: NWPT 2008, Abstracts, Tallinn, Estonia, (2008).
25. Nummenmaa, T.: Adding probabilistic modeling to executable formal DisCo specifications with applications in strategy modeling in multiplayer game design. Master's thesis, University of Tampere (2008).
26. Nummenmaa T.: Executable formal specifications in game development: Design, validation and evolution. PhD thesis, University of Tampere (2013)..

Securing Scrum for VAHTI

Kalle Rindell, Sami Hyrynsalmi, Ville Leppänen

University of Turku, Department of Information Technology, Finland,
kakrind@utu.fi, sthyry@utu.fi, ville.leppanen@utu.fi

Abstract. Software security is a combination of security methods, techniques and tools, aiming to promote data confidentiality, integrity, usability, availability and privacy. In order to achieve concrete and measurable levels of software security, several international, national and industry-level regulations have been established. Finnish governmental security standard collection, VAHTI, is one of the most extensive example of these standards. This paper presents a selection of methods, tools, techniques and modifications to Scrum software development method to achieve the levels of security compliant with VAHTI instructions for software development. These comprise of security-specific modifications and additions to Scrum roles, modifications to sprints, and inclusion of special hardening sprints and spikes to implement the security items in the product backlog. Security requirements are transformed to security stories, abuse cases and other security-related tasks. Definition of done regarding the VAHTI requirements on is established and the steps to achieve it are described.

Keywords: Scrum, agile, VAHTI, software security, security standards

1 Introduction

Software industry has always been under scrutiny by regulators, standardization organizations and a plethora of industry-specific and more or less general standards. The purpose of all this has been to guarantee a certain level of evidence about certain quality aspects or functionality of the software. Consequently, also software security is becoming increasingly regulated. Several security standards and audit criteria have been established, and even more academic and commercial software development methods have been suggested to meet the standards. A number of the development methods and best practices have achieved a standardized status themselves. Main issue with the standardized methods is that while they provide the necessary *security assurance* required for certain regulated environments and security audits, they in general do not adhere to the agile values or ideology.

The term ‘agile’ reflects approaching the development of software using new focus, ideology and values [3]. As opposed to the sequential waterfall model, which attempts to maximize the efficiency by not allowing any change, and by locking down the output of each development phase, the agile mindset anticipates change, and even welcomes it. At the core of agile software development lies Scrum, one of the oldest, and the most established and widely-used development approaches among agile methods [19].

The scopes of security standards can be divided into industry specific, national and international regulations. Standardization types include standards for software *safety* and

privacy, control and management of *data, software, assets, personnel* and *processes*, and any other aspects of information security. Consequently, these regulations also concern the design and development of these aspects. The main subject of this study, software development security, can further be divided into its components: security of the used technologies, security of used processes and methods, and, consequently, the security of the produced software. To control the security of its information systems, the Finnish government has published its own public security requirements, VAHTI instructions [10]. VAHTI is the *de facto* standard for the governmental information systems' software maintenance, use and development. VAHTI instructions for application development, published in 2013, include definite requirements regarding the development of the software and inherently the used development method itself [9].

VAHTI, however, is not directly compatible with the mainstream agile software development methods. Therefore, this constructive paper presents modifications to Scrum that are needed to complete with the governmental requirements. We have identified the VAHTI requirements regarding the development method and the development phase, and outline the necessary mechanisms and measures to be instantiated and integrated into Scrum necessary to meet these requirements. These means include security-related roles, processes and techniques, done at the three security levels defined in VAHTI: basic, heightened and high. The security modifications to Scrum are selected by selecting applicable security measures from international standards and established security frameworks and methodologies. While Scrum does not define security roles or processes, we analyse the VAHTI requirements and translate these to concrete software development concepts to be applied to Scrum.

This article consists of following sections: in Section 2, we discuss the background and motivation of this article, as well as cover the related work done in the field of introducing software security mechanisms in agile methods. In Section 3, we present the Scrum method, its key terminology and how a software project is conducted using Scrum. Section 4 introduces the VAHTI instructions for Application Development, the process and reasoning in selecting the key requirements from VAHTI, which are then grouped by their security level and assigned to Scrum roles. In Section 5, we provide the modified Scrum process to meet the VAHTI requirements. The security compliance and security assurance is achieved by importing specific elements of established security frameworks into Scrum. An example project structure is also provided. Finally, in Section 6, we conclude the results and discuss avenues for future research.

2 Motivation, Research Questions and Related Work

Security in its various aspects has been a key topic in information technology since the first computers. Perhaps it is the military background of the development of computing devices, networks and the Internet which still molds the security procedures, aiming to be well-structured, thoroughly documented and strictly regulated — even to the point of being rigid and a hindrance to production. On the industry side, finance and banking were among the earliest to introduce computers to their business processes, and financial data is among the most conspicuously protected information assets in any organization. With the emergence of personal computing and near-ubiquitous Internet services, privacy

and identity protection issues have gained importance among security topics. On the other end, security also concerns countries and governmental entities. These typically have a special focus on the *continuity* of services, especially ones critical to running the society. When comparing the compliance requirement the various security standards and regulations are setting to the ideology presented in the agile manifesto, the task of providing security assurance and complying with standards appears to be a non-agile, or even an anti-agile task. Even the VAHTI instructions for application development state that the ‘mandatory’ and ‘well-documented’ S-SDLC (Secure Software Development Life-Cycle) should ‘*define exit criteria for different development phases*’, a clear presumption that the waterfall model will be used. Against this background the key questions in our research were:

- How to make an S-SDLC conforming with the agile mindset as much as possible?
- Selecting Scrum, the most widely-used software development method, as the reference method; how should it be modified to comply with VAHTI?
- If a requirement stated in VAHTI is considered an item in the backlog, what is the definition of done for each of these items?
- How can the definition of done be achieved with the least security overhead?

To address these questions, we used a Conceptual-analytical research approach and a constructive research strategy [13]. That is, in this study we analysed the integral elements of two existing tools (i.e., Scrum and VAHTI). Based on the results of this conceptual analysis, we present the needed modifications to Scrum that it will fulfill the requirements set by VAHTI. This study presents only the result of conceptual analysis and further works are needed to verify and validate the proposed model.

In extant literature, several secure software development methods have been introduced. Some of these even claim to be agile, or at least use an agile method such as Scrum or eXtreme Programming (XP) as a starting point. Fitzgerald et al. [8] have applied Microsoft Security Development Lifecycle (SDL) [14] to Scrum and created their own version of the methodology to meet organization-specific security requirements. In recent work done at the University of Oulu, Vähä-Sipilä, Ylimannela and Helenius (in [15]) have gathered various additions and modifications to agile programming methodologies, such as an initial ‘sprint zero’ for security definition purposes, use of security stories for communicating requirements, abuse cases for testing, hardening sprints and overall modifications to Scrum’s basic structure, identifying control points crucial for security activities. Additionally, to handle the measurably increased complexity the security requirements add to the software and the processes, even a change of paradigm to aspect-oriented programming (AOP) has been suggested [5]. Typically the suggested S-SDLC methods were instantiated only for a single project within a single company, if instantiated at all.

Implementation of the security controls and tasks is a more clear-cut field: Microsoft suggests its SDL, and even in its current published version 5.2 provides ‘SDL for Agile’ extension to their method. SDL may be considered a security framework offering a full set of tools, methods and techniques to implement the security tasks they suggest. Although not directly security-oriented, the Capability Maturity Model Integration for Development (CMMI-DEV) was studied by e.g. Diaz et al. [7] regarding the relationship

of heavy software processes of CMMI's managed level 2 processes [2] to Scrum. Similarly, our approach was to examine industry standards and best practices, such as SSE-CMM [12], BSIMM-V [4] and Microsoft SDL, and modify Scrum with a minimum set of items that are necessary for VAHTI compliance. In our approach, the processes are kept as lean as possible by not striving for formal 'maturity levels', and by minimizing any security overhead deemed unnecessary.

3 Scrum

Scrum is an iterative and incremental project management framework, based on the principle that customer requirements and other agents may change during the project's execution, and by working in iterations, allows the team to react to the change. Scrum gives the organization quite a lot of freedom in how to organize and execute the project, but certain key roles and concepts are defined. This section's introduction to Scrum is based on the Scrum Primer book [6].

Scrum is designed promote productivity and mitigate management and governance overhead, by e.g. giving the developers as much freedom as possible in defining and implementing their tasks (self-organizing teams) and all but eliminating the role of the traditional project manager. The project may still require some project and product management functions such as financial or other reporting, but the Scrum project does not have or need a dedicated project manager. Scrum has an emphasis on intra-team communication, favoring team co-location or at least tight online collaboration.

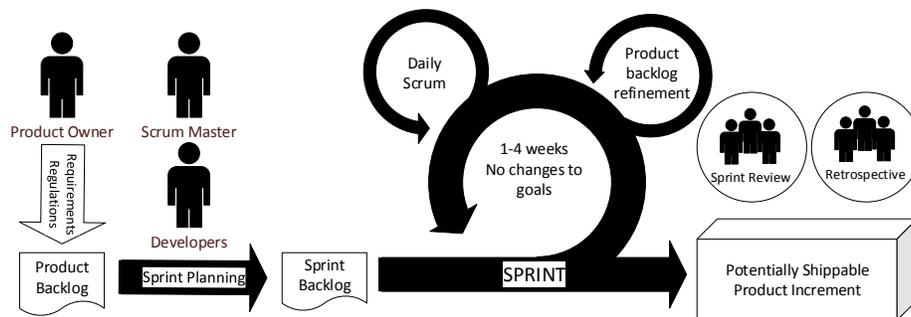


Fig. 1. The Scrum process (adapted from [6])

Figure 1 shows an overview of the Scrum process. This is a slightly modified version of 'pure' Scrum, allowing redefining the product backlog during the sprints. The key concepts of the methods are:

1. The *Team* consists of three core roles: *Product Owner* (PO), representing the customer and stakeholders, *Scrum Master*, facilitating for the team and removing any impediments, and (typically) 3-9 *Developers*, who as a cross-functional and self-organizing team utilize their skills to create *Potentially Shippable Product Increments*.

2. *Stories* are product requirements often written from the product owner's perspective.
3. *Product Backlog* is the list of stories, use cases, requirements and other items that require completion in order to deliver the product.
4. *Tasks* and *Sub-tasks* are concrete steps which the team members create and complete based on the backlog items.
5. In *Sprint Planning*, the team selects from the product backlog items that can (and should) be completed within the next sprint. The prioritisation of the items comes ultimately from the stakeholders, i.e., the customer. In Sprint Planning, the items are converted into completable tasks and sub-tasks, which added to the *Sprint Backlog*.
6. *Sprints* are the iterations in which the team completes items (i.e., tasks) in the sprint backlog within a pre-scheduled time box (typically 7-28 days). During sprints the product backlog items may be refined, added or deleted, while the sprint backlog remains as unchanged as feasible.
7. In *Daily Scrum* meetings the team members brief each other in what they did yesterday to complete the product, what they plan to do today, and whether there are any impediments to the work.
8. The *Definition of Done* is a set of consistent criteria to determine when an item in the product backlog is considered 'ready', typically after regression testing. Determined by the Scrum master with input from the stakeholders through the product owner.
9. *Sprint Review* is held at the end of each sprint. In this event, the team reviews the completed work, and also the incomplete items on the sprint backlog.
10. *Sprint Retrospective* is for the team to review the sprint itself and the work done in it. It aims for continual improvement of the process, and allows for reflection on what was done well, and what needs improvement.

As any software project, a Scrum project starts with pre-planning and requirement gathering. The requirements are either functional or non-functional, and may come in as user stories, regulations, or other requested features that must be implemented in the product. These items are added into the product backlog. One key point in Scrum is to 'deliver value to business' as effectively as possible. In order to do so, the product owner, as customer's voice, prioritizes the items that are selected into each sprint's backlog at sprint planning event. This way it is at least theoretically possible, that after a few sprints (or even one) the 'potentially shippable product increment' can already be utilized despite some less important features being still under development. This way the value can be realized earlier and, as a bonus, user feedback and bug reports can be utilized to make the end product better. In each sprint, the developers pull items into their own work flow from the sprint backlog, entitling the term 'self-organizing team'.

In addition to sprints, research and prototype work may be done in 'spikes'. Similar to sprints, spikes are time-boxed efforts to produce something that contributes towards a completion of a complex backlog item or work as a proof of concept, without necessarily aiming to complete the item and delivering shippable features.

During sprints, environmental and requirement changes can be taken flexibly into account, in which case the product backlog is adjusted accordingly – typically the sprint backlog remains unchanged. After each sprint the results of the work (the potentially shippable product increment) is evaluated, the definition of done for the product backlog items is verified. The definition of done is an important concept in Scrum, as it is the only way items can be removed from the backlog.

For measuring the progress of the project, agile methods utilize a number of techniques. Probably the single most important technique is *story points*: product owner prioritizes some items in the product backlog over others due to their business value, and the developers evaluate them by the estimated development effort. The story points are not directly translatable to work hours. To emphasize this, a non-linear scale such as Fibonacci-like sequence is used. Sometimes the work estimates include an option to declare items too complex to implement in the current sprint. Thus the story points represent the team's view of the required effort, subjective to their current skill set.

Scrum is an *empiric* method: it readily admits and submits to the fact that not everything is known or understood, and that things will change. In security engineering, *defined* methods would be preferred: a process is started and allowed to complete, producing invariably the same results each time [20]. The waterfall model aims to this goal, but by being excessively rigid, it introduces a very concrete risk of failure to meet the stakeholder's and environment's changing requirements. Traditional old school security engineers tend to scold the agile methods' iterative approach as 'trial-and-error'. Despite this lack of acceptance, Scrum is far from unstructured: it just gives the team more freedom in deciding the order in which the items are implemented, and possibility to iterate on the initial requirements, much based on assumptions. At certain point the definition of done criteria will be met, security compliance achieved and security assurance provided. In the process of doing so, the quality of the produced software may actually be higher than those made using sequential methods [17].

4 VAHTI instructions for application development

VAHTI instructions is a wide collection of governmental security regulations, published by Finnish Governmental Steering Group for Information Security (Valtionhallinnon tietoturvallisuuden johtoryhmä, VAHTI). First publications in the VAHTI collection are dated 2001, and they exist to support the data security legislation and Finland's strategies for National Knowledge or Information Society¹ and Information Security². The instructions for application development were published in 2013. Compliance with VAHTI has been mandatory for state agencies, partners and suppliers since 2014.

VAHTI requirements for software development [9] consist of 120 individual requirements, divided into 15 categories. The categories span the whole life cycle of the application or information system, ranging from strategy and resourcing to the eventual end of life and ramp-down of the system. Of this requirement set, only the ones most relevant to the development process were selected for this study. The candidates for inclusion were directly involved with either the tasks during development process, such as security design, audits or reviews; candidates for exclusion considered organizational issues, strategies, policies, method-independent techniques, IT environment, or issues related to the post-development phases of the application's life cycle such as continuity management or system ramp-down. The result set comprises of 23 requirements considered to affect the development method directly. Table 1 shows the final selected VAHTI

¹ http://www.tietoyhteiskuntaohjelma.fi/esittely/en_GB/introduction/index.html

² http://www.lvm.fi/c/document_library/get_file?folderId=339549&name=DLFE-10210.pdf&title=Julkaisu%2051-2009

requirement grouped by the security level. For each requirement, the expected frequency of the task is projected, and the role(s) responsible or affected by the task are displayed. The only requirement directly concerning the development method, SKM-001, states that the development process itself is required to be a Secure Software Development Life Cycle process. The method is to be documented, development personnel trained in its use, utilized at all times, and comply with all the security requirements.

Table 1. VAHTI requirements for development method per security level

Code	Requirement name	Frequency	Dev	SM	PO
OSK-001	Security Training	1	x	x	
OSK-008	Additional security training after change	0 or 1	x	x	
VTM-005	Application Risk Analysis	1 or more	x	x	x
TST-002	Test Plan Review	1	x		
VTM-008	Threat Modeling - recommended	1	x	x	
VTM-010	Threat Modeling updates - recommended	1 or more	x	x	
ESI-001	Goal and Criticality Definition	1	x	x	x
ESI-002	Business Impact Analysis	1			x
VTM-001	Documentation of Security Solutions	1	x		
VTM-006	App. Security Requirement Definition	1 or more	x	x	x
TST-007	Security Auditing	1	x	x	x
TST-009	Security Testing - recommended	Every-sprint	x		
KTY-002	Application Security Settings Definition	1	x	x	
TSK-001	Architecture and Development Guidelines	1	x	x	
SNT-004	External Interfaces Review	1 or more	x	x	x
SNT-006	Attack Surface Recognition and Reduction	Every-sprint	x	x	x
VTM-009	Architectural Security Requirements	1	x	x	
SNT-016	Internal Communication Security - if applicable	Every-sprint	x	x	
TST-001	Security Test Cases Review	1 or more	x	x	x
TST-004	Test Phase Code Review	1	x		
TST-006	Use of Automated Testing Tools	Every-sprint	x		
TST-008	Security Mechanism Review	1	x		
TST-010	Development-time Auditing	1	x	x	x

Dev = Developers, SM = Scrum Master, PO = Product Owner

The included compliance requirements were selected based on their effect to development-time activities, and grouped into following categories:

1. Prerequisites
2. Documentation
3. Code, interface and test case reviews
4. Development-time and product audits
5. Security testing

VAHTI instructions define three security levels: basic, heightened, and high, each with cumulative security requirements. In a Scrum project, the product owner will specify the target level, preferably using tools specified in VAHTI instructions 3/2012

‘Instructions for determining the security level of the technical ICT environment’ [1], the state office’s own instructions, and any applicable legislation. In the next sections, we go analyze each requirement, suggest a means to comply with it and then present the whole VAHTI-compliant Scrum structure for all defined security levels. The security levels are separated by horizontal lines: the first section covers the requirements for basic level, and the heightened and high levels are below that, respectively.

Generic technical requirements, such as ‘the application design must follow secure design patterns’, and architectural requirements, were considered external to the development method. Also, while technology-specific training for the project personnel is included, any organizational security awareness-type training and general risk management activities were also considered to be out of a single development project. Also, technical or programming technique specific requirements were excluded. These are considered to be part of design patterns and individual developer’s proficiency and ability to recognize and utilize them. Moreover, they are depending on the characteristics of the software under implementation. As an exception to the principle of doing only least amount of work, the optional Threat Modeling tasks (VTM-008 and 010) were included already to basic level - although they remain optional. For some reason, VAHTI does not require threat modeling at all. At highest security level, this can be covered by Attack surface recognition and reduction task (SNT-006), or even included in creation of secure patterns and architecture (TSK-001, VTM-009). Similarly, a clearly implementation-dependent requirement for Internal Communication Security (SNT-016) was included, applicable in case of n-tier applications. It is conceivable that a vast majority of software developed for the VAHTI high-security tier is deployed using a secure multi-tier architecture (i.e., separate database, application, and web server components), so the inclusion of this requirement was deemed necessary.

Majority of the tasks fall on the developers; it is conceivable that establishing a role of dedicated security developer would benefit the team, allowing the developers to concentrate on their main vocation. Scrum master, as the ‘servant-manager’, is also involved in most tasks, albeit indirectly by facilitating the team’s work. The Scrum Master is also directly involved in all modeling and planning tasks, reviews, and audits. Following the agile philosophy, the product owner is engaged in development wherever deemed beneficial: as customer’s and stakeholders’ representative, they have the best knowledge of the software’s purpose, use and impact. In addition to the Business Impact Analysis, which is direct input from the stakeholders via the product owner, they participate in external interface and security testing definition tasks, as well as all the audits. These are, after all, done to the customer’s benefit. In the next section, we execute these tasks using Scrum.

5 VAHTI-Scrum

Scrum, as a flexible and relatively adaptable framework, defines only the very rudimentary structure for software development. To achieve compliance with VAHTI, additions and modifications are required to the requirement gathering, sprint planning and the sprint structure themselves. Some security activities justify having a dedicated security or ‘hardening’ sprints. These may prove especially useful just before audits and

planned releases, if the organization has decided to use those. It is also recommendable to arrange a security sprint, 'sprint zero', before commencing the actual implementation; alternatively, some security work may be completed as security spikes. Spikes could occur at any time - mainly in the early phases of the project, so they do not have even approximate placement on the project time line. In similar fashion, the hardening sprints may be inserted between the implementation sprints, to prepare for audits or just enhance security. Especially on heightened and high levels the amount of security work can be so big that a separate sprint is justified. It is important, though, that security elements become integral part of all the team's work: all tasks should include some security considerations, at least in the form of risk analysis, just as all sprints contain security tasks.

Figure 2 presents the modified Scrum process with the additional requirement sources, roles, security related actions during the sprints and the artifact types produced by the security processes.

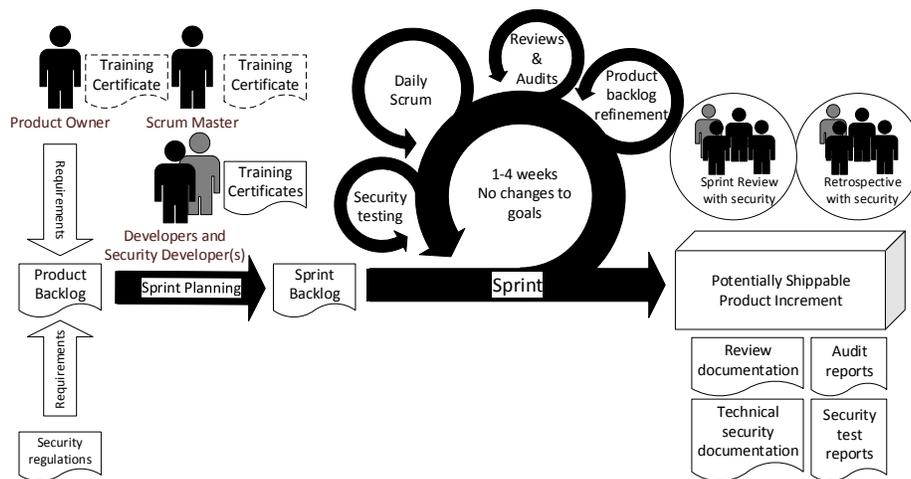


Fig. 2. VAHTI compliant Scrum process

The modifications to standard Scrum are listed below:

- The *security regulation*, although working for the benefit of the stakeholders, is considered a separate source of security requirements. The items, or tasks, added by to the backlog by the regulation are:
 1. Creating security-related documentation artefacts: application risk analysis, threat models, security architecture, goal and criticality definition, business impact analysis, security solutions documentation, application security requirements definition, security settings definition, and architecture and development guidelines.

2. Security training regarding the selected platforms and technologies, such as operating systems, database engines, programming languages and frameworks.
 3. Code, test case and interface reviews
 4. Security testing
 5. Development time and security audits
- There should be at least one dedicated *security developer* in the team, preferably the same person nominated to be responsible for the organization's software security. This results in separation of duties, which enhances security by reducing the amount of *group think*: also somebody else than the developers themselves should design security tests, risk analyses, threat models and attack surface analysis.
 - On heightened and high security levels, all sprints contain certain amount of specifically security-related work and security testing.
 - Audits are performed at a predetermined point in the project. These are set after completing the corresponding items in the product backlog. The development-time audits on the high security level are suggested to be held at control points, which in agile projects map to product backlog items.
 - The team/organization must have nominated a person responsible for the security.

This the following subsections provide a breakdown of security roles, tasks and artefacts on each of the VAHTI security levels.

5.1 Roles

The roles in security-centric Scrum have certain modifications to the standard, plus a new one: the security developer. In VAHTI-Scrum, the team has at least one person in the role of security developer. While still a developer in the Scrum terminology, this role is responsible for security reviews, security test cases and such. It may be beneficial to have more than just the one security developer in a project. The Scrum Master will need a substantial amount of security knowledge and practice, or at least they need to be versatile in adapting to the changing security requirements. Security is an on-or-off deal: there is no middle ground in testing or audits, and ignoring security problems cannot be considered a sustainable idea.

Similarly to the other roles, also the Product Owner has new responsibilities. As the stakeholder's representative, they need to be aware of the security regulations, legislature, customs and other rules, in addition to the normal duties. Being a product owner in a Scrum project may very well become a full-day job on the higher levels.

5.2 Tasks

For each task, we identify the role(s) responsible for its execution (see also Table 1), and the artefacts produced by the task.

Basic level

Security Training (OSK-001).

- The Scrum Master facilitates for (preferably certified) internal or external training and participates when necessary. This is likely to be performed as a spike as it does not directly contribute to the product increment.

Additional security training after change (OSK-008).

- The Scrum Master facilitates for internal or external training after the need has been identified, and participates when necessary.

Application Risk Analysis (VTM-005).

- The team identifies the security concerns, and technology and environment specific risks. Sources such as OWASP Top 10³, Tsipenyuk et al. [18] or Howard et al. [11] in addition to VAHTI's own recommendations should be used to identify software risks. The result of the analysis is used as input for threat modeling. Input mainly from the developers, producing a risk analysis document.

Test Plan Review (TST-002)

- The person nominated to be responsible for the security reviews the test plan. The produced review report is part of the security evidence to prove the software and process is VAHTI compliant.

Threat Modeling (VTM-008)

- Although optional, it is recommended that a formal threat model is created based on the application risk analysis. Done by the developers, results a threat model document.

Threat Modeling updates (VTM-010)

- Optional. When the requirements or environment changes and a new risk analysis is performed, the threat model should be updated.

Heightened level

Goal and Criticality Definition (ESI-001)

- In practice the same requirement as Business Impact Analysis, although from the VAHTI perspective and done by the whole team: the impact and business use of the software is analyzed and its data confidentiality assessed. The resulting document is used as input for security requirement definition.

Business Impact Analysis (ESI-002)

- Similar as above, but concentrates on the software's criticality and impact on the customer's business. Analysis is provided mainly via the product owner.

Documentation of Security Solutions (VTM-001)

- Component level security documentation produced by the developers.

Application Security Requirement Definition (VTM-006)

- Security requirement uses the goal and criticality definition and risk analyses (optionally, also threat models) as an input to define the security requirements. This is a formalization of work that has already been done: the software is already determined to belong to the heightened security level. Done by the whole team, involving also PO.

³ <http://owasptop10.googlecode.com/files/OWASP%20Top%2010%20-%202013.pdf>

The resulting document is used as input for the maintenance phase.

Security Auditing (TST-007)

- Performed by an external and independent auditor, facilitated by the Scrum master. Mandatory parts include automated penetration testing; administrative audit to verify the software's maintenance processes (post development phase - does not concern development); architectural audit from security point of view.

Security Testing - recommended (TST-009)

- While a dedicated set of security tests is not mandatory, it is our recommendation to perform such tests to help passing the security audit. Security test case review is mandatory on the High security level. Inconsistently, actual security testing remains non-mandatory in VAHTI!

Application Security Settings Definition (KTY-002)

- At the end of implementation and before deployment the software settings are documented and a maintenance guide with hardening instructions is written.

High level

Architecture and Development Guidelines (TSK-001)

- This is an organizational document that defines also some coding practices, such as exception handling. If this document does not exist, it will have to be created before security audit. Input from the organization's developers and Scrum masters (if several) can be done in e.g. Scrum of Scrums, a process of synchronizing the Scrums.

External Interfaces Review (SNT-004)

- The external interfaces of the software are reviewed against the architectural guidelines. Input from the developers, review is facilitated by the Scrum master.

Attack Surface Recognition and Reduction (SNT-006)

- All attack vectors are to be identified and security mechanisms planned accordingly. In practice just different approach to threat modeling; formalization of the work done at architectural planning by the developers.

Architectural Security Requirements (VTM-009)

- Based on the attack surface recognition (and threat modeling), the architecture is analyzed against recognized attack vectors by the developers.

Internal Communication Security – if applicable (SNT-016)

- The developers should be aware of the technical environment of the software for this one. For example, if a web application uses separate database or application servers, the communication interfaces must be hardened.

Security Test Cases Review (TST-001)

- Review the quality of the security test cases. The cases should be derived from other documentation and their validity and comprehensiveness is verified (i.e., sanity-checked). Main responsible is with the security developer.

Test Phase Code Review (TST-004)

- Dubbed informal and performed by the person nominated to be responsible for security. Review findings are documented. Developer task.

Use of Automated Testing Tools (TST-006)

- Partially organizational requirement, as acquiring the tools may cause administrative

overhead. VAHTI specifically mentions e.g. fuzzers and code analyzers. This task is performed by the developers; the Scrum master facilitates acquiring the tools.

Security Mechanism Review (TST-008)

- Code level review of security mechanisms. Check list is to be derived from architectural level documents and other relevant documentation produced during the VAHTI-Scrum development process. Done by the developers and the security developer.

Development time Auditing (TST-010)

- One or more external audits to be performed at different phases of development. VAHTI only states that the software's security is to be audited, so it is to be agreed with the stakeholders how to handle this. A good baseline would be architecture, interfaces, security mechanisms and/or the review documentation. Involves all roles

5.3 Artifacts

Software security assurance is provided by evidence, and in most cases this means documentation: reports, plans, technical documents, memos and other document artifacts considered relevant for security. VAHTI is not an exception to that, and pragmatically speaking, producing the required artifacts *fulfills the VAHTI Definition of Done*. The security documentation reflects a significant part of work. However, the figurative security burn down chart does not reach zero even when the last document has been finalized: security tasks will remain a part of every sprint even after that.

The produced artifacts in a VAHTI-Scrum and their dependencies are outlined in Figure 3.

Figure 3 is a matrix with the security levels on the x-axis and development phases on the y-axis. Arrows indicate input; the document is a result of a process of the same name. It should be noted that if the project is operating on high security level, the Architectural and Development Guidelines document may and should be used as input for other documentation. For clarity, the arrows do not cross the security level barriers from higher level to a lower one. Items with dashed lines are optional, yet recommended. On the bottom row, we have the external documents on higher levels: audition reports, and input for the deployment and maintenance phases of the software's life cycle.

6 Conclusions and future research

Software security and security regulation aim at a defined process, producing a measurable, evidence-backed result. This result is called security assurance or security compliance. In Finland, the state agencies have formalized their security requirements into the form of collection of VAHTI instructions; these instructions also concern application development. Standardized secure software development methods have deep roots in the waterfall era, but 'agile' is not the antithesis for defined processes or structured way of working.

In our earlier work, we were interested about the adaptability of the agile methods to the software development in general [16]; this study further utilized the findings and analysis, and provided an example how and with what mechanisms the Scrum method can be adapted to provide compliance with VAHTI instructions. This answers to our two

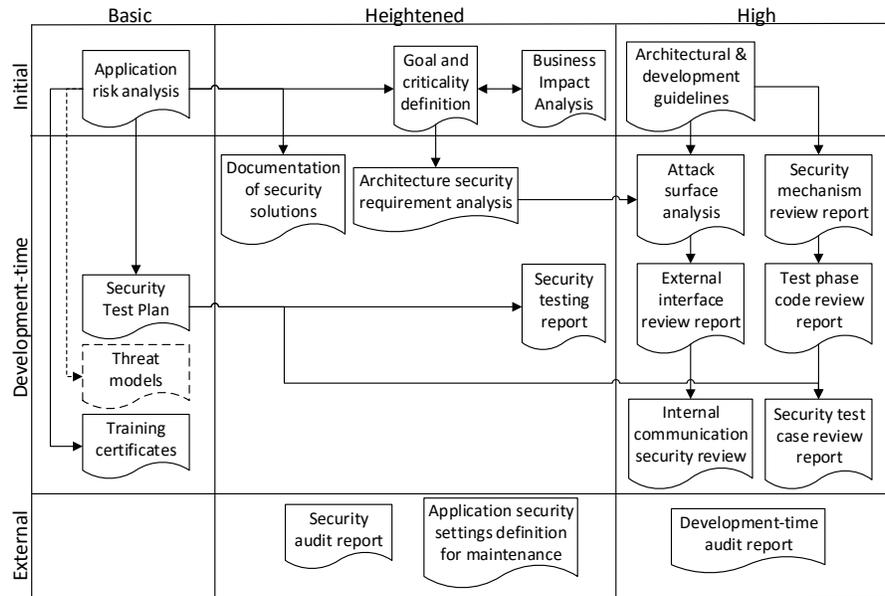


Fig. 3. VAHTI documentation artifacts

first guiding research questions. The Definition of Done in security context is twofold: the formal requirements may be fulfilled, and contribute to the actual DOD; security may be considered ‘done’ only when the project finishes - or, even when the software life cycle ends.

Naturally, this study has limitations. The presented model is based on a conceptual-analytical approach and it has not been empirically evaluated. While we have carefully addressed the two studied concepts and created the new model based these results, it is possible that some of the proposed modifications can be improved. Thus, further work is needed to validate and verify the model with, e.g., a case study conducted in an university’ course with students or in an industrial setting.

The research field offers several complex and interesting opportunities for future study: while we presented a method to fulfill VAHTI’s requirement with agile approach, it would be fruitful to understand how industry is currently working with the requirements. A qualitative case study with selected companies is planned to help identify the best practices and methods to use.

Furthermore, benefits and drawbacks introduced to security and safety development by agile methods should be studied. While in a quick glance, it seems that agility in development and security of the product are competing objects that cannot be easily achieved in the same project, an analysis of advantages and disadvantages of using agile in secure software development should be performed. This would, furthermore, bring an answer to the question, what are the reasons to adopt agile in the an environment which is not the best for it?

References

1. Teknisen ympäristön tietoturvaso-ohje, <https://www.vahtiohje.fi/web/guest/3/2012-teknisen-ympariston-tietoturvaso-ohje>, ref. 18th August 2015
2. Cmmi for development, version 1.3 (2010), <http://www.sei.cmu.edu/reports/10tr033.pdf>, ref. 18th August 2015
3. Beck, K., Beedle, M., Van Bennekum, A., Cockburn, A., Cunningham, W., Fowler, M., Grenning, J., Highsmith, J., Hunt, A., Jeffries, R., Kern, J., Marick, B., Martin, R.C., Merllor, S., Schwaber, K., Sutherland, J., Thomas, D.: Manifesto for agile software development (2001)
4. BSIMM: The Building Security In Maturity Model, ref. 2015.08.20
5. De Win, B., Vanhaute, B., De Decker, B.: Security through aspect-oriented programming. In: De Decker, B., Piessens, F., Smits, J., Van Herreweghen, E. (eds.) *Advances in Network and Distributed Systems Security*, IFIP International Federation for Information Processing, vol. 78, pp. 125–138. Springer US (2002), http://dx.doi.org/10.1007/0-306-46958-8_9
6. Deemer, P., Benefield, G., Larman, C., Vodde, B.: *The Scrum Primer: The lightweight guide to the theory and practice of Scrum*. InfoQ, version 2.0 edn. (2012)
7. Diaz, J., Garbajosa, J., Calvo-Manzano, J.: Mapping cmmi level 2 to scrum practices: An experience report. In: O'Connor, R., Baddoo, N., Cuadrado Gallego, J., Rejas Muslera, R., Smolander, K., Messnarz, R. (eds.) *Software Process Improvement, Communications in Computer and Information Science*, vol. 42, pp. 93–104. Springer Berlin Heidelberg (2009), http://dx.doi.org/10.1007/978-3-642-04133-4_8
8. Fitzgerald, B., Stol, K.J., O'Sullivan, R., O'Brien, D.: Scaling agile methods to regulated environments: An industry case study. In: *Proceedings of the 2013 International Conference on Software Engineering*. pp. 863–872. ICSE '13 (2013)
9. FMoF: Sovelluskehityksen tietoturvaohje (2013), <https://www.vahtiohje.fi/web/guest/vahti-1/2013-sovelluskehityksen-tietoturvaohje>, ref. 17th March 2015
10. FMoF: Vahti-ohje (2015), <http://www.vahtiohje.fi>, <http://www.vahtiohje.fi>, Referenced 17th March 2015
11. Howard, M., LeBlanc, D., Viega, J.: *19 Deadly Sins of Software Security*. McGraw-Hill, Inc., New York, NY, USA, 1 edn. (2006)
12. ISO/IEC: *Information Technology - Security Techniques - Systems Security Engineering - Capability Maturity Model (SSE-CMM) iso/IEC 21817:2008*
13. Järvinen, P.: Research questions guiding selection of an appropriate research method. Series of Publications D – Net Publications D–2004–5, Department of Computer Sciences, University of Tampere, Tampere, Finland (December 2004)
14. Microsoft: *Microsoft Security Development Lifecycle*. Microsoft (2012)
15. Pietikäinen, P., Röning, J.e.: *Handbook of The Secure Agile Software Development Life Cycle*. University of Oulu (2014)
16. Rindell, K., Hyrynsalmi, S., Leppänen, V.: A comparison of security assurance support of agile software development methods. In: *Proceedings of the 16th International Conference on Computer Systems and Technologies*. p. TBA. ACM (2015)
17. Solinski, A., Petersen, K.: Prioritizing agile benefits and limitations in relation to practice usage. *Software Quality Journal* pp. 1–36 (2014), <http://dx.doi.org/10.1007/s11219-014-9253-3>
18. Tsipenyuk, K., Chess, B., McGraw, G.: Seven pernicious kingdoms: a taxonomy of software security errors. *Security Privacy, IEEE* 3(6), 81–84 (Nov 2005)
19. VersionOne: *8th annual state of agile survey (2013)*, <http://www.versionone.com/pdf/2013-state-of-agile-survey.pdf>, Referenced 17th August 2015
20. Williams, L., Cockburn, A.: Agile software development: it's about feedback and change. *Computer* 36(6), 39–43 (June 2003)

Collecting Issue Management Data for Analysis with a Unified Model and API Descriptions

Otto Hylli, Anna-Liisa Mattila, and Kari Systä

Department of Pervasive computing, Tampere University of technology, Tampere, Finland

{otto.hylli,anna-liisa.mattila,kari.systa}@tut.fi

Abstract. Reuse of analysis methods and tools for data from different issue management systems is challenging because there are differences in how the data is accessed and represented. While various approaches for collecting and analysing software engineering data have been developed, they do not generally pay so much attention into how to actually get the data from various sources. This paper presents a combined model for issue management data that is based on an investigation of four issue management systems. It also presents a proof of concept tool that can collect issue management data from different services into our analysis and visualization framework using an API description language that defines how to get the issue management data and how to convert it into our model. The aim of this approach is to allow the addition of new data sources by simply providing their API descriptions.

Keywords: issue management, data model, software repository mining

1 Introduction

Issue management is an integral part of software development and management. Various tools have been developed for that purpose e.g. Jira and the issue tracking feature of GitHub. The information collected into the issue management system can be used to analyze the software project and it can give valuable insights, that can help in managing the project, e.g. how to automatically identify valid bug reports [12].

Many issue management systems offer an API that can be used to acquire data. A tool, that could fetch issue management data from multiple sources and offer the same interface and analysis features regardless what the data source is, would be useful for both its users and developers. There are some differences in how different issue management systems handle and represent issues and related concepts. Thus if the same analysis tools and notations are to be used to analyze issues from multiple sources, a common model for issue management has to be defined. Also a generic method for collecting issue data from different sources and converting it into this model has to be developed. This paper presents a combined data model based on four different issue management systems. It also

presents a tool that uses API descriptions in collecting and converting issue management data from different services.

This paper is organized as follows. Section 2 discusses the motivation and background of this research in more detail. Section 3 presents our investigation into different issue management systems. Section 4 describes the combined issue model that is based on the investigation. Section 5 describes the implementation of a issue collection tool that uses API descriptions in collecting issue management data according to the model. Section 6 presents discussion about our approach. Section 7 presents related work and section 8 presents conclusions.

2 Background and Motivation

An issue management system serves many purposes in an organization. It is a knowledge repository, a communication and collaboration hub and a communication channel for requests for new features, bug reports or any task that development team should perform [2]. Thus an issue management system contains different types of useful information for analysis. However, issue management systems are different in what data is stored and how the data is accessed. The organization's practices also affect how issues are used.

This research is related to previous research done in our department considering software engineering data analysis and visualization. In [9] we present a study where software engineering data from different data sources were combined and visualized to show realization of continuous deployment. This research has led to the development of a unified model for software engineering data and a framework for collecting, storing and accessing it [10]. This data can come from various sources and represent different domains such as issue management, version control or testing. To collect data from a specific domain an intermediate domain model can be used. This paper presents such a model for issue management data. Data can be first collected according to the intermediate model then converted into the higher level unified model.

In our previous work we have also developed a method for building Internet service compositions [7]. There we dealt with similar issues i.e. fetching conceptually similar data from different services, and using the data in a unified way to implement mash-ups. We developed the concept of generic data types that represented different concepts that many services handle like photo or status update. Then we added information about the generic data types to the service API descriptions so that they could be used in service compositions. In this work we want to use a similar approach for easily gathering issue management data from different services. We do not just want to write separate tools or plug-ins for fetching and converting data from different systems. Instead we want one generic tool that can be given descriptions of the APIs of the source systems. This would then make it much quicker to add different data sources.

3 Issue Management Data

To find a model for issue management data, we surveyed four web based systems that offer issue management. The systems were Jira¹, GitHub², GitLab³ and Bitbucket⁴. Jira is a dedicated issue management system. GitHub, GitLab and Bitbucket offer code project hosting that include in addition to a issue tracker a code repository based on a version control system. We investigated what information a issue holds, what other concepts are related to issues and how the systems record changes and other activity related to the issues.

3.1 Properties

First we defined what information an issue contains, i.e., the properties of the issue. We listed the properties from each system and combined those that meant the same thing. Properties can be simple attributes or relations to separate entities, who have their own attributes. We found 26 different properties and 9 of these properties are common to all the systems. All issues have some kind of unique identifier, a title or summary and a longer explanation about the issue. The issue systems also record when an issue was created and when it was last updated. Issues also have a status or state that indicates the current phase in the workflow. Possible states in the workflow varies by system from just *opened* and *closed* offered by GitHub to the user customizable workflows of Jira.

All of the systems have authenticated users and they can be related to an issue as the creator. All systems support also assigning the issue to a user who then is responsible for progressing the issue's resolution. Issues can also be discussed in all systems with a commenting feature.

Two properties issue labeling and associating issues to milestones are shared with three of the systems. Seven properties are shared between two systems. In six cases those systems are Jira and Bitbucket. They let issues be categorized with types, offer possibility to associate issues with software versions and specific components. They also allow issues to be prioritized.

Jira is the most advanced of the systems. It offers eight properties that the other systems do not offer. It allows the type of the resolution to be recorded for example *fixed* or *cannot reproduce*. It also offers features for estimating and recording the amount of work for the issue. Issues in Jira can also be linked to related issues. In addition Jira is customizable offering a possibility to add custom fields.

3.2 Issue changes

All of the systems record changes to the issues such as changes in the issue state and properties. What changes are recorded and how they are accessed varies. In

¹ <https://www.atlassian.com/software/jira>

² <https://github.com>

³ <https://gitlab.com>

⁴ <https://bitbucket.org>

all of the systems the user who made the change and the time the change was made is recorded.

GitHub records issue changes as issue events. They can be accessed for the whole project or for a specific issue. These events have a type that indicates what kind of change the event represents for example *closed*, *opened*, *assigned* or *labeled*. The event contains also information about what the change was e.g. what label was added to the issue. GitLab has project specific events that include events about issues but have other events also. There are events only for issue opening and closing. Some other changes such as labeling or assigning are just saved as comments of the issue

Bitbucket has also project events. However the feature is limited since only 30 most recent are available. Their content is also quite limited. There are events for issue creation, commenting and updating but the update event does not have specific information about what was updated and the creation event does not tell what issue was created. In Jira each issue has a changelog. It records each change of the issue. A change record contains the property whose value was changed, its old value and the new value.

4 Data Model

This section presents the issue management data model that we developed based on our investigation of the issue management systems. It also shortly presents the unified software engineering data model used by our analysis and visualization framework, presented in [10], and its relation to the issue management model.

4.1 Issue Management Model

Our investigation shows that issues in the different systems have quite much in common. This enables the definition of a combined model for issues. It has eight different entities: *issue*, *user*, *comment*, *milestone*, *version*, *component*, *label* and *change event*.

An *issue* in our model has all of the properties presented in section 3.1. Most of them (18 out of 26) were shared at least with two of the systems and the rest are also useful. Table 1 lists the properties of an issue with their types and descriptions.

Label, *milestone*, *version* and *component* are similar simple entities. They have a name and a description and can be associated with issues. *Milestone* has also a due date, a creation time and a closing time. *User* represents a user of the issue management system. It can be associated with a *issue* as the issue creator and as an assignee. *Comments* consist of the comment message and the commenting time. They are associated with an *issue* and the *user* who posted the comment.

Each *issue* has a changelog. It consists of *change events* that record when the change was made, and optionally what property was changed and how i.e. what is the new value for the property. *Change event* is also associated with the *user* who made the change.

Table 1. The properties of an issue entity.

Property	Type	Description
id	string	An unique identifier for the issue in the management system
number	string	An unique identifier for the issue in a project that is not unique in the whole system
title	string	Describes the issue shortly
description	string	A longer explanation of the issue.
state	string	The current state of the issue in the issue workflow
author	user	User who created the issue
created	datetime	When was the issue created
updated	datetime	When was the issue last updated
assignee	user	The person who is responsible for the issue.
comments_count	integer	How many comments there are about the issue
comments	list of comments	Comments about the issue
change log	list of change events	Changes made to the issue
labels	list of labels	Tags that are used to categorize issues.
milestone	list of milestones	Used to categorize issues to be implemented in a specific version or sprint
priority	string	How important is the issue.
type	string	The type of the issue e.g. bug, feature
resolved	datetime	When was the issue resolved or closed.
affects version	version	The version the issue affects
fix version	version	The version in which the issue should be resolved.
component	component	The software component associated with the issue
watchers	integer	How many users are interested about the issue e.g. they get notified about issue changes
resolution	string	How was the issue resolved e.g. fixed, won't fix
environment	string	In what kind of environment the issue occurs
votes	integer	How many votes the issue has
due	datetime	When the issue should be resolved
estimate	integer	Original estimate of the time required to resolve the issue (minutes)
remaining	integer	Current time estimate (minutes)
logged	integer	How much work has been done for the issue (minutes)

4.2 Relation to Unified Software Engineering Data Model

The unified software engineering data model, mentioned in Section 2 and described in [10], consists of two concepts: *artifact* and *event*. *Events* present actions that happen in software engineering projects. They have an author, a type, the time the *event* happened and a duration. *Events* are related to *artifacts* the *event* happened to. *Artifacts* represent various aspects of software engineering that are interesting for visualization and analysis purposes. For example an *artifact* can be a file in version control and commits to that file are *events* related to it. *Artifacts* can also be related to each other. An *artifact* can have a state and it can be changed by an *event*.

The issue management model presented in the previous section works as a domain specific model for the unified model. It can be mapped to the unified model and so issue management data can be saved according to the unified model. Of the entities issue, milestone, label, component and version are *artifacts*. Change events and comments are *events*. Users can be modeled as *artifacts* or they can be just attributes for *events*. Additionally *events* can be generated from some of the entities' time based attributes. For example issues and milestones have an attribute that tells when they were created. From this attribute creation or opening *events* can be generated.

5 Implementation

This section presents the implementation of an issue collector tool that uses the issue model and the unified software engineering data analysis framework. First an overview of the tool is presented. Then its API description system is presented. Finally a usage example illustrates how the tool works.

5.1 Overview

Our web based data analysis and visualization framework [10] offers a database for the software engineering data and an HTTP API for storing and querying it. These APIs can be used by different data collection, analysis and visualization plug-ins.

To test the feasibility of the issue management data model and data collection approach presented in this paper a tool was developed that can fetch issue management data from different web based issue management systems according to the issue management model. The data collection involves making HTTP requests to various API endpoints like issues and milestones. The responses to these requests will contain lists of items in the JSON format that should be converted into various entities of the issue management model. After getting the data the tool then converts the issue management data into the format of the unified data model and sends it to the database.

The architecture of the tool is shown in Fig. 1. The tool consists of four components. *API descriptions* define how to get issue management data from

different sources. The *user interface* handles user input required for the data collection. The *API description* defines what input data is required. The *Collector* uses the user input to get data from a service described by an *API description*. The *Unifier* converts the data in to the unified model and sends it to the database.

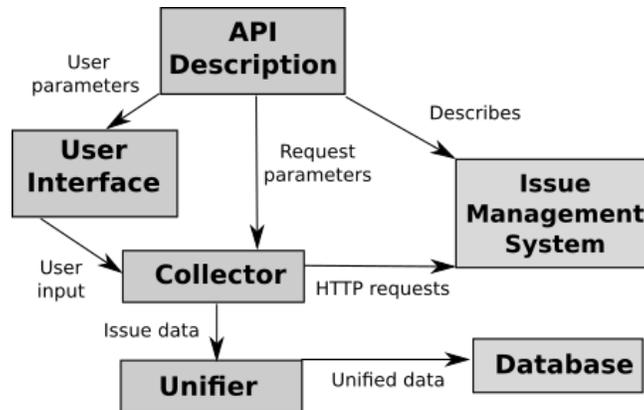


Fig. 1. The architecture of the issue collector tool.

The tool is implemented with Node.js. Currently the *user interface* is command line based. The current version of the tool does not yet cover the whole issue management model. It can process *issues*, *comments*, *milestones* and *change events*.

5.2 API Descriptions

An *API description* is a JavaScript object whose properties describe the API. The *API description* consists of general properties and resource descriptions. General properties describe general information about the API i.e. information that is common to all API calls. Resource descriptions describe information specific to API calls to one particular resource such as issues.

Table 2 lists the general properties. General properties for an API description include the common part of the API URL, possibly some HTTP headers and query parameters. Headers and query parameters are defined as simple objects containing key-value pairs where the key is the header or query parameter and the value its value. If the value is not static, the value undefined is used. This indicates that the value has to come from the user.

For defining what information is required from the user in the *user interface* the *API description* has an *userParams* property. This information is usually project specific information such as identification of project to be targeted. The value of *userParams* is a list of objects that contain the name of the parameter

Table 2. General API description properties.

Property	Type	Description
BaseUrl	string	The part of the URL that is common to all API requests
authentication	list of objects	A list of ways an user can authenticate to the service
pagination	string	How does the API handle pagination.
headers	object	HTTP headers required by every API call
query	object	Query parameters required by every API call
userParams	object	Defines what information is required from the user for issue management data collection
resources	resource description	Information about how to get and convert items from one API end point. Resources can be issues, milestones, changeEvents or comments.

and a description of the parameter that is shown to the user. The parameter name indicates where the value will be used. For example it can be used as a value for a header or query parameter that has the same name.

Issue management services can support multiple ways for their API users to authenticate. The *authentication* API description property lists the authentication methods that the service supports. A value in the list can be a string containing the name of an authentication method that the tool understands. Currently recognized methods are no authentication and HTTP basic. A value can also be an object that defines a custom authentication method which can contain additional headers, query parameters and user parameters.

Most API calls do not return everything at once. Instead they return up to a certain number of items and the client has to request more. The *pagination* property defines how this pagination in API calls is handled. Currently only pagination using a RFC 5988 link header, that has the pagination information, is supported.

The *API description* can contain multiple properties that have a resource description object as the value. The name of the property indicates what kind of entities the description describes. Currently supported values are issues, comments, milestones and changeEvents.

A resource description is an object whose properties describe a particular API resource, i.e., a concrete API end point that we want to make a HTTP call to. A resource could be for example the list of issues in a project or a list of one issue's comments. Table 3 lists the properties that a resource description can have.

The *path* property holds the rest of the HTTP request URL. The value is a RFC 6570 URI template whose variables have to be expanded before making the request. Values for these variables are found from the similarly named

Table 3. Properties of a resource description.

Property	Type	Description
path	string	The rest of the URL as an URI template.
query	object	Query parameters specific for this resource.
headers	object	HTTP headers specific for this resource
filter	function	Function used to decide if the current item will be processed.
item	item description object	How to convert one item from the resource into an entity in the issue management model
createOpeningEvents	bool	Create a change event from the created attribute of the new entity.
createUpdatingEvents	bool	Create a change event from the updated attribute of the new entity.
createClosingEvents	bool	Create a change event from the closed attribute of the new entity.
children	object	Contains resource descriptions of the current resource's child resources.
parentParams	object	Information required from a parent resource for getting a child resource.

user parameters. The *query* and *headers* properties are similar to the corresponding general properties but provide resource specific information. The *createOpeningEvents*, *createUpdatingEvents* and *createClosingEvents* properties indicate if additional change events should be created from the new entity's created, updated or closed properties.

The *filter* property can hold a function that is used to choose which items received from the issue management system are processed. The function is given a single item from the response like an issue and its boolean return value determines if that item should be processed.

The *children* property describes the current resource's child resources such as the comments of an issue. Its value is an object whose properties have resource descriptions as values similar to the general API description. The *parentParams* property is applicable only in child resource descriptions. Like user parameters its values can be used in the URI template, headers and query parameters but the source for the values is the child's parent entity.

The *item* property gives information on how to convert one item from the response in to an entity of the issue management model. The properties of an item description object correspond to the properties of the entity to be extracted. The value tells how to extract the value for the new entity's property. For describing how to extract the value we use JSONPath⁵. JSONPath expressions are used to select a specific part of a JSON document or JavaScript object. The value of an item description property can be a string or an object. The *path* property of that object holds the JSONPath expression. The *source* property tells where the

⁵ <http://goessner.net/articles/JsonPath/>

Listing 1.1. A part of the GitHub API description. Most item descriptions are not shown and only part of the comment's item description is shown.

```

var api = {
  baseUrl: 'https://api.github.com/',
  authentication: [ 'no authentication', 'basic' ],
  headers: { Accept: 'application/vnd.github.v3+json',
    'User-Agent': 'ohylli/issue-collector' },
  pagination: 'link_header',
  userParams: [ { name: 'owner',
    description: 'The user name of the repository owner' },
  { name: 'repo',
    description: 'the repository name' } ],
  issues: {
    path: '/repos/{owner}/{repo}/issues',
    query: { state: 'all' },
    filter: function ( item ) {
      return item.pull_request !== undefined; },
    item: { ... },
    createOpeningEvents: true,
    createUpdatingEvents: true,
    children: { comments: {
      path: '/repos/{owner}/{repo}/issues/{number}/comments',
      parentParams: { number: '$.number' },
      item: { id: '$.id',
        issue: { path: '$.id', source: 'parent' },
        user: '$.user.login',
        message: '$.body', ... } } }, ... };

```

value is to be extracted from. Possible values are `item`, which means the item received from the service, and `parent`, which means the parent entity of the new entity. The *mapping* property can be used to replace the extracted value with another value. If *source* is the item and there is no mapping information, the object can be replaced with a string containing the path information.

5.3 Usage example

As an example of the tool's usage we tested the method to collect issue management data from four public open source projects : `grip`⁶, `glutin`⁷, `gfx`⁸ and `webgl-noise`⁹. The `webgl-noise` project is the smallest of the four projects containing 14 issues where as `gfx` is the largest containing 304 issues. `Glutin` project has 187 issues and `Grip` 107 issues. The projects use GitHub as a code repository and issue management system. Thus we require an *API description* of GitHub's API which is shown in listing 1.1.

When the issue collector is invoked, it first checks what *API descriptions* are present and asks the user which of these she wants to use. The issue collector loads the *API description* the user chose and first checks what authentication methods are available and lets the user choose the one she prefers. As can be seen on the line 3 of the Listing 1.1 GitHub issue collector can be used without authentication or with HTTP basic authentication.¹⁰ If the user chooses basic authentication, the tool next asks the user for her username and password

⁶ `grip` – <https://github.com/joeyespo/grip/issues>

⁷ `glutin` – <https://github.com/tomaka/glutin/issues>

⁸ `gfx` – <https://github.com/gfx-rs/gfx/issues>

⁹ `webgl-noise` – <https://github.com/ashima/webgl-noise/issues>

¹⁰ GitHub supports also OAuth2 authentication but our tool does not yet support it.

required by HTTP basic authentication. Next issue collector checks what additional API specific information is needed from the user. From the lines 7-10 of the Listing 1.1 we see that two user parameters named *owner* and *repo* are required. The issue collector queries inputs for these showing their descriptions to the user. Lastly the tool queries the user for some metadata required by the unified data model.

Next the *collector* can begin the actual data collection. It goes through every resource description, makes HTTP requests they define and converts the data received into the appropriate entities. For constructing the HTTP requests the *collector* gets the beginning of the URL from line 2. Lines 4 and 5 define that all HTTP requests have to contain two specific headers. If HTTP basic authentication was chosen the authentication information provided by the user is also added to the requests. Then, for example, from the resource description for issues the *collector* gets the rest of the URL from line 12. This URI template has two variables *owner* and *repo*. The *collector* gets values for these from the similarly named user parameters. The resource description also defines on line 13 that the URL has to include a query parameter named *state* with the value *all*. After making the request, the *collector* processes each item in the response. Since on lines 14-15 issues resource has a *filter* function, that is executed first and the item is processed only if it returns false. In this case the function is used to filter out pull requests which GitHub includes with the issues.

The *API description* defines on line 19 that issues have comments as children. This means that for each issue entity created its comments should be fetched as well. The *path* on line 20 is expanded with the *owner* and *repo* and also the *number* property of the parent issue. This is defined on line 21 with the *parentParams* property. The actual comment entity is constructed according to the information on lines 22-25. It defines for example that the *message* property of a comment can be found from the response item's property named *body*. It also defines that the *id* of the issue the comment is related to can be found from its parent entity's *id* property.

After each item in a response has been processed, the *collector* checks if the response contains a *link* header that has the URL for the next page of items, and if it does, it makes a request to it. This behaviour is specified on the line 6 of the Listing 1.1. After all entities are collected, the *collector* checks if additional change events have to be created. For example line 17 defines that from each issue entity a change event has to be created. This event's change type will be *opened* and the time the creation time of the issue.

After the *collector* is finished, the *unifier* converts the issue management data into unified model's artifacts and events which are then send to the database. After this the user can use the visualization framework's analysis and visualization features. Figure 2 has an example visualization from the *grip* project's data that shows each artifact's events and life spans on a timeline. From the visualization we can see for example how long different issues have been open and if the issue has been reopened. Also comment, label, reference and delabel times are visible

for each issue. This kind of view enables comparing issue lifespans to each other as well as finding similarities and patterns from issue events.



Fig. 2. Visualization of lifespans and events from Grip open source project. Each artifact has its own row to display events and lifespan. The lifespan starts when the artifact is opened and ends when it is closed. Lifespan is shown as a line in artifacts row. The dots mark different kinds of events. The dot colors are mapped to event types and those are explained in the top of the visualization. At the right of the artifact line the artifact type and at the left the artifact id are presented. All artifact rows are not visible in the figure as the figure is cropped to save space.

6 Discussion

Our issue management data model is based on a survey of four issue management systems. Although there are many more issue management systems we believe that our model covers the most important aspects of issue management. However in our future work we should verify our model by using it with systems that we did not survey and if the need arises to expand our model. Our model is quite simple and not as expressive as for example an ontology based approach. However, our aim was a light weight model for data storage and testing the API description approach, and for that purpose we believe our model is suitable.

The implementation of our issue collector tool shows the basic feasibility of the model. We used the model successfully with GitLab and GitHub for which we currently have API descriptions. The implementation and those API descriptions also shows the feasibility of our data collection approach. This approach has its strengths and weaknesses. The descriptions are declarative so a description author does not need to worry how the data is collected. When the APIs behave similarly such as GitHub and GitLab do when fetching issues and their comments, the approach works well. However, when there are differences in how things are done like with change events, the tool's implementation and API description have to take them in to account, which will cause complexity in the

implementation code and in the API description syntax. API rate limiting of the services can cause problems when fetching data from bigger projects and we must find ways to deal with them. Currently the tool is a proof of concept implementation and probably new issue sources such as Jira and Bitbucket could not be added just by adding their API descriptions since there are many things the implementation does not support yet. For example pagination is supported only with a link header which all services do not support so a custom pagination implementation would be required. More advanced data extraction features for more complex data structures are also required.

In our previous work on Internet service compositions [7] we used the Web Application Description Language (WADL) to describe the service APIs. Then we had to add additional metadata to describe the service and its data. In this work we wanted to try a different approach with our own JavaScript based API descriptions. It allowed us to combine the API description and the data description required in the data conversion. We could also add features that support common higher level tasks such as authentication and pagination. We can also add functionality to the API descriptions with JavaScript functions which we used in filtering the items. They could also be used for example with custom pagination implementations in the future. Though this approach requires the author of an API description to know JavaScript, the descriptions are quite simple and do not use advanced features of the language.

7 Related Work

On a high level this work can be seen to be related to research into extract, transform and load (ETL) processes used in data warehousing to integrate data from different sources for business reports. ETL research deals with similar problems as our research such as how to combine data from different schemas into a single schema and what is the workflow of the ETL process [11]. More precisely this work is a part of the research in to software repository mining where different tools for collecting and analyzing issue management data among other software engineering data have been developed. However these tools do not pay so much attention in making the data collection generic. Fischer et al. [3] developed a SQL based release history database for collecting and analyzing data for version control and issue management. The system does not include special features for data collection from different sources. Issue management data is just collected with custom scripts from Bugzilla.

Some approaches use semantic web technologies and define an ontology for software engineering data. Kiefer et al. [8] developed EvoOnt which focuses on software evolution. It includes models for the software, version history and bugs. The EvoOnt issue model is based on Bugzilla but it is similar to ours though there are some differences in what concepts of issue management are covered. The paper does not go much in to the details of the model like what properties and relations it supports or what if any change data is collected. This system also has no special consideration for data collection. Dhruv [1] offers semantically

enriched features for members of an open source community to work with issues and related information. Dhruv was developed for a particular open source community that uses particular tools though the developers point out that it could be made to work with other communities and tools, because its model should be general enough and its architecture supports expansion.

Evolizer [4] is a tool whose main focus is in analyzing code changes but its software metamodel includes issues and has an exporter for getting issue data from Bugzilla. It is an Eclipse plug-in and its extension including the addition of new issue data importers takes advantage of Eclipse's plug-in extension features. Goeminne and Mens [5] have developed a framework for analyzing and comparing the evolution of open source projects which is mainly focused on different metrics. It uses the FLOSSMetrics data base [6] which defines a schema for various data collectors including issue data collectors. The FLOSSMetrics issue data collector supports two issue management systems: Bugzilla and SourceForge.

8 Conclusions

Analysis of issue management data can give useful insights in to a software engineering project. In our previous work we had developed an unified software engineering data model and a framework for storing and accessing it. Collecting issue management data from various sources and converting it in to the unified format for analysis presents challenges. We tackled these challenges by first investigating four different web based issue management systems. Based on that we developed a combined issue management data model. It consists of eight entities such as issue, milestone and comment. These entities, their properties and relations cover the essential parts of issue management and allows data from various sources to be stored and analyzed.

We also developed a proof of concept issue data collection tool which collects issue data according to our issue management model and then converts the data into the unified data model's format. The tool uses declarative API descriptions which define how data is fetched and converted. The current version of our tool is limited but it proves the feasibility of our approach. The end goal of our approach is to allow new data sources to be added quickly just by providing an API description that can then be used to fetch data from different projects in that source. We believe that this approach can be expanded to cover different types of software engineering data such as version control data. In our future work we will explore the potential of this approach with other issue management systems and other kind of data. This approach might also have uses in other contexts.

Acknowledgments

The research has been supported by Tekes-funded Digile project Need for Speed¹¹ and by Foundation of Nokia Corporation¹².

¹¹ <http://www.n4s.fi/en/>

¹² <http://www.nokiafoundation.com/>

References

1. Ankolekar, A., Sycara, K., Herbsleb, J., Kraut, R., Welty, C.: Supporting Online Problem-solving Communities with the Semantic Web. In: Proceedings of the 15th International Conference on World Wide Web. pp. 575–584. WWW 2006, ACM, New York, NY, USA (2006)
2. Bertram, D., Volda, A., Greenberg, S., Walker, R.: Communication, collaboration, and bugs: the social nature of issue tracking in small, collocated teams. In: Proceedings of the 2010 ACM conference on Computer supported cooperative work. pp. 291–300. ACM (2010)
3. Fischer, M., Pinzger, M., Gall, H.: Populating a Release History Database from version control and bug tracking systems. In: Proceedings of the International Conference on Software Maintenance. p. 23. ICSM 2003, IEEE, Washington, DC, USA (2003)
4. Gall, H.C., Fluri, M., Pinzger, M.: Change analysis with evolizer and changedistiller. *IEEE Software* 26(1), 575–584 (2009)
5. Goeminne, M., Mens, T.: A framework for analysing and visualising open source software ecosystems. In: Proceedings of the Joint ERCIM Workshop on Software Evolution (EVOL) and International Workshop on Principles of Software Evolution. pp. 42–47. IWPSE-EVOL 2010, ACM, New York, NY, USA (2010)
6. Gonzalez-Barahona, J.M., Robles, G., Dueñas, S.: Collecting data about floss development: the flossmetrics experience. In: Proceedings of the 3rd International Workshop on Emerging Trends in Free/Libre/Open Source Software Research and Development. pp. 29–34. ACM (2010)
7. Hylli, O., Lahtinen, S., Ruokonen, A., Systä, K.: Resource description for end-user driven service compositions. In: IEEE 2nd International Workshop on Personalized Web Tasking (PWT 2014) (June 2014)
8. Kiefer, C., Bernstein, A., Tappolet, J.: Mining Software Repositories with iSPARQL and a Software Evolution Ontology. In: Proceedings of the 15th International Conference on World Wide Web. p. 10. MSR 2007, IEEE, Washington, DC, USA (2007)
9. Mattila, A.L., Lehtonen, T., Systä, K., Terho, H., Mikkonen, T.: Mashing Up Software Issue Management, Development, and Usage Data. In: Proceedings of RCoSE – 2nd International Workshop on Rapid Continuous Software Engineering (2015)
10. Mattila, A.L., Luoto, A., Terho, H., Hylli, O., Sievi-Korte, O., Systä, K.: Unified model for software engineering data. In: 3rd IEEE Working Conference on Software Visualization (VISSOFT 2015) (September 2015)
11. Vassiliadis, P.: A survey of extract-transform-load technology. *International Journal of Data Warehousing and Mining* 5(3), 1–27 (2009)
12. Zanetti, M.S., Scholtes, I., Tessone, C.J., Schweitzer, F.: Categorizing bugs with social networks: a case study on four open source software communities. In: Proceedings of the 2013 International Conference on Software Engineering. pp. 1032–1041. IEEE Press (2013)

LOGDIG Log File Analyzer for Mining Expected Behavior from Log Files

Esa Heikkinen, Timo D. Hämäläinen

Tampere University of Technology, Department of Pervasive computing,
P.O. Box 553, 33101 Tampere, Finland

Email: esa.heikkinen@student.tut.fi
timo.d.hamalainen@tut.fi

Abstract. Log files are often the only way to identify and locate errors in a deployed system. This paper presents a new log file analyzing framework, LOGDIG, for checking expected system behavior from log files. LOGDIG is a generic framework, but it is motivated by logs that include temporal data (timestamps) and system-specific data (e.g. spatial data with coordinates of moving objects), which are present e.g. in Real Time Passenger Information Systems (RTPIS). The behavior mining in LOGDIG is state-machine-based, where a search algorithm in states tries to find desired events (by certain accuracy) from log files. That is different from related work, in which transitions are directly connected to lines of log files. LOGDIG reads any log files and uses metadata to interpret input data. The output is static behavioral knowledge and human friendly composite log for reporting results in legacy tools. Field data from a commercial RTPIS called ELMI is used as a proof-of-concept case study. LOGDIG can also be configured to analyze other systems log files by its flexible metadata formats and a new behavior mining language.

Keywords: Log file analysis, data mining, spatiotemporal data mining, behavior computing, intruder detection, test oracles, RTPIS, Python

1 Introduction

Log files are often the only way to identify and locate errors in deployed software [1], and especially in distributed embedded systems that cannot be simulated due to lack of source code access, virtual test environment or specifications. However, log analysis is no longer used only for error detection, but even the whole system management has become log-centric [2].

Our work originally started 15 years ago with a commercial Real Time Passenger Information System (RTPIS) product called ELMI. It included real-time bus tracking and bus stop monitors displaying time of arrival estimates. ELMI consisted of several mobile and embedded devices and a proprietary radio network. The system was too heterogeneous for traditional debugging, which led to log file analysis as the primary method. In addition, log file analysis helped discovering the behavior of some black-box parts of the system, which contributed to the development of open ELMI parts.

The first log tools were TCL scripts that were gradually improved in an ad-hoc manner. The tooling fulfilled the needs very well, but the maintenance got complicated

and the tools fit only for the specific system. This paper presents a new, general purpose log analyzer tool framework called LOGDIG based on the previous experience. It is purposed for logs that include temporal data (timestamps) and optionally system-specific data (i.e. spatial data with coordinates of moving objects).

Large embedded and distributed systems generate many types of log files from multiple points of the system. One problem is that log information might not be purposed for error detection but e.g. for business, user or application context. This requires capability to interpret log information. In addition, there can be complex behaviors like a chain of sequential interdependent events, for which simple statistical methods are not sufficient [1]. This requires state sequence processing. LOGDIG is mainly intended for *expected behavior* mining.

State-of-the art log analyzers connect the state transitions one by one to the log lines (i.e. events or records). LOGDIG has an opposite new idea, in which log events do not directly trigger transitions but events are searched for states by special state functions. The benefit is much more versatile searches and inclusion of already read old log lines in the searches, which is not the case in state-of-the-art. LOGDIG includes also our new Python based language Behavior Mining Language (BML), but this is out of the scope of this paper.

The new contributions in this paper are i) New method for searching log events for state transitions, ii) LOGDIG architecture and implementation iii) Proof-of-concept by real RTPIS field data.

This paper is organized as follows. Section 2 describes the related work and Section 3 the architecture of LOGDIG. Section 4 presents an RTPIS case study with real bus transportation field data. The paper is concluded in Section 5.

2 Related work

We focus on discovering the realized behavior from logs, which is required to figure out if the system works as expected. The behavior appears as an execution trace or sequential interdependent events in log files. The search means detecting events that contribute to the trace described as *expected behavior*.

A typical log analyzer tool includes metamodels for the log file information, specification of analytical tasks, and engines to execute the tasks and report the results [2]. Most log analyzers are based on state machines. The authors in [5] conclude that most appropriate and useful form for a formal log file analyzer is a set of parallel state machines making transitions based on lines from the log file. Thus, e.g. in [1], programs are validated by checking conformity of log files. The records (lines) in a log file are interpreted as transitions of the given state machine.

The most straightforward log analyzing methods is to use small utility commands like **grep** and **awk**, spreadsheet like Excel or database queries. More flexibility comes with scripts, e.g. Python (NumPy/SciPy), Tcl, Expect, Matlab, SciLab, R and Java (Hadoop). There are also commercial log analyzers like [3] and open source log analyzers like [4]. However, in this paper we focus on scientific proposals in the following.

Valdman [1] has presented a general log analyzer, and Viklund [5] analysis of debug logs. Authors in [6] have presented execution anomaly detection techniques in distributed systems based on log analysis. Execution anomalies include work flow errors and low performance problems. The technique has three phases: at first abstracting log files, then deriving FSA and next checking execution times. The result is a model of behavior. Later it can be compared whether the learned model was same or not than current model to detect anomalies. This technique does not need extra languages or data to configure analyzer. A database based analysis is presented in [7] for employing a database as the underlying reasoning engine to perform analyses in a rapid and lightweight manner. TBL-language is presented in [8] for validating expected behaviors in execution traces of system. TBL is based on parameterized patterns. Domain specific LOGSCOPE language is presented in [9] that is based on temporal logic.

LFA [10] has been developed for general test result checking with log file analysis, and extended by LFA2 [11; 12] with the idea of generating log file analyzers from C++ instead of Prolog. That improved general performance of LFA and allowed to extend the LFAL language by new features, such as support for regular expressions. Authors in [13] have presented a broad study on log file analysis. LFA is clearly closest to our work and most widely reported.

Table 1 presents comparison between our work and LFA/LFA2. The classification criteria is modified from [1] and for brevity we exclude detailed description of each. To conclude, the main difference to LFA/LFA2 is that it connects transitions directly to the events (lines) of the log files. LOGDIG has a completely different approach enabling more versatile searches, like the search from old (already read) lines or the search “backward”. New system-specific search features are also easier to add.

Table 1. Comparison of LOGDIG and LFA/LFA2 analyzer

Feature	LOGDIG	LFA/LFA2
1. Analyzed system and logs		
1. System type	Distributed	Distributed
2. Amount of logs	Not limited	1
3. size of logs	Not limited	Not limited
4. formats of logs	Not limited	Limited
2. Analyzing goals and results		
1. Processing	Batch job	Batch job
2. type of knowledge	Very complex	Complex
3. Results	SBK, RCL, test oracle	Test oracle
3. Analyzer		
1. technology or formalism of engine	State-machine	State-machine
2. using event indexing	Time, Id	No
3. Search aspects (SA)	Index, data, SSD	Data
4. Adjustment of accuracy of SA	Yes	Yes
5. pre-processing (and language)	Yes (PPL)	No
6. analyzing language	BML	LFAL
7. versatile reading from log	By index not limited	Only forward
8. operational modes	Multiple-pass	Single-pass
9. state-machine-based engine	Yes	Yes
1. state machines	1	Not limited
2. state-functions	ES algorithm	No
3. state transitions	Limited (F,N,E)	Not limited
4. state transition functions	Yes	Yes
5. variables	Yes	Yes
4. Analyzer's structure and interfaces		
1. implementation	Python (Tcl)	Prolog/C++
2. extensibility	Via SBK, BMS, SSC	?
3. integratability	Command line	?
4. modularity	BMU,ESU,SSC,SSD,lib	Lib
5. Analyzer's other features		
1. robustness (runtime)	Exit transition	Error transition
2. user interface	Command line	?
3. compiled or interpreted	Interpreted	Compiled
4. speed	Medium	Fast

3 LOGDIG architecture

Fig. 1 depicts the overall architecture and log processing of LOGDIG. The analysis process refines raw data from the log files (bottom) into information (middle) and finally to knowledge (top). The straightforward purpose is to transform the dynamic behavior data from the log files into **static behavior knowledge** (SBK file). This can be thought of linking many found events from many “rows” of log files into one “row” of the SBK file. The static knowledge can be reported and further processed by legacy visualization, database and spreadsheet tools like Excel. Besides SBK, LOGDIG produces a Refined Composite Log called RCL file. Its purpose is to link found data on the log files together into one composite, human readable text format described in BML language. This helps simple keyword-based searches for reporting and further analysis.

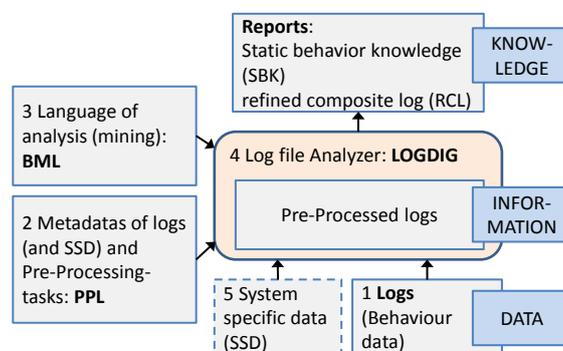


Fig. 1. Overview of LOGDIG analyzer log processing.

The LOGDIG processing has two main phases: log data pre-processing and behavior mining. Pre-processing transforms the original log files into uniform, mining-friendly format, which is defined by our own Pre-Processing Language (PPL). PPL is not discussed further in this paper, but we use the common `csv` format with named columns as output of the transform. The behavior mining phase uses our BML language to define searching the events of expected behavior.

Logs may consist of rows of unstructured text that can be parsed with regular expressions (Python). Timestamp is mandatory in every line. Pre-processed logs are structured in rows and columns as `csv` files. The first row is header line that includes column (variable) names. The columns are row number, timestamp and log file specific data.

Sometimes analyzing needs systems specific utility data to make decisions in behavior mining phase. This is called System Specific Data (SSD) and it is not log data. SSD files can come from system documentation and databases without any standard structure and format as in our case study. SSD can be e.g. spatial data like position boundary boxes for bus stops in RTPIS. SSD is parsed using System Specific Code (SSC) written in Python.

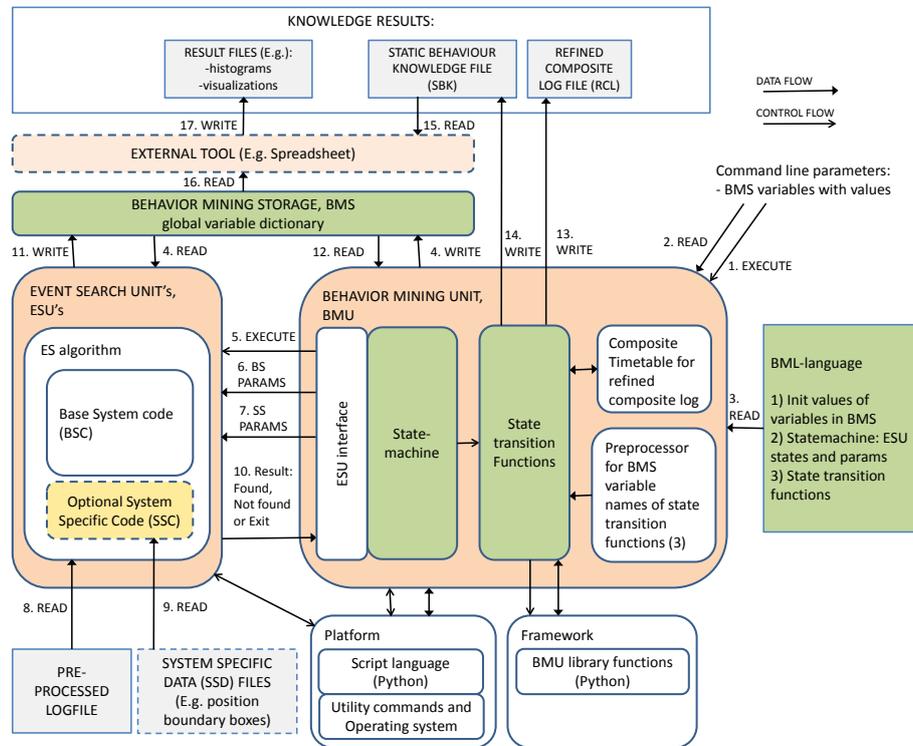


Fig. 2. Detailed architecture of behavior mining phase of LOGDIG.

The detailed behavioral mining process is presented in Fig. 2. We first describe the concepts and functional units. There are two main functional units: Behavior Mining Unit (**BMU**) and Event Search Unit (**ESU**) and also one global data structure named as Behavior Mining Storage (**BMS**). There can be many ESU's, one BMU and one BMS in LOGDIG.

BMS is a global and common dictionary type data structure, which includes variable names and their values used in LOGDIG. Initial BMS can be described in BML-language or given in command line parameters. It is possible to add new variables and values to BMS on the fly. Used variables can be grouped as follows: command line, internal, and result variables. The result variables are grouped as identify, log, time, searched, calculated, counter, error flags and warning flag variables (See example in table 2). BMS can be implemented as shared memory structure wherein it is very fast and it can be used in many (external) tools and commands even in real time.

BMU consists of five parts: pre-processor of variable names, state-machine, state-transition functions, ESU interface and composite timetable. **Pre-processor** converts variable names in state transition functions of a BML code to be used in Python. State machine is the main engine of BMU that is described in BML language. State transi-

tion functions are called from the state machine and they are also described in BML-language.

ESU interface connects the state machine to ESU. It converts execute command and parameters suitable to ESU and converts the result of ESU for the state machine. The ESU interface can be modified depending on how ESU has been implemented and where it is located, compared to BMU. ESU can be a separate command line command and it can be even located on separate computers. This gives interesting possibilities for the future to decentralization and performance.

A **composite timetable-structure** is for the Refined Composite Log (RCL) file to collect together all time indexed “print”-notifications from state transitions. The timetable is needed because all searched data from log files are not always read in time (forwarding) order, because BMU has capability to re-read old lines from same log and also to read older line from other logs.

BML language presents the expected behavior with certain accuracy and it is straightforward representation of state machine, (ESU) states with input parameters and state transition functions. It can also include initialization of BMS variables and values. State transition functions can include Python code and BMS variables. Because BML is out of the scope of this paper, we do not go into details.

ESU is a state of the state machine and it includes state function which is named Event Search (ES) algorithm. ES algorithm includes a Basic System Code (BSC) and optionally a System Specific Code (SSC). The BSC is the body of ES algorithm and SSC is an extension to BSC. BSC reads Basic System (BS) parameters from the state machine and on their basis start searching from log. If SSC is in use (set in mode parameter), also System Specific (SS) parameters are read.

Base System (BS) parameters are needed to set the search mode, log file name expression, searched BMS log variable names, time column name, as well as start and stop time expressions for ES algorithm. **System Specific (SS) parameters** are used in SSC-part of ES algorithm. E.g. in case of spatial data like in our case study, SS parameters are: latitude and longitude column name and also SSD filename expression.

Each search mode has the common task: searches an event between the start and the stop time (ESU time windows in figure 5) that values of variable names are same in log and BMS.

The search modes can be:

- “EVENT:First”: Searches the first event from the log
- “EVENT>Last”: Searches the last event from the log

In the case of SSC, search modes can also be System Specific (SS) modes (as in our case study):

- “POSITION:Entering”: Searches object’s entering position to boundary box area (described in SSD) from the log
- “POSITION:Leaving”: Searches object’s leaving from boundary box area (described in SSD) from the log

3.1 Mining process

The mining process is described in figure 2. The user of LOGDIG starts the BMU execution (1), which reads command line parameters (2) and BML language (3) as input. On their basis BMU writes (4) BMS variable values directly or via state transition function, and executes (5) ESU with search BS parameters (6). If SS mode is in use in current ESU, also SS parameters (7) are read. Then ESU reads (8) pre-processed log file. If SS mode in use, it reads also (9) SSD file. Then starts the searching.

After ESU has finished the searching, it returns (10) the result of the search, which can be “Found” (F), “Not found” (N) or “Exit” (E). If “Found”, ESU writes (11) found variables from the log file to the BMS variables. Then, BMU reasons what to do next based on the action attached to the result. This is described in BML language. Depending on the action, BMU runs possible state transition function which can read (12) BMS variables. Based on the action, a transition function may write one row to (13) RCL or (14) SBK **knowledge-result** files.

After that BMU normally executes (5) a new search by starting ESU with new parameters. BMU exits the mining phase if there are no searches left or “Exit” has been returned. “Exit” is a problem exception that should never happen, but if it occurs, user is notified and LOGDIG is closed cleanly.

When BMU has completed, the SBK file can be read (15) by an **external tool** (like spreadsheet) to write (17) better visualizations from the knowledge results. All tools supporting the csv format can be used. Another option is to read (16) directly data from the BMS variables by a suitable tool to write (17) other (visualization) knowledge-results.

State transition functions can use **framework BMU library** functions, like setting and writing RCL and SBK files and also calculating time differences between timestamp. It is also possible to add new library-functions depending on the needs.

Because LOGDIG analyzer has been implemented in Python and it works on top of any operating system, all features of Python and the operating system can be used as a **platform** in BMU and ESU.

4 Case study: EPT-case in ELMI

The real time passenger information system ELMI [14], deployed in Espoo, Finland in 1998 - 2009, displayed the waiting time of buses at bus stop monitors. ELMI included 300 buses and 11 bus stop monitors [15], 3 radio base stations, central computer system (CCS) and communication server (COMSE). Buses sent login of line and location information to CCS via radio base stations every 30 seconds. Then CCS calculated the waiting time values and sent them to the bus stop monitors via COMSE. Because all messages of the system went through the COMSE, it was also used for logging and running the LOGDIG analyzer. There are 3 types of original logs: CCS, COMSE and BUS. Every bus has own BUS log identified by bus number. In pre-processing phase of LOGDIG, original logs are divided message-specified log files: CCS_RTAT, CCS_AD, BUS_LOGIN and BUS_LOCAT. That makes analyzing faster and simpler.

We have named the main requirement as EPT (Estimated Passing Time) to see when the bus pass the stop. The *expected behavior* can include one or more EPT's.

Other definitions include RTAT (Real bus Theoretical Arrival Time), including bus number, line and monitor, as well as AD (Advanced or Delayed) compared to RTAT.

The sequence of requirements for one EPT case is: 1) a bus driver sent a login for a line in a bus terminal, 2) the bus started to drive from the terminal, 3) when the bus is in the line and if necessary (started, delayed or advanced of schedule) CCS calculated estimated waiting times and sent to target monitors, 4) the bus arrived to a bus stop and its monitor, and 5) COMSE removed the waiting time from monitor.

There are two SSD's (System Specific Data) in this case: 1) customer's requirement document that requires the error of estimation should be below 120 seconds and 2) boundary box areas of bus stop's from database of ELMI-system.

Various EPT cases are processed by the LOGDIG state machine shown in Fig. 4. It can search and analyze all EPT's in a time window of Expected Behavior (EB) for specific lines and monitor as described in Fig. 3. The time window can be set from the command line or BML language.

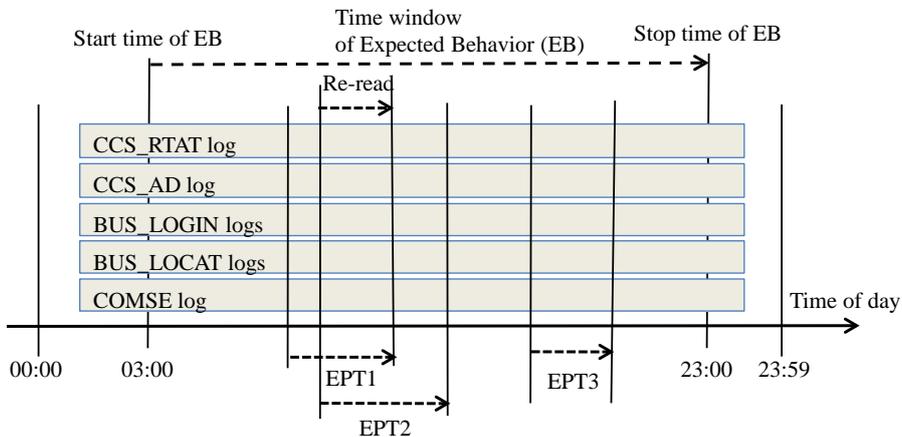


Fig. 3. Time window from 03:00 to 23:00 and 3 example EPT's in logs

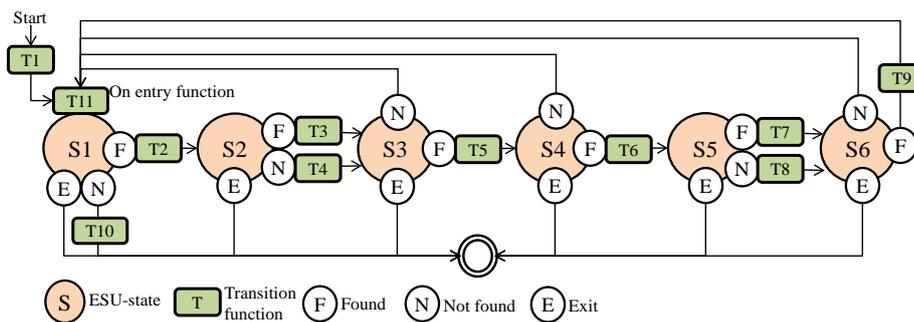


Fig. 4. State machine structure of the EPT's case study

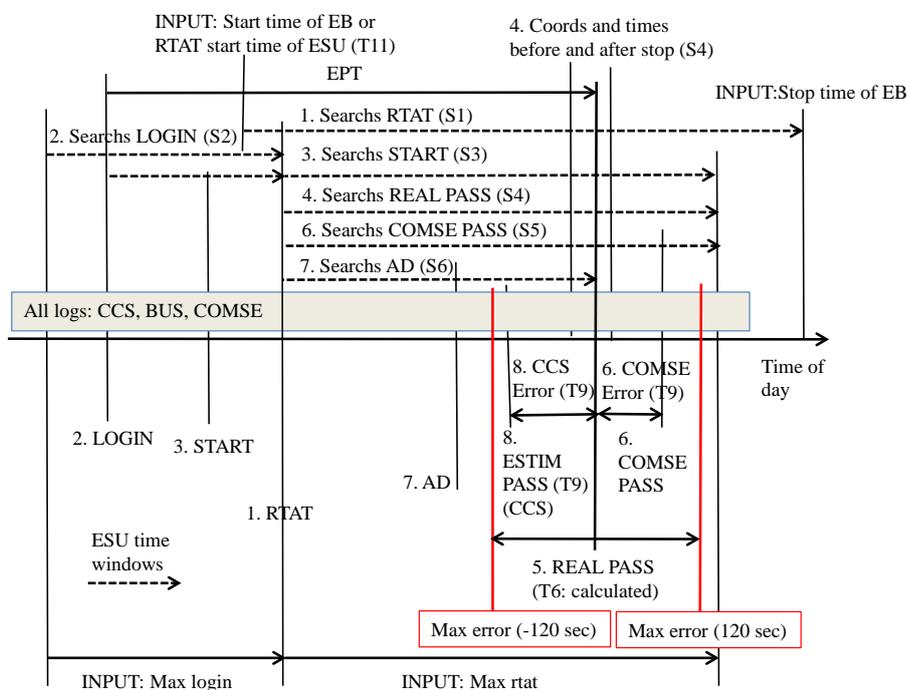


Fig. 5. Timing diagram for one EPT

Behavior and ESU-states of the state machine of LOGDIG in EPT's case is depicted in Fig. 5. Inputs of the EPT's case (from T1 transition function or command line) are start time, stop time, line, max login (means maximum time from LOGIN to RTAT) and max rtat (means maximum time from RTAT to PASS) and also maximum error values (SSD 1: -120 – 120 seconds) for the estimations of CCS and COMSE. The sequence is following:

- S1: Searches first RTAT message (inputs: line) from CCS_RTAT log file in given ESU time window (see Fig. 5):
 - On entry function (before searching), calls T11: Sets start time of ESU time window to search next RTAT message or original EB start time in first “round” of search
 - Found: Sets RTAT variables. Calls T2: Sets input variables for next state.
 - Not found: Calls T10: Prints final results.
- S2: Searches first LOGIN message (inputs: log-type, bus, line, direction) from BUS_LOGIN log file in given ESU time window:
 - Found: Sets LOGIN variables. Calls T3: Sets input variables for next state.
 - Not found: Calls T4: Sets input variables for next state.
- S3: Searches starting (leaving) place of the bus from terminal bus stop in given ESU time window from BUS_LOCAT log file (input: bus). This needs SSD 2 to check positions:

- Found: Sets LOCAT variables. Calls T5: Sets input variables for next state.
- Not found: -
- S4: Searches arriving place of the bus to the target bus stop in given ESU time window from BUS_LOCAT log file. This needs SSD 2 to check position:
 - Found: Sets LOCAT variables. Calls T6: Sets input-variables for next state. Calculates the real passing time of the bus
 - Not found: -
- S5: Searches first BQD message (inputs: bus, line) from COMSE log file in given ESU time window:
 - Found: Sets BQD variables. Calls T7: Sets input variables for next state.
 - Not found: Calls T8: Sets input variables for next state.
- S6: Searches last AD message (inputs: line, direction, bus) from CCS_AD log file in given ESU time window:
 - Found: Sets AD variables. Calls T9: Calculates errors of the estimated waiting time, writes one row to SBK file and writes RCL knowledge from the composite timetable structure to RCL file
 - Not found: -

There are exit transitions in every ESU state if a problem reading a log file occurs, e.g. the file is missing or there is a syntax error. Variables and their values (BMS variables in Fig. 2) comes directly from the log files and the command line initialization parameters, but there are also result specific variables in the SBK files. Result variables of SBK file are set in almost every state and state transition function and they are added (by one row) to SBK file at the end of successfully analyzing of one EPT in the last transition (T9 in Fig. 4). RCL knowledge is written in almost every state transition function to the composite timetable-structure (see Fig. 2) and in T9 they are added to RCL file. Searching time windows in every state can vary depending on (initializing variables and) variable values (found events timestamps) given from previous states.

Time indexing and possibility to read already processed log lines is utilized in two ways. See the re-read between EPT1 and EPT2 in Fig. 3. When searching different log files, RTAT-message from CCS_RTAT log is read before LOGIN message from BUS_LOGIN log even though LOGIN (EPT requirement step 1) comes before RTAT (EPT requirement step 2) in time. This way we can limit the complexity of the search and analyze only certain RTAT and its line or monitor. Within the same log file, RTAT or AD messages in many EPT cases exist. If we want to analyze only one monitor and its line, we will re-read the old log lines.

4.1 Case study results

The content of the SBK file in the ELMI case study is given in Table 2. There are described three example EPT's (bus drives in line 2132 and towards bus stop monitor 1031), which are also shown in Fig. 6. We consider a more detailed reading of the first EPT (EPT1). At 6:24:31 (LOGIN_MSG) in the terminal, the bus driver (number111) has logged in to line 2132 and its direction 1. Then at 6:25:01 (DRIVE) the

bus has started from the terminal. At 6:42:33 (RTAT_MSG) CCS has sent the first estimated arrival time (6:45:52, RTAT_VAL) to the monitor. At 6:45:03 (AD_MSG) CCS has sent the last time correction for the waiting time. AD_VAL -25 tells that the bus has been 25 seconds in advance of the schedule. At 6:45:40 (PASS) the bus has passed the bus stop that is the estimation of LOGDIG. COMSE has estimated the arrival time to be 6:46:02 (BQD_MSG). That means there has been 13s error (PASS_TIME_ERR) between the estimated arrival time (RTAT_AD_VAL: 6:45:27) and real arrival time (PASS: 6:45:40). The error to the COMSE estimation has been -22 seconds (BQD_TIME_ERR). Because absolute values of the errors are smaller than 120 seconds, error flag EPT_ERR is 0. Warning flags are 0, that means there have been found LOGIN and BQD messages from the logs. In this case the whole EPT has lasted from 6:24:31 (LOGIN_MSG) to 6:46:02 (BQD_MSG) in total of 21:31 minutes.

Table 2. Three example EPT's (in line 2132 and bus stop monitor 1031) of the SBK file

Knowledge variables			Knowledge data		
Group	Num	Variable	EPT1	EPT2	EPT3
Identify	1	SBK_ID	13	35	52
Log	2	BUS	111	451	228
	3	LINE	2132	2132	2132
	4	DIR	1	1	1
	5	SIGN	1031	1031	1031
Time	6	RTAT_SRCH_TIME	6:36:01	7:55:47	8:48:08
Searched	7	LOGIN_MSG (S2)	6:24:31	7:48:17	8:43:09
	8	DRIVE (S3)	6:25:01	7:48:47	8:44:10
	9	RTAT_MSG (S1)	6:42:33	7:55:47	8:48:08
	10	RTAT_VAL (S1)	6:45:52	7:59:02	8:51:18
	11	PASS (S4)	6:45:40	8:01:57	8:51:45
	12	BQD_MSG (S5)	6:46:02	8:02:17	8:52:08
	13	AD_MSG (S6)	6:45:03	8:01:48	8:51:09
	14	AD_VAL (S6)	-25	166	11
Calculated	15	RTAT_AD_VAL (T9)	6:45:27	8:01:48	8:51:29
	16	PASS_LOGIN (T9)	1269	820	516
	17	PASS_DRIVE (T9)	1239	790	455
	18	PASS_RTAT (T9)	187	370	217
	19	PASS_TIME_ERR (T9)	13	9	16
	20	BQD_TIME_ERR (T9)	-22	-20	-23
Error flags	21	EPT_ERR	0	0	0
Warning flags	22	LOGIN_WARN (T4)	0	0	0
	23	BQD_WARN (T8)	0	0	0

The same EPT's has also been presented in a visual form in Fig. 6. The visual presentation has been generated from the SBK file using gnuplot and Tcl script language. The horizontal axis represents the time of day and the vertical axis time differences in relation to the passing time. The values can be directly found from the SBK variables numbers: 16-20.

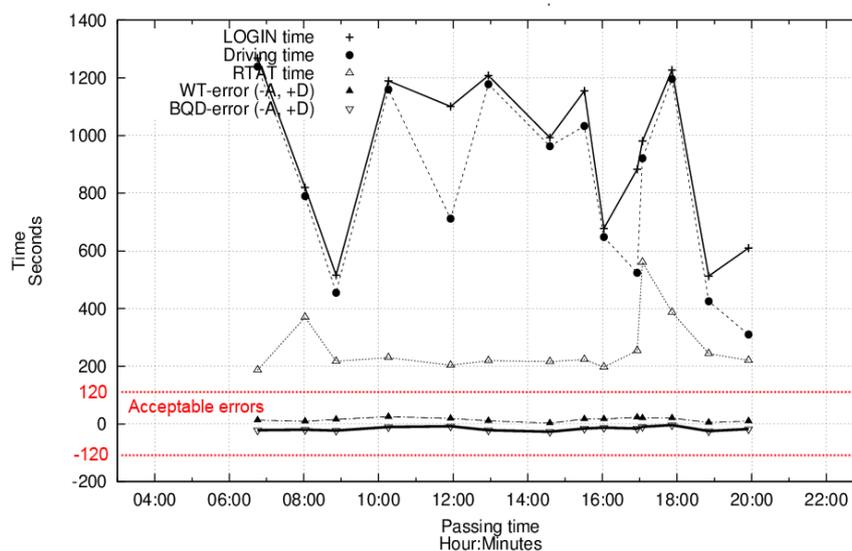


Fig. 6. Example of visual representation of SBK-file (bus line 2132 and monitor 1031)

We can reason a few things in Fig. 6. Times between 14 EPT's have been varied a lot. For example "LOGIN time" in about 8:00 – 9:00 and 16:00 - 17:00 seems to be smaller. These times are typically morning and afternoon rush hours. We see also that real waiting time values have been quite short time (RTAT time: 200s) in monitors, but that is because in this case the trip from the terminal bus stop to the monitor was quite short. Estimation of waiting times are seemed to work quite well, because error values (WT-error and BQD-error) are between +120 - -120 seconds in all EPT cases. These were quite normal results. Additionally traffic jams or other problems can also be easily see from the visual presentation like that.

All detected errors can later be explored statistically from SBK file to get more detailed information of causes of errors. For example bus, line and sign (bus stop) specific errors such as the following error percentages:

BUS,463	=	1 / 24 = 4.2 %
BUS,53	=	1 / 29 = 3.4 %
LINE,2132 DIR 1	=	1 / 42 = 2.4 %
LINE,2132 DIR 2	=	4 / 139 = 2.9 %
SIGN,1033	=	1 / 62 = 1.6 %
SIGN,1061	=	3 / 77 = 3.9 %

These kind of results gave valuable “feedback” knowledge to understand the real behavior of the system. The results also helped to fix bugs and behavior of the system and thus improve the quality of the system.

The execution time of analyzing of one bus line (2132 in this case study) was about 15 seconds using normal PC. There was at all 181 EPT's in the analysis. Other EPT's were for other bus stop monitors. The execution time depends on the amount of daily log data and analyzed bus line. There have been three key bus lines and its monitors used for analyzing the ELMI system.

LOGDIG can also be used to detect anomalies, like work flow error or low performance of the execution. For example, the work flow error can be checked by structure of the state machine, and the low performance by time window limits of the ESU's.

5 Conclusion

We have introduced the LOGDIG analyzer framework that is capable to analyze very complex expected behavior from the logs. LOGDIG was motivated by the fact that there were no other tools flexible enough, even though LFA is very close to the needs. Visualizations and other statistical post-processing have been left out, since there are lots of them available.

LOGDIG is best suited for complex behavior problems, since the setup takes some time compared to e.g. simpler command line search tools. To improve the performance, the Python implementation can be further optimized or even written in compiled language, and the overall execution distributed. If the source log files are available in different machines, the ESUs can be distributed directly there.

In the future, LOGDIG could be used as a higher-level analyzer that uses lower level analyzers. For example, the event search algorithm can be replaced by e.g. LFA/LFA2. To conclude, LOGDIG fulfills the requirements for RTPIS kind of applications, and offers a flexible framework for other domains.

6 References

1. Valdman, J. Log file analysis. Technical report, Department of Computer Science and Engineering, University of West Bohemia in Pilsen (FAV UWB), Czech Republic, , Tech.Rep.DCSE/TR-2001-04 (2001) pp. 1-51.
2. Oliner, A., Ganapathi, A. & Xu, W. Advances and challenges in log analysis. *Communications of the ACM* 55(2012)2, pp. 55-61.
3. Jayathilake, D. Towards structured log analysis. *Computer Science and Software Engineering (JCSSE), 2012 International Joint Conference on, 2012, IEEE.* pp. 259-264.
4. Matherson, K. Machine Learning Log File Analysis. Research Proposal (2015).

5. Viklund, J. Analysis of Debug Logs. Master of Science. 2013. Luleå University of Technology, Department of Computer Science, Electrical and Space Engineering, Sweden. 1-39 p.
6. Fu, Q., Lou, J., Wang, Y. & Li, J. Execution anomaly detection in distributed systems through unstructured log analysis. Data Mining, 2009. ICDM'09. Ninth IEEE International Conference on, 2009, IEEE. pp. 149-158.
7. Feather, M.S. Rapid application of lightweight formal methods for consistency analyses. Software Engineering, IEEE Transactions on 24(1998)11, pp. 949-959.
8. Chang, F. & Ren, J. Validating system properties exhibited in execution traces. Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering, 2007, ACM. pp. 517-520.
9. Groce, A., Havelund, K. & Smith, M. From scripts to specifications: the evolution of a flight software testing effort. Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 2, 2010, ACM. pp. 129-138.
10. Andrews, J.H. & Zhang, Y. General test result checking with log file analysis. Software Engineering, IEEE Transactions on 29(2003)7, pp. 634-648.
11. Aulenbacher, I.L. Master of Science, Generating Log File Analyzers, The University of Western Ontario, London Ontario Canada (2012) pp. 1-88.
12. LEAL-AULENBACHER, I. & ANDREWS, J.H. Generating C Log File Analyzers. WSEAS Transactions on Information Science & Applications 10(2013)10.
13. Andrews, J.H. & Zhang, Y. Broad-spectrum studies of log file analysis. Proceedings of the 22nd international conference on Software engineering, 2000, ACM. pp. 105-114.
14. Aaltonen, J. Implementation of GPS based real time passenger information system. Licentiate in Technology. 1998. Tampere University of Technology. 1-76 p.
15. Heikkinen, E. Informaatiotaulun protokollakortin ohjelmisto. Master of Science. 1996. Tampere University of Technology. 1-68 p.

Mining Knowledge on Technical Debt Propagation

Tomi ‘bgt’ Suovuo, Johannes Holvitie, Jouni Smed, and Ville Leppänen

TUCS – Turku Centre for Computer Science,
Software Development Laboratory &
University of Turku,
Department of Information Technology,
Turku, Finland
{bgt, jjholv, jouni.smed, ville.leppanen}@utu.fi

Abstract. Technical debt has gained considerable traction both in the industry and the academia due to its unique ability to distinguish asset management characteristics for problematic software project trade-offs. Management of technical debt relies on separate solutions identifying instances of technical debt, tracking the instances, and delivering information regarding the debt to relevant decision making processes. While there are several of these solutions available, due to the multiformity of software development, they are applicable only in predefined contexts that are often independent from one another. As technical debt management must consider all these aspects in unison, our work pursues connecting the software contexts via unlimited capturing and explanation of technical debt propagation intra- and inter-software-contexts. We mine software repositories (MSR) for data regarding the amount of work as a function of time. Concurrently, we gather information on events that are clearly external to the programmers’ own work on these repositories. These data are then combined in an effort to statistically measure the impact of these events in the amount of work. With this data, as future work, we can apply taxonomies, code analysis, and other analyses to pinpoint these effects into different technical debt propagation channels. Abstraction of the channel patterns into rules is pursued so that development tools may automatically maintain technical debt information with them (the authors have introduced the DebtFlag tool for this). Hence, successfully implementing this study would allow further understanding and describing technical debt propagation at both the high level (longitudinal technical debt propagation effects for the project) and the low level (artifact level effects describing the mechanism of technical debt value accumulation).

1 Introduction

Technical debt is a software development concept that is interested in exposing asset management characteristics for project trade-offs [5]. Working with scarce resources to fulfill ever-changing requirements, software projects often need to

emphasize certain development driving aspects over others, such as delivery deadlines over thorough documenting. Further, invalid or lacking knowledge on certain aspects of the development may lead to emphases made that improperly reflect the actual situation. In both cases the informed and uninformed decisions result to trade-offs that accumulate technical debt [13].

It has been argued [16] that a key factor for the adoption of technical debt management into software development is the capability to produce and maintain technical debt information within the project. That is, the project trade-offs must be identified, their distribution and effects defined, and this information must be maintained to reflect the true software project state. Undoubtedly, failures in the information delivery result in unmanaged technical debt, or decisions being made based on outdated information, both of which, implicitly or explicitly, affect the project.

Technical debt research has been proficient in suggesting identification, tracking, and governance solutions to overcome the technical debt information production issues [12]. The problem is that while solutions have been proposed and trialed on various software contexts, no prior research has properly investigated the whole software context space. That is, identifying and classifying where and how technical debt exists and how does it propagate intra- and inter-software-contexts. This higher level structure may be described in some studies as the concept of technical debt interest and its accumulation, but it has not been explicitly examined; being less important to the relevant studies' goals. Arguably, however, in order to make technical debt management applicable, the various solutions must function together, and in this the enabling factor is technical debt propagation.

Today, the software projects that plug into social media services through APIs (Application Programming Interface) are an exemplar field of software context versatility. Updates to these APIs, invoked by their external authors, indicate sources of technical debt accumulation and propagation in their clients', often business critical, software. Mining Software Repositories (MSR) for the clients that are subject to these updates enables studying the software context space to address the cap in technical debt propagation knowledge.

In the 1980s software applications were relatively simple and they were delivered as is. They were relatively bug free and needed no updates. Once an application was released, any existing technical debt was outside the organization's control. As software grew increasingly complex, especially with the emergence of the Internet in the 1990s, bigger applications were released with more issues remaining. The practise eventually turned out having regularly released patches as a norm, as they were also easily distributed through the net. Technical debt was feasible and also realized. Now, in the 2010s we have complex applications that not only utilize third party libraries, but also third party services through APIs. There are regular updates to the libraries and the APIs, as well as to the client applications themselves. These all are sources of technical debt. Further, as previously shown [6], a singular technical debt instance rarely limits to a single software development component but rather spans over multiple (e.g., design,

implementation, and testing), making the emerging debt even more cumbersome to track.

Our intention is to understand the technical debt propagation context by investigating the latest trends: use of external APIs and especially those of social media services. The paper is structured as follows: we begin in Section 2 by reviewing the background. Section 3 builds on this and introduces our technical debt propagation research objectives. We introduce our approach to overcome the objectives and initial results in Section 4. The concluding remarks appear in Section 6.

2 Background

We will introduce here related work regarding technical debt, propagation in the software context, and APIs. Whilst defining core concepts for the article's foundation, empirical work is also visited so as to further understand the state of current research.

2.1 Technical Debt and Its Propagation

The term “technical debt” was initially coined by Ward Cunningham [2]. In his experience report, releasing code was paralleled to going into debt: trade-offs are made in the software project to meet a deadline, and these trade-offs can be considered debt that should be paid off when resources permit. Until the debt is paid off, it will incur interest payments—that is, later work in the project must accommodate the inoptimalities resulting from the trade-offs. This description has remained applicable to these days. Later revisits to the definition have mainly captured dimensions that further explain the role of the debt in the project: McConnell [13] provides a definition for intentional and unintentional technical debt, while Brown et al. [1] give a further description of the debt's effects via reflection to the financial domain and discussion on the resolution probability.

Firstly, McConnell [13] provided a definition for the intentionality behind the debt: intentional debt is a trade-off made whilst fully aware of its consequences, an investment with an expected return. Unintentional debt on the other hand is accumulated due to, for example, lack of knowledge. This type is a cause for concern as it remains unmanaged until discovered. Secondly, Brown et al. [1] gave a further description of the debt's effects via reflection to the financial domain: the earlier trade-offs accumulate interests payments manifesting as increased future costs, and trained decisions should evaluate if paying the interest is more profitable over reducing the loan via refactoring. Differing from the financial domain, here, the debt's interest has a probability that captures if the trade-off will have visible effects on future development: debt within a software artifact that will not be visited has a realization probability of zero.

Management of technical debt requires that we are capable of identifying and tracking the trade-offs, the atomic instances, that form the debt for a project.

Without this information readily available, trained decision regarding the debt's governance cannot be made [16]. The software context, however, makes the identification, and especially, the tracking an arduous task: instances of technical debt can span over multiple development phases and the most affected part is the software implementation [6] which arguably grows exponentially complex in the future through various abstraction layers and techniques. Nevertheless, the tracking should be able to follow a technical debt instance in this context.

From the latest systematic mapping study on technical debt [12] we can see that several solutions for tracking technical debt are available. However, we also observe (see Figure 10 in [12]) that there are areas in the software development context that are not covered by any solution; whilst most of the solutions cover sub-contexts focusing on predefined environments and specific parts of the software life-cycle. Furthermore, from Kruchten et al. [10] and Izurieta et al. [7] we can see that the causes for technical debt are various and they can be described using various characteristics. We consider all these findings indicative of the multiformity of the context of technical debt in software projects. Thus, in addition to searching for solutions in this context, technical debt research should pursue mapping the full context space and an understanding of technical debt's value in it.

Lastly, we note that technical debt tracking is the process of indicating technical debt propagation in the software context. To this end, the authors identify only the work by McGregor et al. [14] to explicitly address this issue. Here, considering mainly the software implementation, they note that technical debt for a new software asset is affected by the technical debt in relied upon assets, the amount of abstraction layers may diminish the amount of technical debt that propagates, and, in another scenario, rather than being directly accumulated from integrated assets, the technical debt has an indirect effect on the asset's users—for example, by making adoption more difficult.

2.2 Software Change Analysis

What is pursued herein is a better understanding of the context of technical debt propagation in software. We argue that *software change* should be considered the fundamental unit for this. Something that Schmid [15] also considered core to technical debt modelling during software evolution. Capturing software changes and distinguishing between technical debt inclined and other changes (that is, changes using information relatable to technical debt properties described by Brown et al. [1] and discussed in Section 2.1, and changes with no such properties) would allow non-restricted observation of technical debt in the full software context. Identifying software change retrospectively for projects corresponds to Mining Software Repositories (MSR).

Kagdi et al. [8] produce a taxonomy on MSR techniques, defining software change as “*the addition, deletion, or modification of any software artifact such that it alters, or requires amendment of, the original assumptions of the subject system.*” Here, a source code change is indicated as the fundamental unit for

software evolution, but as the causes [10, 7] and the manifestations [6] for technical debt do not limit to the implementation, we adopt *software change* as the fundamental unit.

In this work, the mining efforts focus on large open-source, social-networking-enabled, repositories in order to maximally cover the diversity of software change. Tsay et al. [18] note that in GitHub handling of pull-requests is affected by social factors: highly discussed requests enjoy a lower acceptance rate, while submitters relations to—especially the manager of—the accepting project increases acceptance; this is supported also by [3]. Kalliamvakou et al. [9] survey GitHub as a MSR target. They conclude that the repository gives solid data on basic project properties, such as program language use, but synthesizing more abstract conclusions requires careful assessment. The main cause for concern here is GitHub’s utilization as infrastructure for personal projects. This form of usage vastly deviates from others. To counter this bias, Kalliamvakou et al. [9] suggest considering only projects with more than two authors and demonstrated activity in both commit and pull requests.

3 Seeking Technical Debt Knowledge

In the following we address our ongoing technical debt propagation research on two distinct levels: the inter-dependency effects at the software artifact level and the longitudinal effects at the project level.

3.1 Inter-Dependency Effects within Software Artifacts

As discussed in Section 2, a multitude of solutions exist for both identifying and tracking technical debt. However, most of the solutions are intended for pre-defined software development contexts; for example, limiting their use to a specific sub-set of implementation techniques and herein, during continued software development, to certain mechanisms for technical debt propagation.

However, the ability to produce exhaustive technical debt information requires that all possibilities for technical debt propagation are acknowledged. We postulate, based on the properties of technical debt identified by Brown et al. [1] and to the average cover of single technical debt instances queried by Holvitie et al. [6], that the propagation “stream” for technical debt is capable of leaving the current host technique and merging into others. This is indicative of several sub-areas within technical debt research.

Foremost research area for technical debt propagation in software artifacts, is (1) to show that technical debt propagates between software components that can exist in external and independent projects and be implemented using different technologies. The interest and even the whole initial debt can be created in an external, but linked project that is worked by another team. The works referred here do not dispute this information, and may even implicitly assume this, but it is important to recognize this phenomenon explicitly and have quantitative research conducted on it to indisputably point it out.

Second research area, partially reliant on the first, is (2) to accumulate a documentation that describes the possible ways in which technical debt can propagate. Preferably, this would be a taxonomy capturing the unique propagation channels for technical debt. Finally, in order to enable information delivery for technical debt management purposes, (3) the channel descriptions must be enriched with information regarding technical debt value accumulation for all unique accounts of propagation. This would enable, possibly automated, technical debt information maintenance as the taxonomy is capable of tracking and valuating technical debt through out the software project.

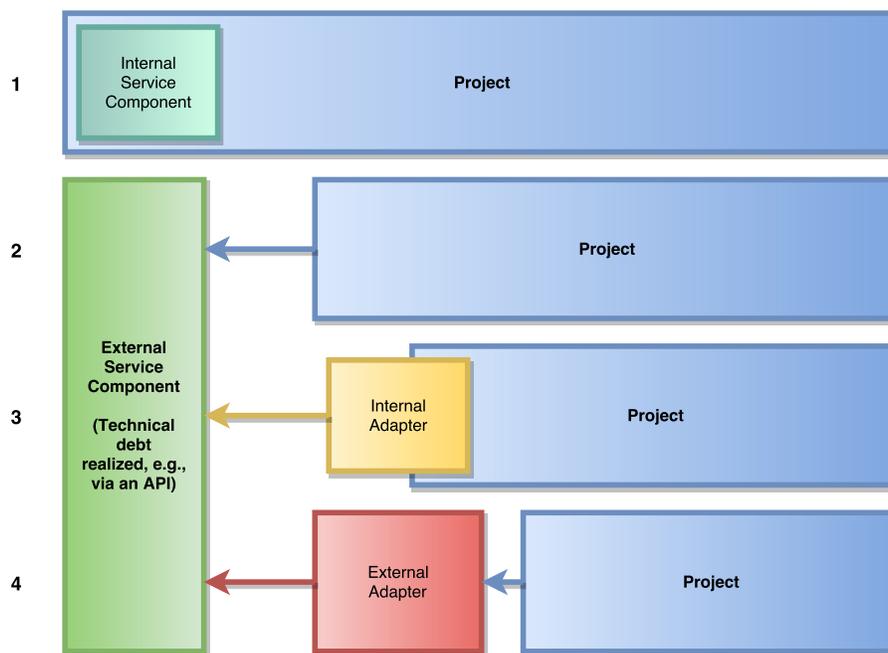


Fig. 1: Coarse classification for different *chains of projects* (COP)

3.2 The Chain of Projects

One way to identify the propagation of technical debt is to make longitudinal studies of increased debt in different phases of a project and connect them with the root causes. Technical debt can be identified as matters, such as discovered vulnerabilities, updates, and feature discontinuation in systems related to the project. Also, adding a new feature in a utilized external service API may cause technical debt when the project customer wants the new feature implemented in the project. We can identify different propagation paths by following how such

an event causes extra work in the *chain of projects* (COP) that are all linked with each other.

If an API is not interfaced directly but through a third party library, it may be that the customer is not happy to wait until the library is updated with the new feature. This will cause the project debt to be paid by implementing this new feature quickly with an internal solution. This will become a new kind of a debt, from the opposite end of the COP, when the referred library is finally updated. Here, the internal solution becomes legacy and requires refactorization into a solution that utilizes the library again, for example, in accordance to the coding conventions followed by the programming team.

There are cross waves moving back and forth in the COP from the root cause, through the library, to the end of chain application. These can be tracked by following the amount of increased work in each area.

Figure 1 demonstrates a sample classification for COPs. Here, case **1** demonstrates a monolith project that has internally implemented services with no outside dependencies. This is a classical, and probably the most studied, scenario for technical debt management, where the debt is only internally caused, felt, and managed. Cases **2** through **4** depict more modern scenarios, where the projects depend on external service providers. In case **2**, the project has a direct dependency to the service and adapts explicitly and directly as invoked by the service. A slightly dampened version, but still fully managed by the project organization is presented in case **3**, where the project, possibly alongside with the organization's other projects, uses an internally produced adapter to access the service. Hence, the project itself does not directly feel changes in the external service, but adaptation to them is still managed internally. Finally, in case **4** the project uses an external adapter to access the service. The external adapter generally serves a broader range of projects and hence is not customized for the needs of specific projects. On the other hand, external adapters tend to retain compatibility as long as possible which dampens change speeds invoked by the external service.

The classification in Figure 1 is especially important from the viewpoint of distinguishing between the “noisy” and the technical debt inclined software changes, as the monolith projects of similar size can be used as the baseline when studying how the external service invokes and propagates technical debt. Further, as per the previous description, it can be expected that the invoked technical debt will propagate quicker in the directly dependent cases than in the indirect cases **2** to **4**.

4 Exploiting Open-Source Projects

Exploiting open source code repositories enables us to make longitudinal surveys of the history. The GitHub code repository service ¹ appears as a treasure trove for this kind of research. We can take a project from GitHub, and we can find for it, neatly logged, each change and its date with great detail.

¹ See <https://github.com/>

GitHub gives an open access to several different projects. However, there is also an option of hosting private projects for premium users as mentioned in Section 2.2. With only the public access to the repositories, the sample is likely to be biased. This means that traditionally non-disclosed for-profit projects cannot be found in GitHub like this, which entails that a lot of professional work is not covered by this study. However, it can be argued that functionality is delivered via the same technologies in closed-source projects.

Furthermore, regarding mapping the *software change* (as discussed in Section 2), the GitHub API gives an easy access to byte-wise size of source files and line-wise size of code change per commit. Through this we have the scale of the whole project in bytes, but the scale of changes in lines of code. Optimally both variables would be measured identically, but we can only rely on these two measures being sufficiently comparable. The only other option would be to go through the source files and count the line breaks outside the GitHub API support.

As elaborated in Section 3, we want to observe the propagation of technical debt on both at the software project and the software artifact levels, and with as little constrain as possible so as to capture the propagation context as complete as possible. Herein, we face the problem of how to identify technical debt in a highly diverse setting, and this is the reason why we emphasize the novelty of researching open-source social-networking-enabled projects.

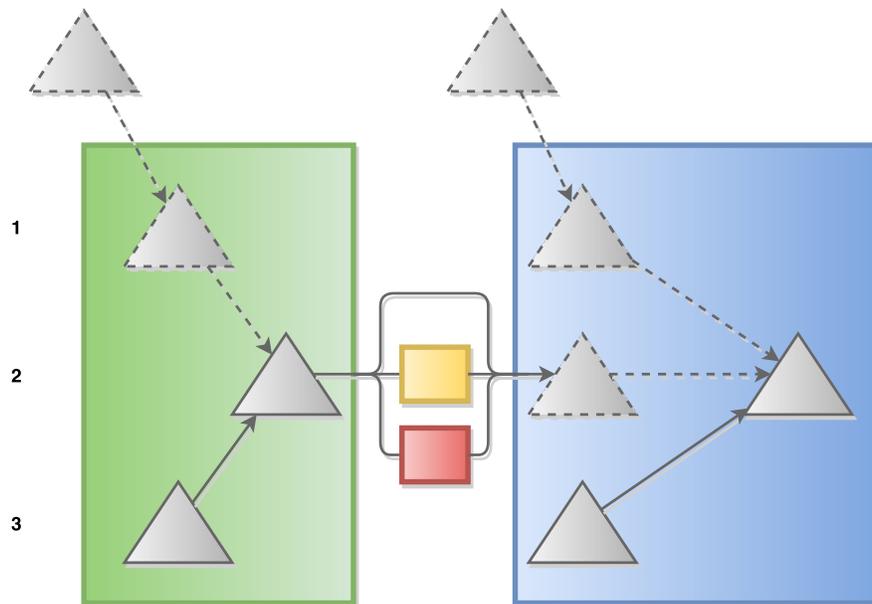


Fig. 2: Coarse classification for technical debt accumulation in projects with dependencies to external services

Figure 2 captures the different technical debt accumulation classes for projects with dependencies to external services. Case **3** depicts the most common situation in which the project accumulates technical debt that realizes at a certain point in time. In case **1** factors external to the component and its development invoke technical debt, and it may realize and invoke management needs at a point in time. In case **2** technical debt has realized (its interest probability is one, or a decision to remove the debt has been made) and it affects the project. In this scenario, the debt will propagate onwards, directly or through intermediaries, and accumulate in dependants. Accumulation channels are addressed in Figure 3.

The classification in Figure 2 is important for distinguishing technical debt inclined software change, as we must be able to distinguish between invoked change (case **2**) and internally accumulated debt (case **1** and **3**). This is because the monolith projects (see Figure 1) are able to internally accumulate technical debt, and we must form the baseline whilst aware of this.

In addition to source code, open-source projects provide access to documentation and other descriptors. Of these, the social media enabled ones form a set of projects that share a joint technical debt inducer: the social media APIs. These APIs provide business critical functionality for the projects, and every time they change, it causes several changes for their clients. Due to the massive adoption of social media services, their APIs (e.g., the Facebook Graph API ² and the Google OpenID API ³) integrate into and affect a vast amount of projects. This diverse collection of technologies, which all connect to the APIs that now cause changes for them, unveils a unique opportunity for technical debt research. As the changes propagate through various different technologies, they demonstrate a variety of technical debt propagation paths. Whilst our survey on to the social media involved open-source does not capture the full propagation space, particularly, propagation to business processes, it does yield a formidable library for the propagation of technical debt in delivered software and its supporting structures. Considering that usually this corresponds to the projects' delivered value, research should have a special interest to it.

Figure 3 demonstrates two channels, from a plethora of foreseeable options, through which technical debt can propagate and accumulate in new components. The upper channel captures a more problematic propagation method, in which no explicit dependency exists. In this, accumulated technical debt in the form of incomplete documentation causes a misunderstanding in a conceptualization phase of software development and leads to a complex component design. The lower channel demonstrates an explicit channel, where an interface change is felt in the dependent project as component disconnection. For example, a referred class is renamed in the service due to which the client can not access it in the original fashion. This leads to an erroneous implementation state in the dependent and undoubtedly invokes reparation efforts. In our MSR of open-source projects, over going both the human-produced messages and the automatically

² See <http://graph.facebook.com/>

³ See <https://profiles.google.com/>

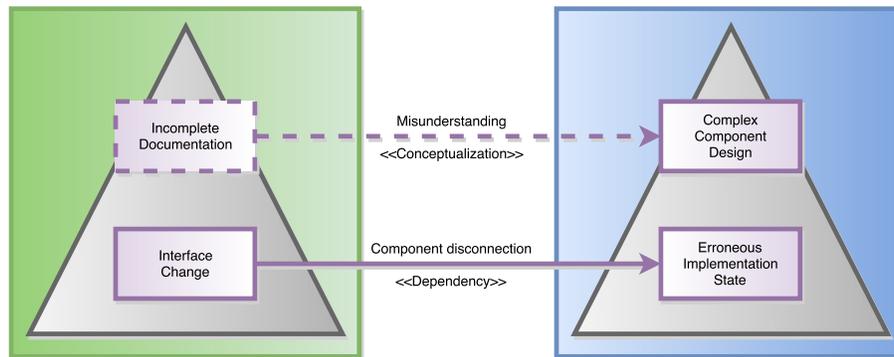


Fig. 3: Two examples of technical debt propagation channels

identified changes should reveal instances that fit both channels shown in Figure 3, but due to its implicit nature, identification of cases in the upper channel will be difficult.

4.1 Study Approach

We use the GitHub API through *PyGithub/PyGithub* library⁴. Our crawler is a Python program⁵ designed to crawl through all commits of a given project and report, for each commit, the date it was committed, the amount of changes (as the amount of added and removed rows), and the changed files. As such, our crawler is in itself an end part of a COP.

For an initial test of concept we chose Google’s closing of OpenID 2.0 service on April 20th 2015 [4] as a source of technical debt. We made a manual search in GitHub and discovered two Java projects which had closed issues mentioning Google closing the service. One was the Passport-based User Authentication system for *sails.js* applications—GitHub repository *tjwebb/sails-auth*. The other was a Grails website that provides information about festivals—GitHub repository *domurtag/festivals*. For a control project we selected another Java project that was similarly a user authentication system for *sails.js* as *sails-auth*, but did not appear to be involved with Google services—GitHub repository *waterlock/waterlock*.

4.2 Initial Results

Our analysis produced the graphs shown in Figure 4. The blue colour is used for *sails-auth*, red for *festivals* and cyan for *waterlock*. The X-axis marks the time. The dots denote the amount of changes in a commit. The bars denote commits

⁴ see <https://github.com/PyGithub/PyGithub> and in similar fashion for the other mentioned repositories as well

⁵ GitHub repository *tomibgt/GitHubResearchDataMiner*

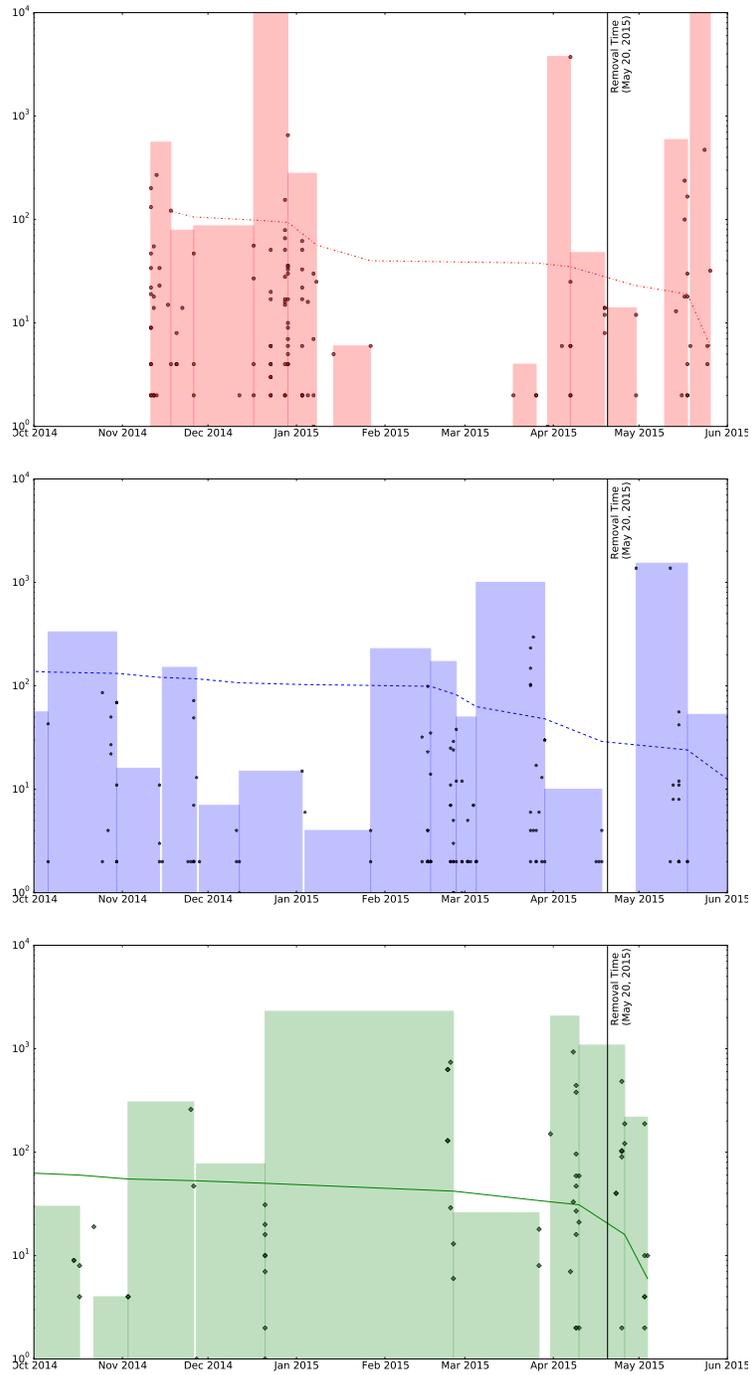


Fig. 4: Commit amount analysis for the three selected GitHub repositories

for a time period at least a week long. The lines denote commit frequency for previous time interval of at least a week. Finally, on the graph is marked the date-of-interest, April 20th 2015.

The lines show a general decline, which would appear to indicate that as a project progresses, less and less changes are made for it. Note that the Y-axis is logarithmic, which makes the lines curve down, instead of appearing linear.

It would appear to be supporting our hypothesis, where, after the marked date, *sails-auth* and *festivals* show decrease in the decline, unlike the control project *waterlock*. With only three projects and without more precise investigation we can not, of course, claim this to be strong evidence, but it is enough to encourage us in continuing with this approach.

Table 1: Commits for the *festival* repository file `show.gsp` around *Removal Time*

Time	Add	Remove	Delta
5/18/2015	0	2	-2
5/18/2015	7	12	-5
4/19/2015	3	1	2
4/19/2015	14	0	14
12/29/2014	11	6	5
12/29/2014	2	2	0
12/29/2014	2	1	1
12/28/2014	7	3	4
12/28/2014	8	14	-6

Table 2: Technique-wise recorded changes around *Removal Time*

Type	Add	Remove	Delta
js	86	2	84
gsp	35	3	32
jpg	.	.	.

With moderate work, the analyser can be modified to point out the files where there has been increasing changes in the commits correlating to the investigated events. (See Tables 1 and 2.) Looking into the changes made into these files should help us to analyse further the effort put by the programmers to pay the specific technical debt. Also, it should be possible to follow the wave of changes throughout the COP and analyse the propagation of the debt and the involved work and communication.

5 Applicability and Limitations

The aforescribed approach is limited by certain factors which we would like to address here. Firstly, we described this method as a possibility to explore the complete software context space, but the study design suggests using service calls to, especially social media, APIs and libraries as the method. It can be assumed, as previously discussed, that this approach does not capture all possible varieties of *software change* (see 2.2). This is a foreseeable data limitation even though it can be argued that the volume of captured *changes* would produce a representative set for analysis; accumulating enough assurance to allow abstraction to non-captured context areas.

Second, there are limitations potentially affecting the identification of technical debt instances. We discussed the technical debt properties which can be used to associate a *software change* with managing technical debt. While this set of properties currently accounts the state-of-the-art from technical debt research, if not exhaustive, the properties may lead to missing particular sub-classes of technical debt. Approach discussed in the following paragraph, can be considered a partial remedy to this.

Finally, foreseeable limitations may also affect the tracking of technical debt instances. As a premise for tracking, [6] showed the instances' ability to span over multiple components. Modelling of the *chain of projects* was introduced as the method to allow capturing this behaviour. The current classification presented in Figure 1 considers one dimension for the COPs—presumed to be the most dominant. This classification can be a limiting factor, especially in large hybrid COP projects, but we argue that this can be countered by iteratively exploring more dimensions for the COPs until all technical debt inclined changes have been successfully associated to the technical debt instances.

Overcoming the limitations and achieving the study's objectives, there is a number of applications for the results (discussed in Section 3.1). Firstly, demonstrating technical debt's ability to propagate, almost boundlessly, between software projects and artifacts should fuel the apparent paradigm shift in software life-cycle management where the inter-connectivity of software project entities carries increased value. Second, documenting the ways in which technical debt can propagate should provide an interface for integrating knowledge from other research domains to enhance technical debt management by for example applying financial models for technical debt strategization. Lastly, associating the documentation's technical debt propagation channels with information regarding their value accumulation allows automated tooling approaches to be introduced, but also makes technical debt an integral and explicit component of the software project's value production and its assessment.

6 Conclusions and Future Work

With similar studies in the future, using different event markers, it is possible to map the propagation of technical debt by observing the amount of increased work caused by different causes of technical debt. It is possible to observe who pays the technical debt and how it is propagated from the original cause (e.g., a change in a fundamental library used by many projects) through facade libraries and components to the final applications.

In an effort to efficiently analyse the propagation of technical debt through propagation channels, a taxonomy of projects in GitHub should be created to help characterize and predict the characteristics of the projects. To this end, and to achieve the goals stated herein, we have analyzed over twenty-eight thousand projects from GitHub and have successfully identified a number of projects with references to suitable external services. According to Lambe [11], even taxonomies founded on criteria that do not stand all scrutiny, can allow for reliable

predictions and descriptions of characteristics of new members of the taxonomy based on very little information. A well created taxonomy combined with our expected mining results should help us identify different propagation channels within the projects without even analysing them at the code level. Should we find two or more clusters of different kinds of change behaviour within a single taxonomy class, it could suggest that the propagation channels between these clusters differ from each other.

There can, of course, be other causes to variance within a class. For example, it would be beneficial to have the information of the process maturity level for each project team. This kind of information would be significant in understanding the project's sensitivity to external changes and the general preparedness and carefulness in the design. [17]

Such work would provide us with a better understanding of the economy of technical debt, which again would help us give good estimates on the actual costs of applying, for example, social media APIs in an application system and compare it with the projected benefits and income. It would help in answering the question: would applying certain features increase the revenue from the service.

Acknowledgment

J. Holvitie is supported by the Nokia Foundation Scholarship and the Finnish Foundation for Technology Promotion, the Ulla Tuominen Foundation, and the Finnish Science Foundation for Economics and Technology grants.

References

1. Brown, N., Cai, Y., Guo, Y., Kazman, R., Kim, M., Kruchten, P., Lim, E., MacCormack, A., Nord, R., Ozkaya, I., et al.: Managing technical debt in software-reliant systems. In: Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research. pp. 47–52. ACM (2010)
2. Cunningham, W.: The WyCash portfolio management system. In: Proceedings Addendum for Object-Oriented Programming Systems, Languages, and Applications (OOPSLA). pp. 29–30. No. 22 (1992)
3. Dabbish, L., Stuart, C., Tsay, J., Herbsleb, J.: Social coding in github: transparency and collaboration in an open software repository. In: Proceedings of the ACM 2012 Conference on Computer Supported Cooperative Work. pp. 1277–1286. ACM (2012)
4. Google Developers: Migrating to google sign-in (2015), <https://developers.google.com/identity/sign-in/auth-migration>
5. Guo, Y., Seaman, C., Gomes, R., Cavalcanti, A., Tonin, G., Da Silva, F., Santos, A., Siebra, C.: Tracking technical debt - an exploratory case study. In: 27th IEEE International Conference on Software Maintenance (ICSM). pp. 528–531. IEEE (2011)
6. Holvitie, J., Leppänen, V., Hyrynsalmi, S.: Technical debt and the effect of agile software development practices on it-an industry practitioner survey. In: Sixth International Workshop on Managing Technical Debt (MTD). pp. 35–42. IEEE (2014)

7. Izurieta, C., Vetrò, A., Zazworka, N., Cai, Y., Seaman, C., Shull, F.: Organizing the technical debt landscape. In: Third International Workshop on Managing Technical Debt (MTD). pp. 23–26. IEEE (2012)
8. Kagdi, H., Collard, M.L., Maletic, J.I.: A survey and taxonomy of approaches for mining software repositories in the context of software evolution. *Journal of Software Maintenance and Evolution: Research and Practice* 19(2), 77–131 (2007)
9. Kalliamvakou, E., Gousios, G., Blincoe, K., Singer, L., German, D.M., Damian, D.: The promises and perils of mining github. In: Proceedings of the 11th Working Conference on Mining Software Repositories. pp. 92–101. ACM (2014)
10. Kruchten, P., Nord, R.L., Ozkaya, I.: Technical debt: From metaphor to theory and practice. *IEEE Software* 29(6) (2012)
11. Lambe, P.: Organising knowledge: taxonomies, knowledge and organisational effectiveness. Chandos Publishing (2007)
12. Li, Z., Avgeriou, P., Liang, P.: A systematic mapping study on technical debt and its management. *Journal of Systems and Software* 101, 193–220 (2015)
13. McConnell, S.: Technical debt. 10x Software Development Blog, (Nov 2007). Construx Conversations. URL= <http://blogs.construx.com/blogs/stevemcc/archive/2007/11/01/technical-debt-2.aspx> (2007)
14. McGregor, J., Monteith, J., Zhang, J.: Technical debt aggregation in ecosystems. In: Third International Workshop on Managing Technical Debt (MTD). pp. 27–30. IEEE (2012)
15. Schmid, K.: A formal approach to technical debt decision making. In: Proceedings of the 9th International ACM SIGSOFT Conference on Quality of Software Architectures. pp. 153–162. ACM (2013)
16. Seaman, C., Guo, Y., Izurieta, C., Cai, Y., Zazworka, N., Shull, F., Vetrò, A.: Using technical debt data in decision making: Potential decision approaches. In: Third International Workshop on Managing Technical Debt (MTD). pp. 45–48. IEEE (2012)
17. Suenson, E.: How Computer Programmers Work – Understanding Software Development in Practise. Ph.D. thesis, Turku Centre for Computer Science (2015)
18. Tsay, J., Dabbish, L., Herbsleb, J.: Influence of social and technical factors for evaluating contribution in github. In: Proceedings of the 36th International Conference on Software Engineering. pp. 356–366. ACM (2014)

Pattern recognition with Spiking Neural Networks: a simple training method

François Christophe, Tommi Mikkonen, Vafa Andalibi, Kai Koskimies, and Teemu Laukkarinen

Tampere University of Technology
Korkeakoulunkatu 1, FI-33720 Tampere, Finland
`firstname.lastname@tut.fi`

Abstract. As computers are getting more pervasive, software becomes transferable to different types of hardware and, at the extreme, being bio-compatible. Recent efforts in Artificial Intelligence propose that software can be trained and taught instead of “hard-coded” sequences. This paper addresses the learnability of software in the context of platforms integrating biological components. A method for training Spiking Neural Networks (SNNs) for pattern recognition is proposed, based on spike timing dependent plasticity (STDP) of connections. STDP corresponds to the way connections between neurons change according to the spiking activity in the network, and we use STDP to stimulate outputs of the network shortly after feeding it with a pattern as input, thus creating specific pathways in the network. The computational model used to test this method through simulations is developed to fit the behaviour of biological neural networks, showing the potential for training neural cells into biological processors.

Keywords: Pattern recognition, Artificial Neural Networks, Spiking Neural Networks, Computational models, Computational Biology

1 Introduction

Software is everywhere: the human environment is populated by more and more software-driven intelligent devices, connected by Internet and other networks. With the apparition of wearables and implantables, computers are getting more and more pervasive and close to the biological world. In such systems, software is expected to expand on various types of platforms.

In the current information technology, the interplay between biology and software has been indirect. Humans use software through various user interfaces, rather than with direct communication links. Concepts from biological systems have inspired various heuristic algorithms to solve computer science problems (typically optimization and search), or inspired software for communication systems to mimic the adaptive behavior of biological systems [15, 12]. Novel ways of programming by training, teaching, imitation and reward are already being

demonstrated in robotics with the help of in-silico chips behaving like neurons, i.e. neuromorphic chips [7].

The work reported in this paper is a first step in a project aiming at developing techniques to support the direct run-time interaction of biological entities and software. Our vision is that eventually software interacts directly with the biological world: software controls biological entities, and biological entities control software systems. Thus, rather than using the biological world as a model of new algorithms, we intend to let biological entities communicate directly with software. In this way, software systems and biological entities form co-operational organizations in which both parties solve problems suitable for them, contributing to a common goal. Typically, biological entities are superior in efficient massive parallel processing of fuzzy data and resilient to damage, while traditional software systems are better suited for making discrete, well-defined logical decisions. Biological entities are also far more energy-efficient than traditional computing devices.

The project focuses on integrating real neural cultures with software systems. A central problem then is the training of biological neural structures for a particular task and the connection of the trained neural culture to the software system. However, to experiment with different approaches to solve these problems, available biological neural cultures impose many practical problems: their detailed structure is difficult to study, their lifetime is limited, and they need constant nutrition. Luckily, for the past decades, Artificial Neural Networks (ANNs) have evolved to the point of being currently very close in behaviour to biological neural structures [13]. Thus, in the first stage of the project, we use ANNs to simulate biological neural networks. In a later stage we aim to transfer the techniques to biological neural cultures currently available on Multi-Electrode Arrays (MEAs) [16].

In this paper we study the basic training problem of biological neural networks using a biologically realistic model of spiking neurons. A simple pattern recognition problem is applied to this model. We demonstrate that a training technique based on Spike-Timing-Dependent-Plasticity (STDP) appears to be sufficient for these kinds of tasks.

The rest of this paper is structured as follows. In Section 2, we discuss related work. In Section 3, we introduce the computational model used in this paper. In Section 4, we evaluate the results we have obtained. In Section 5, we draw some final conclusions.

2 Related work

In [11], Maass draws a retrospective of the techniques used for modeling neural networks and presents the third generation of neural networks: SNNs. This study

classifies neural networks according to their computational units into three generations: the first generation being perceptrons based on McCulloch-Pitts neurons [4], the second generation being networks such as feedforward networks where neurons apply an “activation function”, and the third generation being networks where neurons use spikes to encode information. From this retrospective, Maass presents the computational advantages of SNNs according to the computation of, first, boolean functions, and secondly according to functions with analog input and boolean output. For instance, Seung demonstrated in [14] that SNNs can be trained to behave as an XOR logical gate. The study of single neurons, the population of networks and plasticity [3] provides guidelines on how single computation units, i.e. neurons, function but more importantly on how to structure a network (e.g. number of layers, number of units in a layer) and on the models of evolution of connectivity between neurons, i.e. plasticity.

In their chapter on computing with SNNs [13], Paugam and Bohte present different methods applied for learning in SNNs. In this review chapter, they distinguish the traditional learning methods issued from previous research with ANNs, and learning methods that are emerging solely from computing with SNNs. Among the traditional methods are temporal coding, unsupervised learning such as Hebbian learning or Kohonen’s self-organizing maps, and supervised learning such as error-backpropagation rules. About the “unconventional” learning methods, they are regrouped into so-called Reservoir Computing methods. Reservoir Computing methods regroup Echo State Networks and Liquid State Machines. The main characteristic of reservoir computing models relies in the apparent disorganization of the network between input and output layers. This network, the reservoir, is a recurrent network where neurons are interconnected by a random sparse set of weighted links. More over, this network is usually left untrained and only the output connections are trained and optimized according to the desired answer based on what input is given.

To a certain extent, the simple training method proposed in this paper follows the idea of training only the output layer as STDP will act alone for reinforcing the positive pathways of the network. However, the network studied in this paper is simply a feed-forward network modeled with the BRIAN simulator [2].

3 Computational model

The training method presented in this paper is studied with the computational model presented in this section because it provides a first test of feasibility before testing this method on biological Neural Networks (bioNNs). The model used for this research is composed of two basic elements: neurons and synapses. Neurons are built based on the firing model of Izhikevich which is shown to be very realistic in [9]. Synapses follow the Spike Timing Dependent Plasticity (STDP), meaning that a connection between two neurons will grow if the post-synaptic neuron fires soon after the pre-synaptic neuron. On the opposite, a connection

will decrease if the post-synaptic neuron fires before the pre-synaptic neuron. This section presents these two basic models in more details and then gives a view on the composition of the entire neural network.

3.1 Model of spiking neuron

The model of a spiking neuron used in this study is from Izhikevich [8]. This model reproduces the dynamic behavior of neurons while being computationally simple as opposed to models accounting for the structural parameters of neurons (for example, Hodgkin-Huxley model [6]). The Izhikevich model expresses the variations of electric potential in the neuron's membrane according to the current flowing through the membranes ion channels. These electric potential variations are expressed in the form of two differential equations, as follows:

$$\begin{aligned} C \frac{dv}{dt} &= k(v - v_r)(v - v_t) - u + I \\ \frac{du}{dt} &= a(bv - u) \end{aligned} \quad (1)$$

where v represents the membrane potential, u the recovery current of the membrane, I the input current through the membrane, C the membrane capacitance, v_r the resting potential of the membrane, v_t the threshold potential for the membrane to fire a spike, k , a and b parameters adjusted according to the firing pattern required. The variables v and u of equation 1 are reset after v reaches a peak value v_{peak} , as follows:

$$\text{if } v \geq v_{peak}, \text{ then } \begin{cases} v \leftarrow c \\ u \leftarrow u + d \end{cases} \quad (2)$$

Figure 1 presents an example of simulation of a pyramidal neuron exhibiting regular spiking as stimulated with a constant input current of 70pA. The same simulation among others are compared with recordings from real neurons in [9] showing the precision of this models in reproducing these dynamic patterns while being computationally simple.

3.2 Model of STDP

STDP is a rule for neurons to strengthen or weaken their connections according to their degree of synchronous firing [1]. This rule mostly known in Neurobiology and Neuroscience is similar to the Hebbian learning rule widely used in learning Artificial Neural Networks and Self-Optimizing Maps [5, 10]. Considering a pre-synaptic neuron i and a post-synaptic neuron j , the STDP rule characterizes the changes in synaptic strength as:

$$\Delta w_j = \sum_{k=1}^N \sum_{l=1}^N W(t_j^l - t_i^k) \quad (3)$$

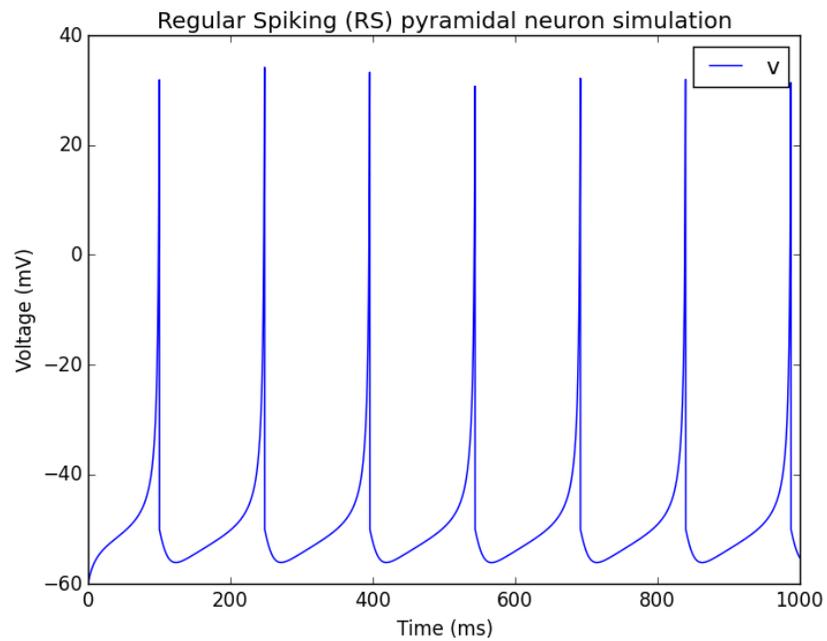


Fig. 1. Example of Regular Spiking pyramidal neuron simulated with Izhikevich model from Equations 1 and 2 (step input stimulation current $I = 70\text{pA}$ from 0 to 1s). Simulated with the following values of parameters: $v_r = -60\text{mV}$, $v_t = -40\text{mV}$, $v_{peak} = 35\text{mV}$, $C = 100\text{pF}$, $k = 0.7\text{pA/mV}^2$, $a = 30\text{Hz}$, $b = -2\text{nS}$, $c = -50\text{mV}$, $d = 100\text{pA}$.

with the function $W(x)$ defining the order of decrease or increase of strength depending on the synchrony of spiking between pre- and post-synaptic neurons, expressed as:

$$W(x) = \begin{cases} A_+ \exp(-\frac{x}{\tau_+}) & \text{if } x > 0 \\ A_- \exp(\frac{x}{\tau_-}) & \text{otherwise.} \end{cases} \quad (4)$$

In equations 3 and 4, t_j^l represents the l^{th} spiking time of neuron j ; similarly, t_i^k stands for the k^{th} spike timing of neuron i ; A_+ and A_- are constants defining the amplitude of change in weight (at $t = 0_+$ and $t = 0_-$, respectively); and, τ_+ and τ_- are time constants of the exponential decrease in weight change.

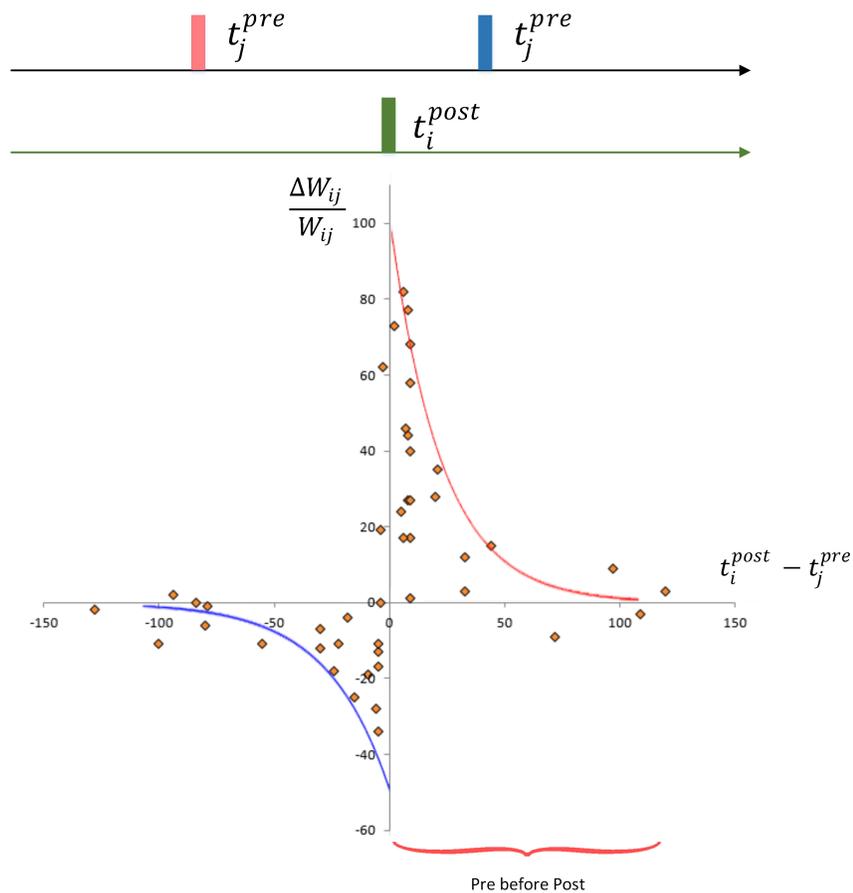


Fig. 2. Model of STDP (reproduced after Bi and Poo [1])

Figure 2 represents equation 4 of relative weight changes in relation with the experiments realized by Bi and Poo in [1]. This figure shows typically a decrease in synaptic weight when the pre-synaptic neuron spikes after the post-synaptic neuron, and on the opposite an increase in connectivity from pre- to post-synaptic neuron if the pre-synaptic neuron spikes just before the post-synaptic neuron.

3.3 Network model

The network developed in this study as an example of pattern recognition is presented in Fig. 3. This network is dedicated at recognizing patterns from a 5x5 pixels image. It is composed of:

- 25 input neurons corresponding to each pixel of the image,
- a hidden layer of 5 neurons, and
- 2 output neurons (1 corresponding to the neuron reacting when a circle appears in the image, the other one being a test neuron for comparison between learning and no stimulation).

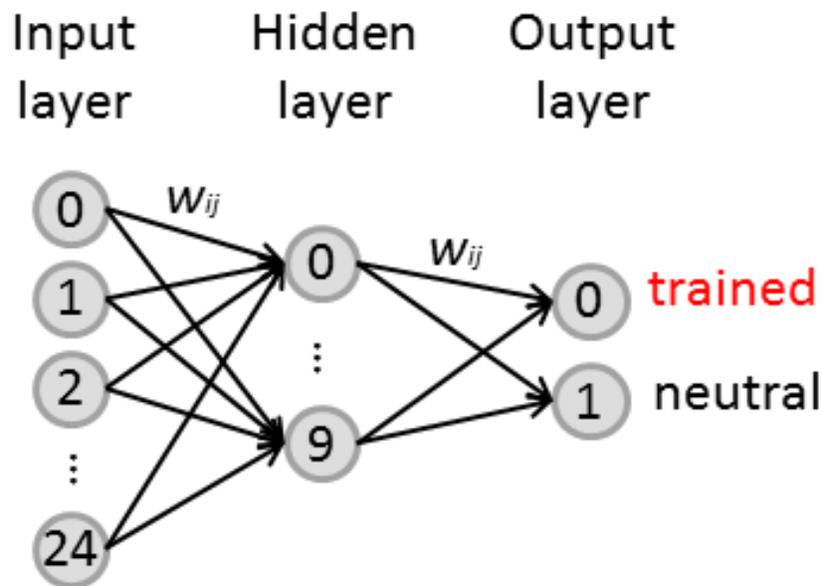


Fig. 3. Representation of the network developed for 1 pattern recognition

4 Training method and Evaluation

This section goes more into the details of the simple learning method using STDP rule for training the network. A pattern recognition task is used as a case study for testing the learning method proposed.

4.1 Pattern recognition task

The pattern recognition task evaluated during this simple case study consists in making the difference between a circle given as input stimuli and other inputs (in that case an X-cross shape). These shapes are represented as coming from a 25 pixels image (5x5 matrix), each pixel being binary: black or white. Figure 4 presents the input circle and X-cross patterns and their respective representations as input stimuli to the network.

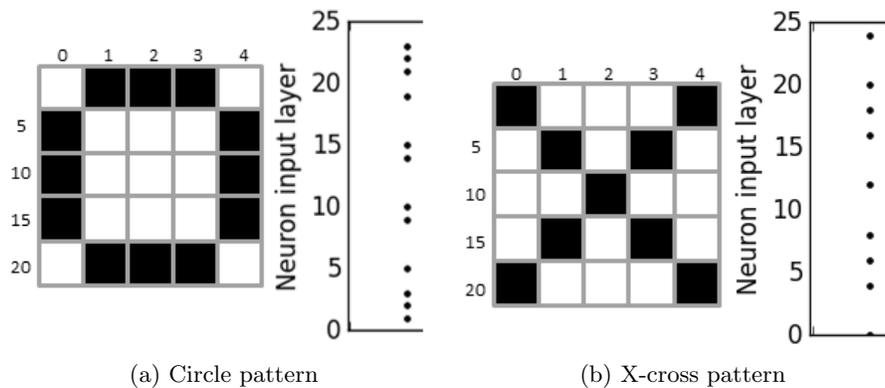


Fig. 4. Input patterns as images and their respective representations as input stimuli

4.2 Training method and test protocol

After initial tests on the learning ability of the network, the training period was adjusted to be 15s during which the input layer is stimulated every 100ms with a circle pattern. 10ms after stimulating the input layer, the output neuron is given an external stimulation making it to spike. This spiking, in relation with the preliminary spiking of neurons from the input layer, reinforces the paths between activated neurons of the input layer and the trained neuron of the output layer. This training can be seen in the first phase of the time diagrams (top and middle) of Figure 5 from $t = 0$ to 15s.

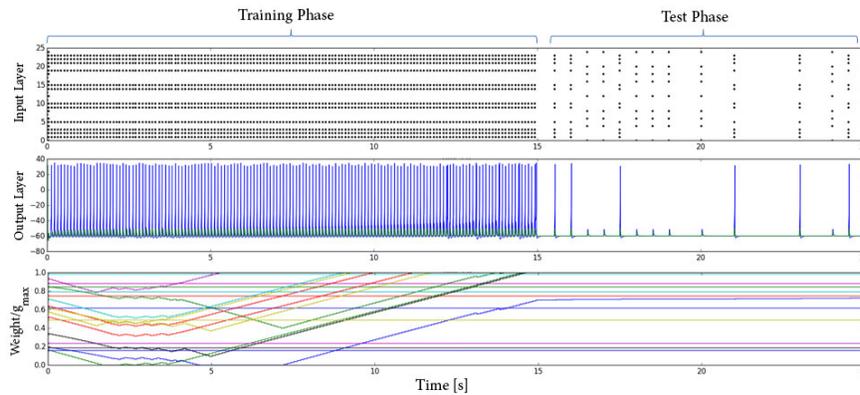


Fig. 5. Training phase and test phase of the circle recognition experiment

The testing phase is composed of 6 stimuli with circle pattern and 7 stimuli with a different pattern (in that case representing an X-cross). These stimuli of the input layer happen without any external stimulation of the output layer. The neuron trained for recognizing a circle fires on its own after the learning phase. These test patterns are sent between $t = 15,5$ to 25s with the following sequence: {circle, circle, cross, cross, circle, cross, cross, cross, cross, circle, circle, cross, circle} at the respective times {15.5, 16, 16.5, 17, 17.5, 18, 18.5, 19, 20, 21, 23, 24, 24.5} seconds. The two upper time diagrams of Figure 5 show this test phase.

The third time diagram of Figure 5 (down) presents the evolution of strength of synapses between neurons from the hidden layer and neurons from the output layer. This evolution shows first a stabilization period from the random strengths given as initial condition to lower values. Secondly, learning can be seen as the strengths of certain synapses increase to high levels of connectivity (i.e. to levels higher than 0.8 times the maximum connectivity and often reaching this maximum).

4.3 Results

The 1000 simulations of this experiment revealed a success rate of 80.3% in recognizing a circle from a cross. This rate was computed after the execution of a thousand experiments. From these experiments, five different cases were observed:

- correct learning: output fires only when a circle is given as input (80.3%)
- some mistakes: output fires sometimes when input is a cross (5.7%)
- always firing: output always fires whatever the input may be (12.8%)
- no learning: output never fires (1.1%)

- wrong learning: output fires only when input is a cross (0.1%)

These different cases are represented in Figure 6.

These results show a high learning rate, i.e. a high rate of correct experiments (80.3%), meaning that such learning method has correct grounds. Indeed, there is space for improving this rate and a lot to learn from the analysis of failed experiments.

First, we can notice that the network keeps on learning even after the training phase has stopped. Each time a pattern is fed as input to the network, small increase in synaptic weights take place.

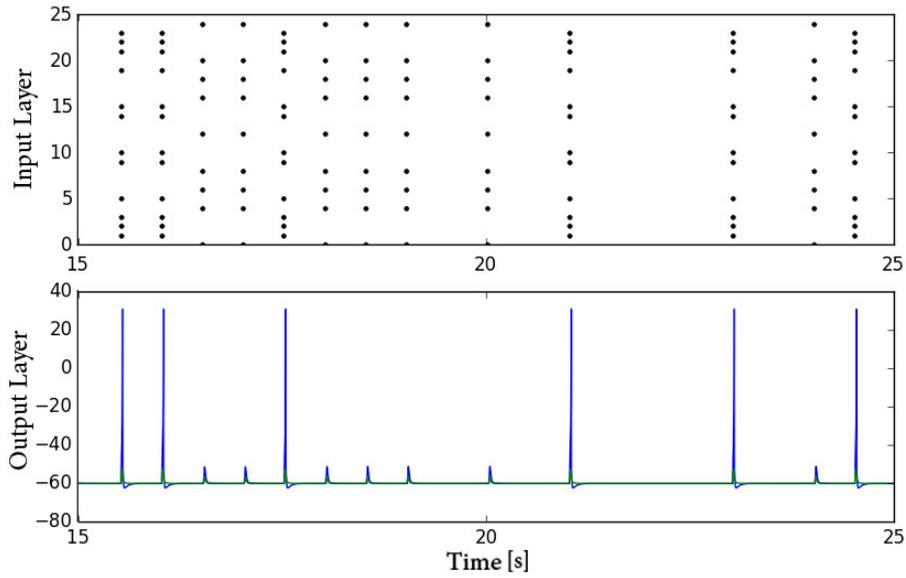
The second important thing noticed due to this continuous learning is that the output neuron trained to recognize a circle also gets trained when receiving another pattern as input. As the synaptic levels may already be high, it requires only few X-cross stimulation signals for the output neuron to start spiking and we notice that when it has learned to spike for a cross pattern, it will then fire each time a cross appears. This is what happens for the cases where some mistakes are found (57 cases out of 1000 simulations).

Third, for 128 simulations the synaptic levels are high from the beginning of the training due to the initial random value of synaptic weights. This causes the network to be “over-trained” and thus to fire for every kind of patterns from the beginning of the test phase. In the contrary, for 11 simulations the output neuron does not fire at all when given any input. These 11 tests show clearly low rates in synaptic weights and low increase in synaptic weights during training expressing the fact that the network does not have time to learn during this period of time. The only simulation where the output neuron fires only and always when given the wrong input could not be explained.

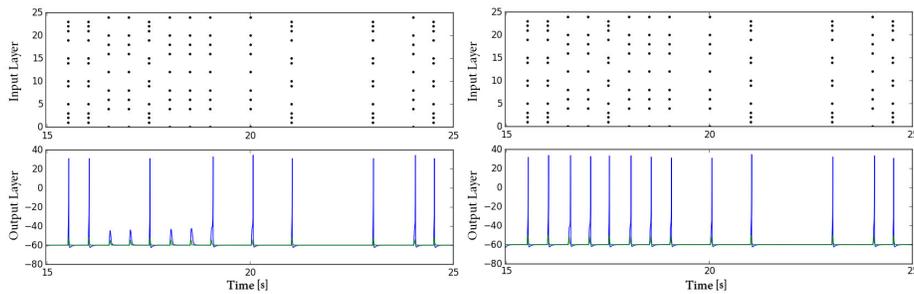
5 Discussion and future work

We have studied a central problem in using biological neural networks as computing resources: how to train the neural culture for a particular task. The SNN presented in this paper shows reasonable success rate in learning (80.3%) to differentiate between two patterns. As the behavior of the simulated network is very close to that of real biological neural networks (bioNNs), this experiment gives preliminary insight for using bioNNs to process complex tasks requiring massive parallelism.

However, there is still possibility for improvement in performing such recognition tasks. In the case where pathways did not have time to form during initial training, it is indeed possible to continue training the network until synaptic level reaches the appropriate levels. On the opposite case, when the network fires for any type of input pattern, meaning that it is “over-trained”, training the network with negative output stimulation should help restoring the differentiation

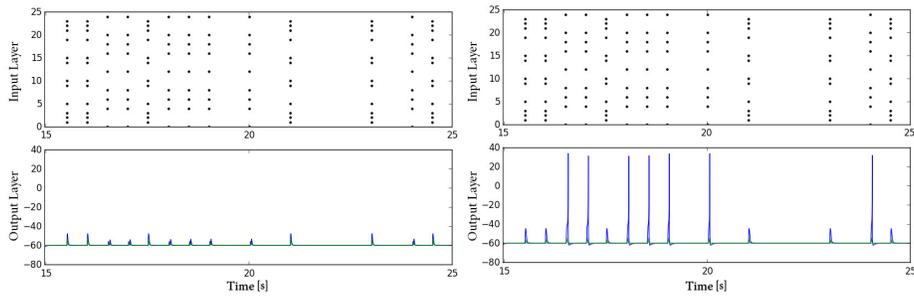


(a) Correct learning (rate 80.3%)



(b) Some mistakes (rate 5.7%)

(c) Always firing (rate 12.8%)



(d) No learning (rate 1.1%)

(e) Wrong learning (rate 0.1%)

Fig. 6. Example of the 5 different cases found in experiments

between input patterns. Such negative stimulation can be realized by stimulating the output neuron just prior to the input pattern, when an input pattern is to be discarded by the network. This way, the synaptic pathways related to this input pattern would decrease due to the STDP rule, thus de-correlating the output neuron with this input pattern.

The proximity in behavior of SNNs from bioNNs should not require efforts to transfer computations from silicon platforms into biological platforms. From this point, transferring the various tasks developed for the past sixty years with ANNs (e.g. classifiers, generators, etc.) to small bio-computers will be possible. Research directions for such transfer lead to the following questions:

- Can a taxonomy of the various tasks performed with ANNs and their hierarchical relations be developed?
- Can we classify tasks as unitary/atomic to some higher level? On the other hand, can tasks be broken down into summation of unitary tasks?
- Is it possible to automatically associate tasks to network structures (number of nodes, number of layers etc.) ?
- Can training also be automatically generated for these specific tasks?

These questions could lead to the construction of a compiler deriving the number of biological neural networks used for an application, their structural parameters and the training associated to each task required for the application.

6 Conclusion

In this paper, we presented a simple learning method using STDP for training pathways of a SNN. This method was tested on a SNN model trained to differentiate between two patterns given as input to the network. The results of this test (80.3% success rate) combined with the fact that the behaviour of the simulated network is very close to the one of a real biological network gives promising expectations for the future of this project. This first test is still a preliminary test towards applying bioNNs to the computation of more complex tasks such as handwritten digit and character recognition ¹. Expectations on the results to such test should give similar success rate as to the test conducted in this paper. Next experiments will be conducted on real biological cells in order to validate the possibility of training bioNNs for pattern recognition tasks. In the final stage we intend to connect such trained bioNNs with software applications requiring pattern recognition capability such as classification of moving objects.

Acknowledgement

This research is funded by the Academy of Finland under project named “Bio-integrated Software Development for Adaptive Sensor Networks”, project number 278882.

¹ Such training is usually tested on the MNIST dataset available from <http://yann.lecun.com/exdb/mnist/>

References

1. Bi, G.Q., Poo, M.M.: Synaptic modifications in cultured hippocampal neurons: dependence on spike timing, synaptic strength, and postsynaptic cell type. *The Journal of neuroscience : the official journal of the Society for Neuroscience* 18, 10464–10472 (1998)
2. Brette, R., Rudolph, M., Carnevale, T., Hines, M., Beeman, D., Bower, J.M., Diesmann, M., Morrison, A., Goodman, P.H., Harris, F.C., Zirpe, M., Natschläger, T., Pecevski, D., Ermentrout, B., Djurfeldt, M., Lansner, A., Rochel, O., Vieville, T., Muller, E., Davison, A.P., El Boustani, S., Destexhe, A.: Simulation of networks of spiking neurons: A review of tools and strategies (2007)
3. Gerstner, W., Kistler, W.: Spiking neuron models: Single neurons, populations, plasticity. Cambridge University Press (2002)
4. Hayman, S.: The mcculloch-pitts model. *IJCNN'99. International Joint Conference on Neural Networks. Proceedings (Cat. No.99CH36339)* 6 (1999)
5. Hebb, D.O.: *The Organization of Behaviour: A neuropsychological theory*. Wiley (1949)
6. Hodgkin, A.L., Huxley, A.F.: A quantitative description of membrane current and its application to conduction and excitation in nerve. *Journal of Physiology* pp. 500–544 (1952)
7. Indiveri, G., Linares-Barranco, B., Hamilton, T.J., van Schaik, A., Etienne-Cummings, R., Delbruck, T., Liu, S.C., Dudek, P., Häfliger, P., Renaud, S., Schemmel, J., Cauwenberghs, G., Arthur, J., Hynna, K., Folorosele, F., Saighi, S., Serrano-Gotarredona, T., Wijekoon, J., Wang, Y., Boahen, K.: Neuromorphic silicon neuron circuits. *Frontiers in neuroscience* 5(May), 73 (1 2011)
8. Izhikevich, E.M.: Simple model of spiking neurons. *IEEE transactions on neural networks / a publication of the IEEE Neural Networks Council* 14(6), 1569–72 (1 2003)
9. Izhikevich, E.M.: *Dynamical Systems in Neuroscience : The Geometry of Excitability and Bursting*, Chapter 8. The MIT Press (2007)
10. Kohonen, T.: The self-organizing map 21(May), 1–6 (1998)
11. Maass, W.: Networks of spiking neurons: The third generation of neural network models. *Neural Networks* 10(9), 1659–1671 (1997)
12. Nakano, T., Suda, T.: Self-organizing network services 16(5), 1269–1278 (2005)
13. Paugam-Moisy, H., Bohte, S.: Computing with spiking neuron networks. In: *Handbook of Natural Computing*, pp. 335–376 (2012)
14. Seung, H.S.: Learning in spiking neural networks by reinforcement of stochastic synaptic transmission. *Neuron* 40(6), 1063–1073 (2003)
15. Suzuki, J., Suda, T.: A middleware platform for a biologically inspired network architecture supporting autonomous 23(2), 249–260 (2005)
16. Taketani, M., Baudry, M. (eds.): *Advances in Network Electrophysiology: Using Multi-electrode Arrays*. Springer (2006)

Author Index

A	
Ahmadighohandizi, Farshad	76
Andalibi, Vafa	296
Antinyan, Vard	1
B	
Berki, Eleni	221
Beszédes, Árpád	46
C	
Chen, Hong-Mei	134
Christophe, Francois	296
G	
Gergely, Tamás	46
Gyimóthy, Tibor	46
H	
Heikkinen, Esa	266
Helenius, Marko	221
Holvitie, Johannes	281
Horváth, Ferenc	46
Hylli, Otto	251
Hyrnsalmi, Sami	61, 236
Hämäläinen, Timo	106
Hämäläinen, Timo D.	266
J	
Jaaksi, Ari	134
K	
Kazman, Rick	134
Kilamo, Terhi	16, 119
Koskimies, Kai	296
L	
Laukkarinen, Teemu	296
Lehtonen, Samuel	31
Lehtonen, Timo	16
Leppänen, Ville	61, 206, 236, 281
Li, Linfeng	221
M	
Mattila, Anna-Liisa	251
Mikkonen, Tommi	16, 119, 134, 296
Mäkelä, Jari-Matti	206
N	
Nummenmaa, Timo	221
P	
Pekkarinen, Esko	106

Penjam, Jaan	149
Poranen, Timo	31, 164
R	
Rahikkala, Jurka	61
Rauti, Sampsa	206
Rindell, Kalle	236
Rintala, Matti	179
Ripatti, Maria	119
S	
Salli, Karri-Tuomas	119
Salo, Risto	164
Sandberg, Anna	1
Smed, Jouni	281
Staron, Mirosław	1
Suonsyrjä, Sampo	16, 134
Suovuo, Tomi 'Bgt'	281
Systä, Kari	76, 251
T	
Tengeri, Dávid	46
Terho, Henri	134
Teuho, Mikko	106
Tyugu, Enn	149
U	
Uitto, Joni	206
V	
Valmari, Antti	91, 179
Vancsics, Béla	46
Veltri, Niccolò	194
Vidács, László	46
Z	
Zhang, Zheyang	164