Extended Abstracts

of the

Eighth Seminar on Advanced Techniques & Tools for Software Evolution,

SATToSE 2015

Anya Helene Bagge (ed.)
Mons, Belgium, July 6–8, 2015.
CONTENTS

Keynotes 7

Anita Sarma:
MANAGING SOCIO-TECHNICAL DEPENDENCIES IN DISTRIBUTED SOFTWARE DEVELOPMENT 7

Martin Pinzger:
ANALYZING CHANGES IN SOFTWARE SYSTEMS: FROM CHANGEDISTILLER TO FMDIFF 7

Tutorials 8

Massimiliano Di Penta:
EMPirical SOFTWARE Engineering 8

Gregorio Robles:
THE METRICSGRIMOIRE TOOLSET – A MATURE, INDUSTRIAL-STRENGTH TOOLSET TO OBTAIN DATA FROM SOFTWARE REPOSITORIES 8

Session 1: Meta-Programming and Transformations 16

Laurent Christophe, Elisa Gonzalez Boix, Coen De Roover and Wolfgang De Meuter:
A META-LEVEL API FOR JAVASCRIPT INSTRUMENTATION PLATFORMS 16

Gustavo Santos, Nicolas Anquetil, Anne Etien, Stéphane Ducasse and Marco Tulio Valente:
MACRORECORDER: RECORDING AND REPLAYING SOURCE CODE TRANSFORMATIONS 20
Reinout Stevens and Coen De Roover:  
Reasoning over AST Changes 24

Vadim Zaytsev:  
Evolution of Metaprograms, or How to Transform XSLT to Rascal 27

Session 2: Mining, Queries and Analysis 31

Vincent Blondeau, Anne Etien, Nicolas Anquetil, Stéphane Ducasse, Sylvain Cresson and Pascal Croisy:  
Predicting the health of a project? An assessment in a major IT company 31

Verena Honsel, Daniel Honsel, Jens Grabowski and Stephan Waack:  
Developer Oriented and Quality Assurance Based Simulation of Software Processes 35

Csaba Nagy and Anthony Cleve:  
Parsing and Analyzing SQL Queries in Stack Overflow Questions 39

Carlos Noguera:  
Example-Driven Model Queries 43

Philip Makedonski and Jens Grabowski:  
Weighted Multi-Factor Multi-Layer Identification of Potential Causes for Events of Interest in Software Repositories 47

Session 3: Tools and Languages + Technology Showdown 52

Brice Govin, Nicolas Anquetil, Anne Etien, Stephane Ducasse and Arnaud Monegier Du Sorbier:  
Overview of Reverse Engineering Tools Features and Extension 52

Andrea Caracciolo:  
On the Evaluation of a DSL for Architectural Consistency Checking 55

Tim Molderez and Coen De Roover:  
A Search-based Approach for Generalizing and Refining Source Code Templates 58

Vadim Zaytsev:  
BibSLEIGH: Bibliography of Software Language Engineering in Generated Hypertext 59
Session 4: Human / Developer Factors 63

Carlos Ulderico Cirello Filho:
Pretty Printers: anatomy and measured effects on productivity of teams of developers 63

Truong Ho-Quang:
Interactive User-Oriented Views for Better Understanding Software Systems 67

Miguel Ángel Fernández, Gregorio Robles and Jesus M. Gonzalez-Barahona:
Collaboration Networks in Software Development: Perspectives from Applying different Granularity Levels using Social Network Analysis 73

Alexandre Bergel, Vanessa Peña and Tobias Kuhn:
Controlled Experiment to Assess a Test-Coverage Visualization: Lesson Learnt 77

Session 5: Bugs and Software Quality 82

Giulio Concas, Michele Marchesi, Cristina Monni, Matteo Orru’ and Roberto Tonelli:
Predicting Software Quality through Network Analysis 82

Haidar Osman:
Null Check Analysis 86

Boryana Goncharenko:
Detecting Violations of CSS Code Conventions 89

Tim Molderez and Dirk Janssens:
Design by Contract and Modular Reasoning in Aspect-Oriented Languages 92

Session 6: Empirical Studies and Industrial Experience 96

Maëlick Claes:
An Empirical Study of Identical Function Clones in CRAN 96

Christophe Ponsard, Jean-Christophe Deprez and Dimitri Durieux:
Driving the Evolution of Cloud Software towards Energy Awareness 100
Mathieu Goeminne:

ON THE USE OF JAVA DATABASE FRAMEWORKS IN JAVA PROJECTS – A LARGE-SCALE HISTORICAL EMPIRICAL ANALYSIS

Ali Parsai, Quinten David Soetens and Serge Demeyer:

MUTATION TESTING: AN INDUSTRIAL EXPERIMENT
Keynotes and Tutorials
MANAGING SOCIO-TECHNICAL DEPENDENCIES IN DISTRIBUTED SOFTWARE DEVELOPMENT

Anita Sarma, University of Nebraska, Lincoln

Abstract. Distributed software development poses many challenges. One of the main factors is the complexity of technical dependencies existing in the code base, which leads to complex social-technical dependencies among developers. Furthermore, as the software evolves, so do these dependencies. This social-technical complexity inevitably leads to software conflicts because of coordination problems. In this talk, I will discuss the challenges that evolving software dependencies pose for coordination, especially in the context of distributed, parallel development. Next I will present different coordination tools that have been developed, some by my group, to facilitate distributed software development. For these tools I will discuss their key selling points and the problems they do not address. I will conclude with a set of research challenges that exist in coordination in software development, especially in distributed development settings.

ANALYZING CHANGES IN SOFTWARE SYSTEMS: FROM CHANGEDISTILLER TO FMDIFF

Martin Pinzger, University of Klagenfurt

Abstract. Software systems continuously change and developers spent a large portion of their time in keeping track and understanding changes and their effects. Current development tools provide only limited support. Most of all, they track changes in source files only on the level of textual lines lacking semantic and context information on changes. Developers frequently need to reconstruct this information manually which is a time consuming and error prone task. In this talk, I present three techniques to address this problem by extracting detailed syntactical information from changes in various source files. I start with introducing ChangeDistiller, a tool and approach to extract information on source code changes on the level of ASTs. Next, I present the WSDLDiff approach to extract information on changes in web services interface description files. Finally, I present FMDiff, an approach to extract changes from feature models defined with the linux Kconfig language. For each approach I report on cases studies and experiments to highlight the benefits of our techniques. I also point out several research opportunities opened by our techniques and tools, and the detailed data on changes extracted by them.
Empirical Software Engineering

Massimiliano Di Penta, University of Sannio

Abstract. Empirical research is of paramount importance in any field of software engineering, as it helps to gain evidence of phenomenon occurring in software products or processes, as well as in conducting appropriate evaluation of existing approaches and tools. The goal of this tutorial is to provide a general overview of principles and techniques needed when conducting empirical studies in software engineering, with particular emphasis on controlled experiments and case studies. More specifically, the tutorial will provide an overview of the main principles for empirical study planning and design and, more importantly, it will provide some practical insights about how to perform suitable statistical analyses on the results of empirical studies or empirical evaluations. Such an overview will be provided using available datasets and showing how to perform statistical analyses using the R environment.
The MetricsGrimoire toolset
A mature, industrial-strength toolset to obtain data from software repositories

Gregorio Robles
GSyC/LibreSoft
Universidad Rey Juan Carlos
grex@gsyc.urjc.es

Abstract

In this tutorial we will introduce the MetricsGrimoire toolset, a mature, industrial-strength toolset to obtain data from software repositories. The MetricsGrimoire toolset is the results of over 12 years of development. Its development started 12 years ago at the Universidad Rey Juan Carlos. With the creation in 2013 of Bitergia, it is now the technological basis of a start-up that offers results and analysis to free/open source software companies and communities.

1 A little bit of history

MetricsGrimoire has its origins in the GSyC/Libresoft research group at the Universidad Rey Juan Carlos as a toolset named LibreSoft Tools. After several years of development, a small community started to grow around it, including developers and users from all around the world.

Bitergia, a spin-off company founded from the research group, provides professional services around these tools. In parallel to this process, the LibreSoft Tools evolved into a more neutral, community-oriented project: MetricsGrimoire.

2 About MetricsGrimoire

MetricsGrimoire is a toolset to obtain data from repositories related to software development: source code management (aka version control), issue tracking (aka bug reporting) systems, mailing lists, etc. Data and metadata about the software development processes is retrieved from those repositories (information about commits, ticket management, communication in mailing lists, etc.), and then organized and stored into SQL databases that can later be mined for specific patterns or summaries of activity.

The MetricsGrimoire tools support many kinds of repositories, including those provided by GitHub (git and GitHub issue tracking). It has been already used to analyze many different projects, and together with visualization tools like VizGrimoire it is possible to get dashboards or reports like the ones provided by Bitergia.

The following is a description of the current set of tools with a brief description. This is a live project, so if you are interested in more information, we urge you to visit the webpages of the tools online.
The majority of tools is written in Python and use a SQL (usually a MySQL or a Postgres) database. Some external libraries may be required to be installed in the system prior to use the tools; a description of these dependencies is always given in the tool page.

All tools are developed in the open and currently are hosted in GitHub. Anyone is invited to join the community, to share code, comments and results.

In general, the tools presented do the (hard) work of parsing the data sources and storing the data into a database that can be used to extract meaningful data by performing queries. Because what is meaningful varies from analysis to analysis and from project to project, the flexibility of being able to write SQL queries is a requirement to properly use the toolset. However, vizGrimoire (see 2.4) offers the possibility to obtain an already valuable analysis scenario.

2.1 CVSAnalY

CVSAnalY retrieves and organizes information from source code management (version control) systems. It uses the RepositoryHandler library for the interactions with the repository, so it supports CVS, Subversion, git and other types of repositories. CVSAnalY can be extended with various extensions, some of them delivered with the application itself.

The database schema of CVSAnalY¹ is offered in Fig 1, although we advise to better look at the tables after using it as the tool is continuously evolving.

![Figure 1: CVSAnalY database schema.](image)

Once installed, CVSAnalY has to be run in the top directory of a local repository. For instance, if we had a local repository of a project in the `project` subdirectory, the commands to introduce in the shell would be:

```
$ cd project/
~/project$ cvsanaly2
```

This would fill a database with data extracted from the versioning system (which is automatically identified). This is the basic way of using CVSAnalY; options can be seen by introducing `cvsanaly2 --help` in the shell.

---

2.2 Bicho

Bicho retrieves and organizes information from issue tracking systems. It is a command line-based tool used to parse bug/issue tracking systems. It gets all the information associated with issues and stores them in a relational database. Currently Bicho supports:

- Bugzilla
- Sourceforge.net (abandoned)
- JIRA
- Launchpad
- GitHub
- Maniphest
- Redmine
- Gerrit (unstable)
- Google Code (abandoned)

The general format of a bicho command in the shell is:

```
```

For more guidance, please see doc/UserManual.txt.

It is very important to use a delay. If you run Bicho against big sites without a delay between bug queries, your IP address could be banned!

2.3 MailingListStats

Mailing List Stats (mlstats) is a tool to parse and analyze mail boxes. It is useful to retrieve and analyze mailing list archives. The parsed mail boxes are stored in a database. mlstats is able to retrieve mailing lists from the web, and store the data of every email in a database, from where to obtain different kind of reports. It supports both archives as local files in mbox format or web-accessible Mailman archives. The mlstats database schema can be found in Figure 2.

Assuming you have a MySQL database running on localhost, you might run mlstats with these commonly used options (replace the text in ALL CAPS with your db username, db password and mailing list URL):

```
$ mlstats --db-user=USERNAME --db-password=PASS http://URLOFYOURLIST
```

If you have a different configuration or need more options, more detailed information about the options, can be obtained by running mlstats --help.

2.4 VizGrimoire

vizGrimoire is a toolset and framework to analyze and visualize data about software development. Currently, it is focused on data produced by the MetricsGrimoire tools (CVSAnalY, Bicho and MailingListStats).

Figure 3 provides an overview of the vizGrimoire architecture.

Figure 4 offers an example preview of the type of visual output that can be obtained using the vizGrimoire tool. In the case of the Figure, it is part of the results from the OpenStack project, one of Bitergia’s customers. More dashboards can be seen online at http://bitergia.com/projects.html.

There are two ways of obtaining these type of dashboards: vizGrimoireR, the now outdated scripts that allow to analyse data in databases produced by MetricsGrimoire using R, and vizGrimoireJS, the JavaScript framework to create dashboards and interactive reports and visualizations.

2.5 Other

In the following, other tools are presented that are not yet mature or which are not end-user tools, but in which a lot of effort is put in recent times.

2.5.1 Repository Handler

RepositoryHandler is a Python library for handling code repositories through a common interface. The supported repository types are following:

- CVS
- SVN
- GIT
- BZR (Preliminary support)
- Tarball

Usually, RepositoryHandler is not used directly, and may only be interesting if you want to perform a different analysis as the already existing tools offer.

2.5.2 CMetrics

CMetrics extracts some measures (size, complexity, etc.) from C code. It was adopted by Israel Herraz for his thesis on software evolution, importing the C Metrics package with permission of the original author, Brian Renaud. The metrics are the following:

- SLOC: Source Lines of Code (text lines excluding blanks and comments)
- LOC: Text lines in the source code file (including blanks, etc)
Figure 3: vizGrimoireJS architecture.

- BLANK: Number of blank lines
- COM.L: Number of lines that are exclusively comments (no code)
- COM.N: Number of comments in the file (a comment can be multiline)
- H.LEN: Halstead’s Length
- H.VOL: Halstead’s Volume
- H.LEVEL: Halstead’s Level
- H.MEN.D: Halstead’s number of mental discriminations
- MAXCY: Maximum McCabe’s cyclomatic complexity (between all the functions)
- MINCY: Minimum McCabe’s cyclomatic complexity
- AVGCY: Average McCabe’s cyclomatic complexity
- MEDCY: Median McCabe’s cyclomatic complexity
- TOTCY: Total McCabe’s cyclomatic complexity (sum of all functions)

### 2.5.3 Sibyl
Sibyl aims at extracting information from question and answer sites and storing it into a database.¹

### 2.6 IRCAnalysis
IRC Analysis Tool² is a tool that gathers data from IRC channels, which are often used by free software projects to communicate synchronously.

---

¹https://github.com/MetricsGrimoire/Sibyl
²https://github.com/MetricsGrimoire/IRCAnalysis
2.7 Sortinghat

Sortinghat\(^5\) is a tool to manage identities. It allows to merge several identities with a unique person (i.e., developer) and to add additional information to the developer, such as its affiliation, etc.

2.8 Eventizer

Eventizer\(^6\) is a Python tool created by Bitergia to retrieve information from Meetup groups. It stores the data in a MySQL database.

3 Summary and future

This paper has presented an overview of the tools used by the GSyC/LibreSoft research group at the Universidad Rey Juan Carlos and by Bitergia to analyze software development. Given the fact that Bitergia bases its business model on offering the best data in the best way to its customers, and that the tools are developed in the open, it is a good opportunity for researchers to use state-of-the-art, industrial-strength tools for their analysis. We encourage to join this lively community and to share ideas, results and code.

Acknowledgements

The author would like to thank the Bitergia team and all the volunteers that have worked on the MetricsGrimoire toolset. Much of the information included in this paper has been obtained from the websites of the tools, so all Bitergians and other contributors should be considered as well as authors of the paper.

This work has been funded in part by the Spanish Gov. under SobreSale (TIN2011-28110) and by the Comunidad de Madrid under “eMadrid - Investigación y Desarrollo de tecnologías para el e-learning en la Comunidad de Madrid” (S2013/ICE-2715).

\(^5\)https://github.com/MetricsGrimoire/sortinghat
\(^6\)https://github.com/MetricsGrimoire/eventizer
Contributed Talks
I. Introduction

JavaScript has become ubiquitous on server and client tiers of contemporary web applications. JavaScript offers many dynamic and reflective features (e.g., runtime code evaluation with eval, scope chain modification through with, a dynamic inheritance hierarchy) which makes it hard to analyze JavaScript programs statically [6], [4]. However, dynamic analyses such as performance profiling [5], taint analysis [8] and concolic testing [10] do not suffer from the same limitations and remain applicable.

A well known technique to implement dynamic analysis is program instrumentation which consists in inserting analysis-related code into the target program. Program implementation has the advantage of being independent from language implementations which is of prime importance considering the pace of browsers evolution. Several instrumentation platforms exist for JavaScript programs [2], [3], [7] which let the analyst override language-level operations. Although powerful, those techniques commonly suffer from three shortcomings: (i) Since they rely on analysis of post-mortem data obtained through execution of programs, they cannot implement deployment-time analyses such as detection of cross site scripting vulnerabilities; (ii) Analysts have to implement low-level concerns such as detecting when a value is polluted with analysis data and clean this value up before sending it to functions external to the analysis; (iii) High knowledge of JavaScript global object is required to correctly intercept all the operations relevant to the target analysis. For instance, in JavaScript, `Object.defineProperty(o, "a", {value:x})` would trigger a generic `apply` trap whereas it has a semantic similar to `o.x = x`.

An alternative to instrumentation platforms consists in using reflective APIs which does not suffer the same shortcomings: (i) Deployment-time analyses can be implemented through reflection since it happens online; (ii) Low-level concerns are being take cared of by reflective APIs when they provide a clean separation between the base and the meta layer; (iii) It is easier to implement analyses right since reflection APIs match semantic operations instead of language constructs. Unfortunately current JavaScript reflection capabilities fail to match important dynamic analyses requirements:

1) For performance reasons, language-level operations can only be intercepted for objects and not for primitive values such as strings [9]. Consequently, JavaScript reflection is not directly applicable for dynamic analyses such as concolic testing that are concerned with the manipulation of primitive values. Although Austin et al. addressed this issue in [1], they rely on a modified version of a JavaScript engine which impairs the applicability of their approach and goes out of the scope of this paper.

2) Code location is not available at runtime; in JavaScript, it is not possible to query the current code location. Therefore it is not possible to include code location into analysis’s feedbacks and decision makings. For instance, object profilers cannot pinpoint code regions which feature high access rate and security analyses cannot differentiate operations originating from trusted code from the one originating from untrusted third-party scripts.

3) There exists no ad hoc construct to shadow the global object and provide analysis-specific implementation of built-in functions. For instance, `Array.prototype.map` have to be threatned as black box whereas it can easily be implemented using regular JavaScript array accesses. Expressing elaborate high-level built-in functions in terms of lower level built-in functions can improve the precision of dynamic analysis such as concolic testing.

In this paper we propose a new instrumentation platform for JavaScript which incorporates a meta-programming API that tackles the shortcomings enumerated above. First, every JavaScript value can be interceded in a way that looks like regular JavaScript object proxies [9]. Second, the precise syntactic node where the interceded operation occurred is given as argument to the triggered trap. As such, access to the AST is provided. Third, the analyst may provide a JavaScript value that will faithfully mock the global object for the target application. While the target program cannot reason about itself; our platform acts as a meta-layer that reasons about the target code. Although our implementation is largely dependent on ECMAScript5 specifications, we believe that the approach we made can be applied to other languages as well.

II. Motivating Example

In this section, we demonstrate the capabilities of our approach by providing three implementations of taint analysis requirements which consist in the following:

1) Untrusted third party code are bundled into the target web application inside specific labeled statements `untrusted: (...)` so that it is easy to decide whether an AST node is to be trusted or not:
2) Values created within untrusted code should be taint-ed; created values include literals and results from external functions.

3) On calls to external functions, taint should be propagated to the resulting values and the arguments. Note that arguments need to be tainted as well to preserve the soundness of the analysis when external functions feature side-effects — e.g.: `Object.defineProperty` and `Array.prototype.push`.

4) Tainted values should not be passed to external functions that are marked as dangerous. For instance:

```javascript
function external (target, loc, ctx, ast) {
    var decorated = Meta(target, test, external);
}
```

Note that the upcoming JavaScript intercession API, namely harmony proxies, are not applicable in this scenario because they cannot wrap primitive values and because they do not provide access to the AST.

### A. Global Traps

In the base layer of our API, we provide global traps for language-level operations as well as a mean to shadow the global object:

```javascript
1 exports.primitive = function (prim, ast) { ...};
2 exports.function = function (fct, ast) { ...};
3 exports.test = function (fork, ast) { ...};
4 exports.apply = function (fct, self, args, ast) { ...};
5 exports.shadow = ...;
```

The primitive trap is triggered on primitive literals such as `null` and numbers. The function trap is triggered on function literals. The test trap is triggered on conditional structures. The apply trap is triggered on function calls and on language-level constructs that can be represented as calls — e.g: `a` is replaced by `Reflect.get(o, "a")` and `x + y` is replaced by `Binary.plus(x, y)`. Finally, the shadow export can be used to specify a value that will mock the global object. The snippet below utilizes this API to provide a first implementation of our taint analysis requirements:

```javascript
1 var wrappers = new WeakSet();
2 function wrap (value, ast) {
3     var wrapper = {inner:value, taint:trusted(ast)?null:ast};
4     wrappers.add(wrapper);
5     return wrapper;
6 }
7 function propagate (taint, value) {
8     if (wrappers.has(value)) value.taint = taint;
9 }
10 exports.function = wrap;
11 exports.primitive = wrap;
12 exports.test = function (fork, ast) {
13     return Boolean(wrappers.has(fork) ? fork.inner : fork);
14 }
15 exports.apply = function (fct, self, args, ast) {
16     if (wrappers.has(fct)) return fct.inner.apply(self, args);
17     var taint = trusted(ast)?null:ast;
18     function unwrap (value) {
19         if (wrappers.has(value)) return value;
20         if (value.taint) taint = value.taint;
21         return value.inner;
22     }
23     var rawSelf = unwrap(fct);
24     var rawArgs = args.map(unwrap);
25     if (taint && blacklist.indexOf(fct) !== -1)
26         throw new Error('Value tainted at "'+taint.loc.start.line+':
27             "taint.loc.start.column" prevented to be passed to "+fct
28                 .name";
29     if (taint) {
30         propagate(taint, self);
31         args.forEach(propagate.bind(null, taint));
32     }
33     return wrap(fct.apply(rawSelf, rawArgs), taint);
34 }
```

There is multiple problems with this implementation. First, it features code at lines 1, 8, 14, 17 and 20 for ensuring a proper stratification between analysis data and data from the target program. Such intricate mixing between analysis-specific concerns and stratification concerns make the code more difficult to maintain. Second, the unwrapping strategy defined at line 19 does not suit every external function. For instance, `JSON.stringify({a:1})` should return `{"a":1}` whereas the simplistic shallow unwrapping defined at line 19 will make it return `{"a":{"inner":1,"taint":null}}` . Third, re-wrapping carelessly the result of external functions might create undesired wrapper duplications that can be exploited to by-pass the security policy. For instance, the snippet below exploits this loophole by using the external function `Object.defineProperty` which returns its first argument:

```javascript
1 ... // we assume the wrapper of obj1 is not yet tainted
2 var obj2 = Object.defineProperty(obj1, key, descriptor);
3 untrusted: { obj2.a = '00FF' } // taint obj2's wrapper
4 eval(obj1.a); // obj1's wrapper is intact
```

### B. Value-Centric Traps

The problems listed above can be addressed by providing an additional layer that factorizes low-level concerns out of the analysis-specific code. First, this layer provide proper unwrapping / re-wrapping strategy based on an extensive case analysis of the external function being applied. Second it ensures robust stratification by adopting value-centric traps much like harmony proxies [9].

In our approach, analysts can decorate values of any type with two traps: the test trap which is triggered when the value is used inside a conditional statement and the external trap which is triggered when the value is accessed by an external function. The external trap features two specific arguments: `loc` and `ctx`. `loc` is a data structure containing information about the trap activation `ctx` is a plain JavaScript object that serves as a communication channel between the different traps triggered by the external call. These decorated values can be created as follow:

```javascript
1 function test (target, ast) { ... }
2 function external (target, loc, ctx, ast) { ... }
3 var decorated = Meta(target, test, external);
```
Note that traps can be reused across multiple meta values because target is always passed as first argument. The second part of our layer intercept value creations which consist in primitive literals, function literals and result from external calls:

```javascript
1 exports.primitive = function (prim, ast) { ... };
2 exports.function = function (func, ast) { ... };
3 exports.external = function (ext, exe, ctx, ast) { ... };
```

The arguments given to the external export are (i) ext: the external function (ii) exe: a procedure that will execute the external call, (iii) ctx: the context passed to the argument traps (iv) ast: the syntactic node where the external call happened. Below is the second implementation of our tame analysis requirements using value-centric traps:

```javascript
1 function test (target, ast) { return Boolean(target.inner) };
2 function external (target, loc, ctx, ast) {
3 if (target.taint) ctx.taint = target.taint;
4 if (!ctx.targets) ctx.targets = [];
5 ctx.targets.push(target);
6 return target.inner;
7 }
8 function wrap (value, ast) {
9 var target = (inner: value, taint: trusted(ast)?null:ast);
10 return Meta(target, test, external);
11 }
```

Thanks to factoring out low-level concerns, the second implementation is cleaner and more concise. However the analysis remains very conservative: every time a tainted value is used inside an external function, the taint is automatically propagated to all the arguments as well as to the return value. For instance, setting a tainted value into an untrusted object propagates the taint to the entire object and its return value. For instance, setting a tainted value into an untrusted object propagates the taint to the entire object and keys: a tainted object produces tainted keys:

```javascript
1 exports.primitive = wrap;
2 exports.function = wrap;
3 exports.external = function (ext, exe, ctx, ast) {
4 var taint = trusted(ast)? ctx.taint : ast;
5 if (taint && blacklist.indexOf(ext) != -1)
6 throw new Error("Value tainted at " + taint.loc.start.line + "\n" + taint.loc.start.column + " prevented to be passed to " +
7 .name);
8 if (taint)
9 ctx.targets.forEach(function (t) { t.taint = taint });
10 return exe(function (res) { return wrap(res, taint) });
11 }
```

C. Shadowing of the Global Object

Shadowing of the global object can be used to implement precise dynamic analysis without diving into complex case analysis over JavaScript built-in functions. The idea is to provide an accessible JavaScript implementation of non-fundamental built-in functions so that dynamic analyses can automatically reason about them. For instance, the entire meta-object protocol of JavaScript can be implemented on top of three very low-level functions: get(obj, key), set(obj, key, val) and keys(obj). The snippet below is an adaptation of our second implementation which makes use of this minimal interface to support precisely the entire JavaScript meta-object protocol without any additional knowledge of it:

```javascript
1 var obj = { a: "safe" };
2 untrusted: { Reflect.set(obj, "b", "BOOM") } // taint a
3 eval(Reflect.get(obj, "a")); // taint error yet safe call
```

III. Conclusion and Future Work

We presented a novel approach for supporting developers into building dynamic analysis tools for JavaScript. We focused our efforts onto helping analysts to properly interface analysis data with external functions. We showed that global dispatching traps, as often proposed by instrumentation platforms, are not appropriated for implementing real-world dynamic analysis tools. The first problem is that they mix analysis-specific logic with lower-level concerns such as stratification, unwrapping and rewrapping. The second problem is that they do not help analysts reason about the complexity of some of JavaScript built-in functions. We believe that the state-of-the-art have consistently overlooked these issues which impairs its applicability. We solved the first issue by providing value-centric traps which factorizes out low-level concerns. We proposed to tackle the second problem by expressing elaborate functions from the meta-object protocol in term of fewer simpler functions. We are currently actively developing our prototype\(^1\) and are looking for a proper way to validate our work.

\(^1\)Prototype available at: https://github.com/lachrist/aran.
References

MacroRecorder: Recording and Replaying
Source Code Transformations

Gustavo Santos, Anne Etien, Nicolas Anquetil
and Stephane Ducasse
RMoD Team, INRIA Lille Nord Europe
University of Lille, CRIStAL, UMR 9189
{firstname.lastname}@inria.fr

Marco Tulio Valente
Department of Computer Science
Federal University of Minas Gerais
mtov@dcc.ufmg.br

Abstract

During its lifetime, a software system is under continuous maintenance in order to improve its structure, to fix bugs, or to adapt to new APIs, for example. In such cases, developers sometimes systematically perform sequences of the same code changes (e.g., create a class, then extract a method) on groups of related code entities. In this paper we introduce MacroRecorder, a tool that records and replays a sequence of source code changes. We also provide discussion on ongoing work about this topic.

1 Introduction

Software systems must constantly evolve to continue to be used. Evolution can be achieved from day-to-day activities, as developers add features, improve the system’s maintainability, fix bugs, and adapt to new APIs. During such activities, developers often perform sequences of source code changes in a systematic way [Men13]. These sequences are composed of small code transformations (e.g., create a class, override a method, and update a statement) which are applied to diverse but similar code entities (e.g., some classes in the same hierarchy).

The manual application of these changes can be (i) tedious, due to the repetition of similar but not identical transformations; and (ii) error prone, due to the detection of candidate entities and application of the correct sequence of transformations. Previous work tackle this problem by storing the sequence of transformations and making it replicable for further application.

In this paper, we introduce MacroRecorder, a tool which allows the developer to (i) record once a sequence of source code changes; (ii) store and generalize this sequence of changes, in order to afterwards (iii) apply automatically these changes to different code entities. For the application, the developer can point out the code entities explicitly or specify an application condition.

MacroRecorder is based on the principle of “build once, use often” to execute reusable code transformations. The tool is based on previous work on the existence of repetitive code changes in real software systems [San15]. We discuss ongoing work on MacroRecorder, which is currently under development. The paper is structured as follows. In Section 2 we present our approach and tool. In Section 3 we discuss related work. In Section 4 we discuss future work, and we conclude the paper in Section 5.

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


20
2 MacroRecorder

MacroRecorder implements an approach of recording and replaying sequences of code changes, which is depicted in Figure 1. The approach requires the existence of an extensible development tool and a code transformation tool. It provides one or more executable change operators in the transformation tool for each code change generated in the development tool. This adaptation is highlighted in grey in Figure 1.

The approach consists in three steps: adapting the development tool to record code changes (Section 2.1); providing an environment to configure and generalize code transformations (Section 2.2); and providing support to replay the sequence of changes in different code entities (Section 2.3). We discuss specific contributions of MacroRecorder in Section 2.4.

![Figure 1: Overview of MacroRecorder's approach (highlighted in grey)](image)

2.1 Recording changes

The current prototype of MacroRecorder is developed in Pharo, but it can be applied to other IDEs such as Eclipse, for example. The tool relies on Epicea [Dia15], which extends the Pharo IDE to record code changes. Thereby, MacroRecorder is a client of Epicea, in which the developer explicitly indicates when the tool must record changes made to the code. The recording process is invisible to the developer. When recording, the developer can resume editing code until the change example is finished.

Figure 2 shows the recording part of MacroRecorder (left part, Operators List). The recorded changes in Figure 2 represent the deletion of a method (named `classDecl`), followed by the deletion of a class (named `PDASTClassDeclNode`). Epicea’s change events are first-class entities, which means that MacroRecorder has a reference for each of the resulting modified code entities.

2.2 Generalizing code transformations

In order to parameterize the recorded changes, MacroRecorder generates a transformation operator for each Epicea event. Specially, a sequence of code changes is stored as a composite operator. Each transformation operator has a parameter resolver. A resolver describes the parameters necessary to replay the transformation operator. For example, a “remove method” operator requires the method itself. We address the resolver in the next section to describe the transformation replication process.

Figure 2 shows the configuration part of MacroRecorder (right part, Parameters List). In the recording process, MacroRecorder adds the modified entities to the resolver. These entities are set as default values. Specifically for Pharo, the method can be obtained from one class and the name of the method, which are described in Figure 2 as `@method1` and `@class1`.

In the generalization process, the developer can configure parameters with different values. In Figure 2, the developer specified the name of the method (now named `caseStatement`). Moreover, the developer also specified that the name of the class derives from the previously removed method (i.e., add “PDAST” as a prefix

---

1[www.pharo.org]
2For Java, the types of the parameters would be also needed to identify the method.
Figure 2: Overview of MacroRecorder’s prototype

and “Node” as a suffix). Specifically, MacroRecorder provides parameterization of the transformation by defining expressions with dependent variables, i.e., entities that depend on a parameter.

2.3 Replaying code transformations

Ultimately, a transformation operator is executed with assistance of a transformation tool. MacroRecorder relies on the Pharo IDE to execute transformations. However, similar examples of such tools are the refactoring tools provided by Eclipse. As introduced in Section 2, MacroRecorder specifies, for each Epicea change event, which are the operations that the transformation tool must execute to apply the change automatically. It is worth noting that Epicea features Redo and Undo operations for the recorded changes. The contribution of MacroRecorder is to parameterize these operations in order to apply them in other code entities.

In order to execute the entire sequence of changes, a composite operator executes all its containing transformation operators. Specifically, a transformation operator asks its resolver which are the entities to be modified, and finally calls the transformation tool to execute the transformation with these entities as parameters. If the developer did not generalize the transformation, MacroRecorder will try to replicate the same change as it was previously recorded. Moreover, if there is an execution error in a transformation operator, the whole composite operator is discarded and previous changes are rolled back.

2.4 Contributions

MacroRecorder has three main contributions. First, the developer builds a composite change operator by incrementally composing smaller transformations. The developer is therefore aware of what each recorded change is doing. This contribution addresses the trust of this kind of automated support. Second, the tool allows the developer to explicitly indicate which are the candidate entities to change. The developer is aware that the same sequence of previously manual changes will be executed automatically. This contribution addresses both the tedious and error-prone qualities when applying repetitive manual changes. And third, the tool allows the developer to parameterize and therefore generalize the transformations. This contribution allows variations of the containing transformation operators and the candidate entities they modify. In section 4, we provide discussions about future work on MacroRecorder.

3 Related Work

The most similar related work is presented in LASE tool [Men13]. LASE relies on two or more code change examples from the developer and it creates an edit script. From these examples, the tool infers the context (i.e., the closest unmodified code) to calculate where the change begins. Furthermore, the tool generalizes or specifies changes depending on the differences between the examples. Critics is a similar tool which relies on one change example [Zha15]. As opposed to LASE, it is based on interactive generalization of the example to build a change
Both approaches differ from MacroRecorder in three aspects. First, both tool’s evaluations relied on changes related to code patches, which generally comprise one or few operators. We propose to apply MacroRecorder also to change the software structure. Our previous case studies relied on up to nine operators with higher level of granularity, which included from update statements to changing the hierarchy of classes [San15].

Second, both tools use the context of the change to find new change opportunities. In previous work, we concluded that inferring an application condition from changes is very complex. MacroRecorder allows the developer to indicate where the changes will be applied, either directly or by using a condition which will select the code entities automatically.

And third, LASE suffers from over specification and over generalization of the examples. If the examples are too similar, the resulting edit script will not be able to find small variations of candidate cases. On the other hand, if the examples are slightly similar, the resulting script will be too general and it might be applied to undesired candidates. MacroRecorder’s contribution consists on the parameterization of code entities, which is customizable by the developer. The parameterization is similar to the one proposed in Critics. This contribution leads to setting up more complex composite operators, which we will address in the next section.

4 Discussions
Concerning the current status of this research, the MacroRecorder tool is under development. As a proof of concept, we implemented the generalization step for a small set of code changes: add and remove classes, methods, and variables. As depicted in Figure 2, we already used the tool in a real software case (e.g., PetitDelphi). For current and future work, we propose the replication of more complex transformation operators to apply in more real cases. For specific in-method changes, related work dealt with the challenge of replaying changes in methods that are different, except for the code to be modified. In this case, we propose to extract the context of the changed method (similarly to LASE and Critics’ approaches) in order identify the exact piece of code to modify.

In the generalization step, the developer can define the value of a candidate entity as an expression (see Section 2.2). For future work, we propose to extend this feature to allow the developer to also define an application condition. An application condition selects, from all code entities of the system, which ones are candidate to change. Such contribution will allow the transformation pattern to be applied automatically in all desired cases. On the other hand, this definition will increase the complexity of the approach because it requires the developer to formally express why a group of entities should change together.

5 Conclusions
In this paper we presented MacroRecorder, a “build once, use often” transformation tool to generalize and automate code transformation by example. We presented our initial steps and we discussed related work in this topic. We also presented our early prototype and we discussed future work in this tool.

Acknowledgements
Our research is supported by CAPES (Brazil)

References


Reasoning about AST Changes

Reinout Stevens
Software Languages Lab
Vrije Universiteit Brussel, Belgium
resteven@vub.ac.be

Coen De Roover
Software Languages Lab
Vrije Universiteit Brussel, Belgium
cderoove@vub.ac.be

Abstract

Change distilling algorithms provide researchers with concrete change operations that transform a source Abstract Syntax Tree into a target AST. A distilling algorithm outputs a sequence of operations that need to be applied from left to right to transform the source AST into the target one. Unfortunately, directly querying and using these change operations is non-trivial. In our latest work, we introduce a dependency graph based representation of these change operations which allows for change agnostic querying. We integrate these structures into our history querying tool QwalKEKO.

1 Introduction

The mining software repositories (MSR) community performs studies regarding the evolution of software systems. To this end, Version Control Systems have been invaluable in providing the needed information to perform such studies. They provide a series of snapshots the versioned software project went through. Unfortunately, they do not store concrete modifications that were made to the Abstract Syntax Trees of the software project. To this end, a change distilling algorithm can be used to deduce changes made between two revisions of the same file. A change is either an insert, a move, a delete or an update of an AST node. A change distilling algorithm takes two ASTs as input, and returns a sequence of change operations that, when applied in order to the source AST, transforms it into the target AST. These change operations provide the information needed to detect recurring change patterns, detect which parts of source code are most prone to changes etc.

Unfortunately, directly querying these change operations is cumbersome. A researcher needs to be able to express the following characteristics:

1. Express the characteristics of the change.
2. Express the characteristics of the change subject and its surrounding source code.
3. Express source code transformations without knowing the concrete source code changes beforehand.
4. Retrieve the minimal set of changes that implement a specified transformation and remove irrelevant changes.

In order to provide a query language that allows expressing the aforementioned characteristics the following challenges need to be tackled:

Copyright © by the paper’s authors. Copying permitted for private and academic purposes.
1 https://github.com/ReinoutStevens/damp.qwalkeko
1. There is no single way to describe a certain code transformation. Different sequences of change operations can result in the same transformation, and different instances of the same transformation do not result in the same sequence of operations. As such, the user cannot know specify a change pattern by using the change operations outputted by a change distilling algorithm if he does not know the concrete changes beforehand.

2. A change distilling algorithm outputs a sequence of operations that need to be applied in order. Thus, any operation can potentially depend on any predecessor. Removing or cherry-picking an operation cannot be done without checking all predecessors for potential dependencies.

3. A change query tool must be integrated in a history query language in order to easily find interesting revision pairs. Specifying source code characteristics, change characteristics and history characteristics should be done in a uniform language.

2 Approach

We integrate our implementation of a change distilling algorithm called ChangeNodes with our history querying tool QwalKEKO. We propose and have implemented the following extensions:

1. We structure the changes in a Directed Acyclic Graph where two changes are connected if they are directly dependent on each other, meaning the first change needs to be applied before the second change can be applied. Directly dependent means that there exists no change that can be applied after the first one is applied, but that needs to be applied before the second change. Currently, these dependencies only work on a syntactical level, and do not incorporate semantic information.

2. We provide a query language to navigate and query intermediate AST states, retrieved from applying a single applicable change to the previous AST state. This allows a user to describe the pre- and post state of the application of several changes, regardless of the concrete changes that were applied. This allows us to retrieve the concrete changes that performed a source code transformation.

3. We integrate this dependency graph and the navigation of the graph with the rest of QwalKEKO to have a seamless integration so that existing predicates can easily be used.

3 ChangeNodes

We make use of ChangeNodes\[^2\], an implementation of a Change Distilling algorithm. At its heart lies the algorithm presented by Chawathe et al. [1]. This algorithm is also being used by ChangeDistiller [4]. It works on AST nodes provided by Eclipse Java Development Tools (JDT). The main differences between both implementations is that ChangeNodes works on a language-aware representation of Java code, while ChangeDistiller has a language-agnostic representation.

4 Change Dependency Graph

Change distilling algorithms output a sequence of changes that needs to be applied from left to right. As such, a change can potentially depend on every change that was applied prior to that change. In order to cherry-pick changes these dependencies must be incorporated as well. In order to do this efficiently, we convert the sequence of changes to a Directed Acyclic Graph (DAG), in which two changes are connected if they are directly dependent. Currently, we only incorporate syntactical dependencies, and not semantical ones.

This graph allows easy retrieval of change dependencies, and as such changes can much more easily be cherry-picked.

5 Change Querying

Multiple concrete change sequences can result in the same source code transformation. As such, directly expressing transformations using concrete changes is hard, as there is no way to know beforehand which concrete change sequence the distilling algorithm will output.

We use the change dependency graph to apply changes one by one, and as such get intermediate AST states. A source code transformation can then be expressed by describing a pre- and post state, agnostic of the concrete

\[^2\]https://github.com/ReinoutStevens/ChangeNodes
changes that were used. It is easy to get the concrete change operations that implement a transformation once an intermediate AST with the described characteristics is found. One takes the shortest path through the dependency graph that resulted in that state.

In order to navigate these intermediate ASTs we use Regular Path Expressions, which we have previously used to navigate the history of software projects. Describing the source code characteristics of intermediate AST states is done using a logic program query language. In our case we use the language EKEKO [3].

6 Conclusion

We have provided a high-level overview of the problems of querying changes, and how we can solve these problems by providing a change dependency graph, and use this graph to query intermediate AST states. We have implemented this solution by extending the history querying tool QwalKEKO.

References


Evolution of Metaprograms,
or
How to Transform XSLT to Rascal

Vadim Zaytsev
vadim@grammarware.net
Universiteit van Amsterdam, The Netherlands

Abstract

Metaprogramming is a well-established methodology of constructing programs that work on other other programs, analysing, parsing, transforming, compiling, evolving, mutating, transplanting them. Metaprograms themselves evolve as well, and there are times when this evolution means migrating to a different metalanguage. This fairly complicated scenario is demonstrated here by considering a concrete case of porting several rewriting systems of grammar extraction from XSLT to Rascal.

SLPS [16], of Software Language Processing Suite, was a repository that served as a home for many experimental metaprograms — to be more precise, metagrammarware for grammar recovery, analysis, adaptation, visualisation, testing. Around 2012, final versions of such tools were reimplemented as components in a library called GrammarLab [13]: the code written in Haskell, Prolog, Python and other languages, was ported to Rascal [8], a software language specifically developed for the domain of metaprogramming.

Grammar extraction is a metaprogramming technique which input is a software artefact containing some kind of grammatical (structural) knowledge — an XML schema, an Ecore metamodel, a parser specification, a typed library, a piece of documentation — and recover the essence of those structural commitments, typically in a form of a formal grammar with terminals, nonterminals, labels and production rules [12]. Over the years the SLPS acquired over a dozen of such extractors, plus a couple of more error-tolerant recovery tools. Several of them were essentially mappings from various XML representations (XSD, EMF, TXL, etc), implemented — quite naturally — in XSLT [6].

A fragment of such a grammar extractor mapping is given on Figure 1(a). Readers that can overcome the overwhelming verbosity of the XML syntax, can see two templates that match elements \texttt{eLiterals} and \texttt{eStructuralFeatures} correspondingly, and generate output elements by reusing information harvested from specific places within the matched elements. As a language for metaprogramming and structured mapping in general, XSLT is pretty straightforward and provides functionality for branching, looping, traversal controls,
etc, without going too deep into more complex metaprogramming practices such as naturally recursive rewriting systems or bottom-up traversals. It is also worth noting that XSLT is an untyped software language, so there is no explicit validation that all constructs matched and all constructs produced are type safe.

If we assume that all the types from the input as well as the output schemata are expressed as Rascal algebraic data types, and all the named templates invoked in this snippet are successfully mapped to Rascal functions, then these matched templates will be expressed as pattern-driven dispatched functions in Rascal such as the ones shown on Figure 1(b).

During the case studies on the existing XSLT-based grammar extractors, we found out that the following metaprogramming idioms are equally easy to express in XSLT and Rascal:

- matched template and **apply-templates** — a pattern-dispatched call of the general transform function
- named template and **call-template** — a call to a dedicated possibly polymorphic function
- **choose, when, otherwise, if** — pattern-driven dispatch or explicit matching with := if the conditions are too deep
- **for-each** — list comprehensions

The following features were hard to match:

- Empty **whens**: in XSLT, one can easily loop through input elements and in some cases decide to return nothing by performing `<xsl:when test="..." />` — this is realised somewhat awkwardly in Rascal with the classic FP idiom of a “poor man’s Maybe” (a list which is either empty or a singleton) and inlining.
- Library functions: luckily, early versions of XSLT are quite poor with respect to library functions. However, since XSLT is not by design a language for metaprogramming, its functions are also suboptimal for that domain — the conclusion was that finding a close match or writing a wrapper is almost always less preferable than a manual rewrite of the fragment in question.
- Variables: XSLT is a declarative language which allows fake elements that initialise named variables with certain values to be used later. Despite being multiparadigmatic, Rascal clearly distinguishes between Haskell-like straightforward function style and Java-like imperative style.
- Choices: surprisingly, the idiom `<xsl:template match="a|b"> ... </...>` was quite prevalent yet had absolutely no close equivalent in Rascal. Finally, the mapping of such matches was realised with an external function that was called separately for each of the matches.
- XPath: XSLT uses XPath expressions both in matches and access points; Rascal uses different notations for those two paradigms. It always strictly distinguishes between matches possibly yielding a set/list or a single element, while XPath always returns a possibly empty set of nodes which is incorporated in XSLT by implicit looping in some cases and by more unexpected workarounds in others. For example, `<xsl:apply-templates select="a"/>` is a loop, but `<xsl:value-of select="a"/>` is a concatenation — the latter is almost never the intended result.

Apart from these issues and some type inference for the **value-ofs**, the mapping from XSLT to Rascal was quite possible to implement to migrate the bulk of the code and provide the opportunity to finish the job manually. The real extent of the work and the limitations of this approach in general are not yet studied in enough detail. One of the interesting remaining open questions is about the nature of the mismatches between the two metaprogramming platforms — are they intentional? Should the two learn from each other, or from the migration path itself?
Figure 1: The same fragment of Ecore to BNF-like Grammar Format mapping in (a) XSLT and (b) Rascal. Besides the apparent shrink in size and the boost to readability linked to it, the latter fragment is strongly typed and thus can be automatically validated for its grammatical commitments to both the input and the output. Technically, the second fragment is still imperfect in the sense that it does not implement namespaces as types (just as substrings), which leaves a small door for bugs open.
References


---

\(^1\)The authors are given according to the list of contributors at http://github.com/grammarware/slps/graphs/contributors.
Predicting the health of a project?
An assessment in a major IT company

Vincent Blondeau1,2 Sylvain Cresson1
Pascal Croisy1 Anne Etien2 Nicolas Anquetil2
Stéphane Ducasse2

1 Worldline,
Z.I Rue de la Pointe
59139 Noyelles-lès-Seclin, France
2 RMod team, Inria Lille Nord Europe,
University of Lille, CRIStAL, UMR 9189,
59650 Villeneuve d’Ascq, France

Abstract

More and more companies would like to mine software data with the goal of assessing the health of their software projects. The hope is that some software metrics could be tracked to predict failure risks or confirm good health. If a factor of success was found, projects failures could be anticipated and early actions could be taken by the organisation to help or to monitor closely the project. Allowing to act in a preventive mode rather than a curative one. We were called by a major IT company to fulfil this goal. We conducted a study to check whether software metrics can be related to project failure. The study was both theoretic with a review of literature on the subject and practical with mining of past projects data and interview with project managers. We found that metrics used in practice are not useful to assess project outcome.

1 Introduction

IT companies have a significant number of projects creating a lot of data: project metrics, bugs reports, source code, continuous integration artefacts, production metrics, social network communications, etc. With the emergence of big data methodologies, these companies hope that data science and especially statistics could help them to evaluate their project health i.e., their success or their failure. Healthy projects will speed up the expansion of the company while unhealthy ones can lead to its failure. To achieve this goal, one major IT company asked us to find correlation between metrics of their projects and health of these projects. The hope is that the organization could follow the projects evolution and take preventive actions to avoid project failure. Finding the right metrics in the whole data set is challenging and doing it in a preventive way even more.

Some studies have already been conducted in this field. They usually consider open-source projects and close-sources projects from other companies, but as development environments are likely different, they may not apply our case.

In this paper, we make three contributions. First, we did a literature review on project health predicted by data mining. Second, we experimented more than 10 project metrics on 50 real world, close source, projects. Third, by doing interviews with company project managers, we found indicators that could be linked to project health.

The rest of this paper is organized as follow: in Section 2, we will do the review of the literature. Section 3 presents the result of the data mining of the projects. Section 4 is dedicated to the project managers interviews and conclusion will be presented in Section 5.

2 Literature review

As first contribution, we did a review of the literature. Our hope was that previous work could already have found relationships between project health and project metrics. We found papers that studies both open-source projects and close-sources

Copyright © by the paper’s authors. Copying permitted for private and academic purposes.
projects. From these findings, we extracted the three most relevant papers.

Capiluppi and Fernandez-Ramil’s (2007) goal was to find metrics identifying regressions after refactorings. Easing the maintenance can be considered as improving the health of the project. They used eight open-source software project in C++ to make correlations between four code metrics at function level. They found that no metric alone is a good predictor of regressions. Moreover one has to determine for each individual project which combination of metrics could be used. This study is closer to what we aim to do, they do cross-project correlations. Nevertheless, this study is only using open source software.

Nagappan et al. (2006) described how they apply statistical methods to predict bugs after client delivery. This bug number is not uniform between all the part of the software, so it might reveal some development defects and impact the health of the project. They mine five Microsoft projects, written in C++, and correlate 31 code metrics with post-release failure data retrieved on theses projects. By doing statistical analysis, they found, for each project, a set of metrics related to post-release bugs. This set is changing from one project to the other. It is not possible to apply the same set of metrics to all projects. They found the same results that the previous study, i.e., no set of metrics can be a good predictor alone.

So is it possible in some cases to create a statistical model from one project and have good results by applying it to another project. But it is rare and there is no way to know in advance if it is going to work.

Zimmermann et al. (2009) had for goal to predict class defect from both metrics from source code and project environment. They consider the number of post-release bugs to measure the success of the project. They extracted data from 28 datasets both closed-sources from Microsoft projects and open-source. 40 metrics were gathered. For each metric, they computed median, maximum and standard deviation at class level. Their empirical study gave some results: on 622 cross-predictions between project tested, only 3.5% of the couples can predict each other. For instance, some Open-Source Software (OSS) projects are strong predictors for some closed-source projects but do not predict the other OSS. Some OSS projects cannot be predicted by any of the projects in their study. On the closed-source side, they found some projects that can predict other closed-source projects. However, they also found some projects that do not predict other projects. They also found some systems that are predicting each other.

Others studies are focused not on project metrics, but on social coding.

Wolf et al. (2009) study the link between team communications and the result of the integration process after merging of the developed software parts. It is based on the IBM dataset, Jazz. They studied 1 288 build results, 13 020 change sets and 71 000 comments on 47 different team in 4 months. They used 8 metrics representing the exchanged informations. The authors found no unique measure of social network that indicates if the integration process is a success or failure.

Hu and Wong (2013) examine the influence of social metrics on defect prediction. On six releases of NetBeans and seven of Eclipse, they studied the relations between developers thanks to nine metrics on commits. They studied the impact of these metrics on after release defects. The authors found that the developers relationships metrics are not correlated to the number of after release defects.

Menzies and Zimmermann (2013) reference the progress of the predictive analysis applied on software projects. They precise that it is possible to make studies from various data. However, it is impossible to throw conclusion from a project and apply them to all. As they said: “But more recently, the goal of analytics has changed — the research community has accepted that lessons learned from one project can’t always be applied verbatim to another.” The research heads towards local methods, i.e., be applied to only one project.

To summarize this section, it seems possible to find, for a given project, metrics that allow one to do predictions a posteriori. But finding, a priori, a unique metric or a unique combination of metrics that can be applied to all projects seems unlikely.

However, as development environments between companies and open-source are different, we decided to try with the company data and metrics.

3 Data mining

We conducted a statistical analysis on the company projects. Monthly, project leaders fill Excel files containing information on their project about bugs encountered, budget spent, and budget remaining. In these Excel sheets, we have 12 project metrics available for each month related to several category of bug (Critical, major, minor, in qualification, acceptance or production), to the budget, and to the slippage. We used also a metric representing the length of the project name. It is intended as a placebo metric. We will compare all results to this obviously irrelevant metric.

For this company, the project health is related to client delivery. We know that a project succeed if the application is delivered in time, in budget and with the functionalities the client wanted. A project is failing if one of the previous items are missing.
As project slippage metric is the most followed by project leader, we used three metrics related to it i.e., # months in slippage, # days of slippage and if there is slippage or not. We decided to compare these metrics to the metrics related to bug number and budget data.

We used data from 44 projects during the past 3 years. However, only 19 are exploitable because in these, the data are well filled.

By the value of their metrics, several projects can have a great influence on the sample. Statistical methods advise to take out these kind of extreme values to have a better sample to analyse. Two items of the sample were detected as outliers. We moved them aside to conduct the study.

We correlate the 12 metrics two by two in a matrix. The correlation matrix in Figure 1 highlights whether there is a linear dependency between 2 variables.

From this matrix, we inferred three blocks of variables correlated. First, we can infer a strong correlation between all kind of bugs (the darker square at the top left of the matrix), except the number of bugs in production. These bugs are not strongly correlated to the others. It can be due to the fact that it is the final user who found them. The final user didn’t take part into the project requirements elicitation step of the project. It is a new eye on the project. So it seems natural that the number of bugs found is not correlated to the other ones.

Second, as shown in the middle square in the matrix, the budget variables are also correlated: the total budget and the predicted budget. The difference between the initial and final budget seems also correlated to the budget metrics but not to the slippage ones. It might be due to the fact that the bigger a project is, the more difficult it is to predict the budget. A long project is more likely subject to deviations.

Third, it seems also the slippage metrics are significantly correlated together which was foretold (except the difference between the initial and final budget). However, the number of intermediate releases seems correlated to these slippage metrics. It might be the decomposition in group of functionalities that is difficult to determine by the project managers.

Finally, there is no link between the 3 groups of variables i.e., the bugs, the budget, and the slippage. Moreover, our placebo metric has worked as predicted. The length of the project name is not correlated to the other metrics.

In the light of this analysis, we can conclude that there is no link between the slippage and any other studied variable. To summarize, the correlations we found are quite trivial. Like the papers from the literature survey, we are not able to find metrics to explain the project health.

4 Interviews

To complete the project study, we realized some interviews with project managers of the company to get their feeling on what impacts project health i.e., success or failure. We wanted to know what are their problems in developing projects, how they detect them and resolve them. The interviews lasted one hour and were decomposed in two parts. In the first, we presented the research topic to the interviewee, in the second, we let totally open the discussion to get all the experience of
project managers on project success and failure.

We met project managers whose projects were used in the analysis and not. We interviewed four projects managers. The projects they lead are diversified. There are both successful and failed projects.

In these interviews, they identified the following root cause of project failure:

- Delay at the beginning of the project: if the client decides to begin the project later, the project team is already available and the relationship will deteriorate.
- Collaboration between the team and the client: if the team and the client know well each other, the collaboration will work fine and the project is more unlikely to fail.
- Team cohesion: if the members of the project team support each other, the cohesion is stronger and the project has significantly more chance to succeed.
- Understanding of the specifications: if the project team understands what the client says and succeeds to transcribe it in its own technical language, the project will progress easier.
- Knowledge of the functional concepts: if the project team knows, in more, the business concepts of the client, the project has more chances of success.
- Change of the framework during the development: if the technical tools or the framework, that the project team uses, change, it will cost more to the project.
- Experience with the used frameworks: a team with experience on the development tools or frameworks they use for their application, will be quite capable of doing the project faster.
- Bypass the qualification tests: if the team doesn’t test its application before delivery to the client, by lack of time for example, the client will be unhappy because some functionalities will not work and some tension in the project team will appear.
- High number of bugs listed by the client: as a consequence of the previous item, the client will find more bugs in the application.

These causes of failure are difficult to find in data provided by the project team. That might be why we didn’t find any correlation by applying statistics to the projects.

5 Conclusion

A major company asked us to found metrics that predict project success or failure. We conducted a study to check whether software metrics can be related to project failure. The theoretical study of literature shows that the metrics extracted from a project can’t be used on another one. The mining of data we have done on company project highlights there is no link between project metrics and data. However, the interviews we conducted shows that the metrics linked to success can’t be found by mining project data.

Moreover, as all these studies intervene a posteriori on projects, it seems random for a new project to know which metric or set of metrics uses to assess success. Predictive analysis will not work well if it is not possible to know a priori which statistical model use.

To go deeper, we plan to do a survey to check whether the indicators we identified during the interviews are shared by all the employees of the IT company.

References


Abstract

Software process planning involves the consideration of process based factors, e.g., development strategies, but also social factors, e.g., collaboration of developers. To facilitate project managers in decision making during the project, we develop an agent-based simulation tool which allows them to test different alternative future scenarios. For this, it is indispensable to understand software evolution and its influences. We cover different aspects of software evolution with models tailored towards specific questions. For the investigation of system growth, developer networks and file dependency graphs we performed two case studies of open source projects. This way, we infer parameters close to reality and are able to compare empirical with simulated results. ¹

1 Introduction

To build an agent-based simulation tool aiding software managers in the planning of software development, it is important to get a deep understanding of software evolution processes. Several factors influence how the software evolve, what evolves, and why it evolves. According to Lehman [LR00] finding answers to this questions are the research directions in software evolution. To reduce complexity and parameters, we build different models reflecting different shades of software evolution and related development processes. Since humans – in the shape of developers, users, and testers – constitute a big driver of software evolution, it is reasonable to approach the simulation of software processes agent-based. Agents are autonomous individuals with a behavior specified by certain rules [MN11]. Developers can be considered as active agents changing the passive agents, i.e., software entities.

When tracing aspects of software evolution, we first have to identify influencing factors concerning the question under investigation. Thus, we learn from the past in form of analyzing open source software repositories, which by itself became a large research topic in recent years (e.g., MSR ²). In our approach we combine software quality assurance issues with social and process controlled factors influencing software development. For this, we are interested in examining the contribution behavior of developers as well as the nature of changes and related error-proneness. The knowledge we gain from mining is then transferred into our agent-based simulation model so that we retrieve a concrete instance constructed for answering the specific evolutionary question under investigation. In this paper, we summarize our recent research, which considers system growth, developer collaboration and behavior, and the evolution of software changes.

¹This paper presents a publication summary of [HHG14], [HHGW15], and [Hon15].
2 Approach

In this section, we describe the background and methods which represent the foundations of our work. We comment on methods used for software mining and explain the underlying agent-based model of our approach.

2.1 Software Mining and Analysis

For the estimation of the simulation parameters, we examine open source software projects, which are a gold mine for researchers interested in software evolution and software mining. Since we are interested in different facets of the software development process, we collect data from projects, for which the information of commit logs, issue tracking systems, and mailing lists are available. Once the project is selected, we use tools from data mining and analysis, machine learning, and visualization to first understand it and later observe behavioral rules from it. These rules and information then serve us as input for our simulation model.

2.2 Agent-Based Simulation Model

The current agent-based simulation model for software evolution is an extension of our previous publication [HHG14] which considers only the system growth of the software under simulation. For modeling and simulation purposes we are using Repast Symphony.

The model depicted in Figure 1 contains the environment which knows all other instances and is responsible for the creation of a configured number of developers at simulation start-up. Furthermore, the environment instantiates bugs at scheduled points in time and assigns them to randomly selected software entities. The developer is responsible for creating, updating, and deleting entities. For the estimation of parameters we used K3b in our initial case study. Through the mining process we have recognized four different types of developers.

The Core Developer is the initial contributor being familiar with many entities and performing most commits. The Maintainer is a person who does primarily maintenance work, i.e. he fixes a large number of bugs. Therefore, we assume he has good knowledge about the entire project. The Major Developer knows specific areas of the project and fixes most of the bugs occurred in entities known by him. The Minor Developer executes less than 100 commits and performs less bugfixes. They might be specialists who only implement one specific task or feature. In K3b there is one core developer, one maintainer, 17 major developers, and 106 minor developers.

To model dependencies between the agents, we have created three networks. One to represent dependencies between developers and software entities, one stores information about bugs and the modules they are assigned to, and one represents dependencies between software entities that are changed together several times.

---

3http://repast.sourceforge.net/ [last visited: 27.04.2015]
4http://www.k3b.org/ [last visited: 27.04.2015]
3 Performed Analysis and Case Studies

We briefly summarize two of our case studies and results in this section. These studies include the system growth in number of files, developer collaboration depicted in developer-file networks, and the evolution of software dependencies represented in file networks based on change coupling.

3.1 System Growth and Developer Collaboration

One aspect of our preliminary case study [HHG14] is the growth of software systems. We measure growth in number of files, which is reflected by the creations, modifications, and deletions the developers perform. For this purpose, we selected K3b with a development of over ten years, 125 developers and more than 6000 commits. We observed a super-linear growth trend for K3b and used this to build a statistical model for the growth based on changes made by developers. Using geometric distributions for file creations, modifications, and deletions we were able to reproduce the system growth in number of files of K3b validated by comparing empirical and simulated results.

Moreover, we build developer-file networks, where a dependency between a developer node and a file node is added, if the developer worked on that file. The evolution of the graph depicted in Figure 2 shows that there is one main contributor who is the project creator, whose central status is inherited by its maintainer after 2006.

3.2 Software Dependency Analysis

In our latest work [HHGW15], we analyzed change coupling dependency graphs under the purpose to understand the evolution of file dependencies. The change coupling [BKPS97] degree describes how often software entities are changed together. By calculating the average degree as well as the average weighted degree over the time, we trace the evolution of the files not only in terms of the amount of dependencies to other files, but also in terms of the intensity of their relationship. For this, we used K3b for the estimation of parameters and Log4j \(^5\) for the validation of our results.

The empirical behavior (red) of the average change coupling degree is shown in Figure 3a. To model this trend we used linear regression and retrieved the best fit for a second order model (black) with an adjusted R-squared value of 0.97 [HHGW15]. In Figure 3b the simulated average degree of the software agents is depicted. By comparing the real with the simulated behavior it is recognizable that the simulation exhibits a similar trend and reaches similar values.

For validation we tried out the simulation built on the knowledge gained from K3b with a changed parameter set according to properties of Log4j. With the adapted distribution of developer types and size of the system, we were able to simulate growth trends and network properties. Therefore, the simulation works for projects which are similar in size and duration.

\(^5\)https://logging.apache.org/ [last visited: 27.04.2015]
4 Lessons Learned and Future Work

During our experiments in simulating software processes, we observed problems due to the lack of development phases, which results in a more linear representation of e.g., system growth, on the simulation side. The need of phases is also relevant for the behavior of developers. Although there are different types of developers in the simulation, they spend the same effort and do not learn from their experiences. To include different learning types of developers, we plan to build a Hidden Markov Model (HMM), which learns from their contribution respecting also communication and bug handling [Hon15]. From this, we hope to refine the developer types and incorporate development strategies according to the developers behavior. This model will be transferred into a learning model for development phases. How suitable parameters look like and if this presents a promising approach for the inclusion of development phases in the simulation, is an open question for us. Furthermore, we plan to suit our simulation to different types of projects and to build models mirroring further facets of the software evolution process.

Acknowledgements

We would like to thank the simulation science center Clausthal-Göttingen (SWZ), that funded parts of our work.

References


Parsing and Analyzing SQL Queries in Stack Overflow Questions

Csaba Nagy, Anthony Cleve
{csaba.nagy, anthony.cleve}@unamur.be

PReCISE Research Center, University of Namur, Belgium

1 Introduction

Stack Overflow (SO) is the main Q&A website of Stack Exchange Network, run by Stack Exchange Inc, where software developers can discuss computer programming questions. It was publicly launched in 2008\(^1\) and since then it has become one of the most popular Q&A sites. Within 2 years, the site reached 300,000 registered users and more than 7 million monthly visits (as of August 2010) [MMM\(^*\)+11] and these numbers still grow rapidly. In September 2014, the site had 3.5M users [LP15] while today, after half a year, it has 4.2M users who have asked 9.3M questions with an answer rate of 74%. Its daily traffic is over 8 million visits a day\(^2\).

This rapid growth and increasing popularity of Stack Overflow made it a large knowledge base of several programming topics which also attracts researchers. To mention a few examples, they study actual trends that developers follow [BTH14], design questions of Q&A systems [MMM\(^*\)+11], island parsing techniques to analyze posts [LP15], recommendation systems [RYY14, PBL13, CAG12], and try to model the quality of the posts [PMB\(^*\)+14, PMBL14].

In our paper, we introduce an approach to parse and analyze SQL queries in Stack Overflow questions with the main goal to identify common error patterns among them. Such similar structures in SQL statements can point to problematic language constructs (e.g., antipatterns) which should be avoided by developers.

Stack Overflow does not tolerate to ask the same question more than once, and recommends its users to double-check before sending a question whether it has been already asked or not. Users, however, cannot always realize that their problem was asked before. Moreover, it can be very hard for a non-experienced developer to recognize that a problem is just a specific instance of a more general one. Members with more reputation can mark a question as a duplicate of another, but even this way the question has been already asked and may have some answers as well before it gets to the attention of moderators. Hence, the same problem can show up several times on the site and patterns potentially exist among them. This fact already inspired books in this area, e.g., as Bill Karwin says it in his SO profile\(^3\), “I’ve written a book, SQL Antipatterns: Avoiding the Pitfalls of Database Programming from Pragmatic Bookshelf, based on the most common SQL problems I’ve answered on Stack Overflow and other forums, mailing lists, and newsgroups over the past 15 years” [Kar10].

2 Overview of the Approach

2.1 Introduction

SO can be considered as a forum for computer programming questions. A user can ask a question from the community and mark one answer as accepted of all the answers given by the experts. Users can vote as well, resulting that the answers with better quality have higher scores and are more likely to be accepted, while poor answers will get less attention and may be removed later. Questions with more upvotes represent better-explained problems, get more attention and are more likely to be important for the community. Users can also gain reputation scores and badges based on their activity and on the scores of their posts.

Questions are tagged in order to keep them organized. For instance, a question about querying a MySQL database in PHP can be marked with the php and with the mysql tags so it shows up for the users who are interested in these topics.

\(^1\)http://www.joelonsoftware.com/items/2008/09/15.html
\(^2\)http://stackexchange.com/sites?view=list\#traffic
\(^3\)http://stackoverflow.com/users/20860/bill-karwin
For us, a question represents a coding issue and potentially contains some problematic SQL code fragments, while answers contain solutions for this issue. Recommendation systems rely on this fact and help developers, e.g., by searching for keywords in the questions, and bringing the features of the site closer to the developers in their IDE [BPL12, PBL13]. These techniques mostly work with NLP or LDA [LP15], and island-parsing techniques [BPL12, PBL13].

Figure 1 represents an overview of the main steps of our approach, which can be considered as an automatic technique where we analyze the SQL code fragments in the questions and try to identify error patterns among them.

2.2 Filtering SQL Posts

The data dump of Stack Overflow is published by Stack Exchange in XML format\(^4\), which contains all the relevant data for our analysis. All user contributions of SO are licensed under cc-by-sa 3.0\(^5\).

The database dump is divided into smaller parts, and the size of the XML which contains the posts was about 29 Gb on September 14, 2014 (which we use in the rest of the paper). It contained over 25M posts (answers and questions both).

First, we filter this database to cover only posts that were marked with the `mysql` tag, because our parser is currently only able to deal with MySQL dialect. This step can be simply done by processing the XML and extracting the questions which belong to this tag, and extract their related answers too.

As a result of the filtering, we got a 1.1 Gb XML file, with about 271,117 questions and 500,607 answers.

2.3 Query Extraction

Once we have the posts which are related to SQL issues, it is possible to extract from these questions the code blocks which contain SQL statements. Here, we extract statements only from the questions (and not from the answers), as these statements are more likely to contain problematic language constructs.

A sample SO question with the `mysql` tag, can be seen in Figure 2. After a title and a short description, this example has a code block containing a select statement. Stack Overflow questions are written in Markdown and code blocks are placed between `<code>` and `</code>` tags. So for this step, we simply extract these code blocks and apply some additional filtering in order to keep blocks where we find keywords of SQL statements, which we can parse actually (we check if `select`, `insert`, `update`, `delete`, `create`, `alter`, etc. appears in the block). This filtering is important, because code blocks tagged as MySQL may still contain non-SQL (e.g., Java, PHP) code too. Ideally, we should have only SQL code blocks later, but some non-SQL codes are likely to get through the filtering, which is still acceptable here, as the parser will drop them anyway if it won’t be able to process them. For the example in Figure 2, it means that we keep only the last code block with the select statement and the filter will drop the first one which is used only to present some sample data.

\(^4\)http://archive.org/details/stackexchange
\(^5\)http://creativecommons.org/licenses/by-sa/3.0/
As a result of this step, we had 564,941 code blocks containing SQL keywords, which is about 2 code blocks per question on average.

2.4 Parsing SQL Queries

After extracting the SQL queries from the questions, the next step is to parse the statements and construct their ASTs. For this purpose, we use our robust SQL parser introduced in our previous work [NMC15].

Our approach can be seen as an island-parsing technique, but with a slightly different goal. An island parser typically parses some recognizable structures (the islands) in a text and does not care about the rest (the water). In our case, we parse only the code blocks of the questions and we want to have a complete AST for statements in these blocks. However, these blocks usually contain some non-SQL text too, which makes some parts of the code unrecognizable for the parser. E.g., imagine typos in the text (e.g., ‘form’ instead of ‘from’) or a typical situation when someone writes ‘…’ instead of a complete field list of a select statement. These unrecognizable code parts represent the water in our case. Hence, we have huge islands with just some little water among them. Also, our goal is to find patterns in the structures of the queries, so we want to keep their original structure. For this reason, we insert special (‘joker’) nodes to the AST in the place of the unrecognized code parts.

As a result of this phase, currently, we were able to construct ASTs for 167,992 SQL statements, which means about 0.62 statements on average for the questions with the mysql tag.

2.5 Pattern Detection

The final step is to detect common patterns among the extracted statements. This step is under development at the moment but we have initial algorithms from our previous work [NMC15], where we implemented a tree-matching algorithm to locate the position in the Java source code, where a given query was constructed and sent to the database. The pattern searching problem is similar, but more general. In the matching problem of the concept location task, we had to match a concrete query to several queries extracted from a Java code base. However, it was not always possible to determine the full SQL statements in Java, that is, we had to match a concrete AST to several ASTs containing ‘joker’ nodes. Here, we have to find patterns as similar subtrees among ASTs containing ‘joker’ nodes.

3 Conclusions and Future Plans

Our research work is currently in an early stage and we could report here the main idea with some early results. We implemented the first steps (data preprocessing/filtering, query extraction and parsing) for MySQL, and
we still need to improve and evaluate our pattern detection algorithm. However, the results from these first,
parsing phases are already promising as we are able to extract over 167k SQL statements which are related to
MySQL coding issues. We expect promising results from the pattern detection algorithms too. There are similar
questions addressing the same issues in the database and not just among the questions which were marked as
duplicates. Some of these similarities can be easily spotted in the database manually, for example, one can search
for the problem in Figure 2 as ‘"NOT IN" optimize [mysql]’ and it gives over 200 results, while an ineffective
use of NOT IN with a subquery, can be simply avoided by the proper use of joins. Initial results show that
automatic techniques are able to identify error patterns in this huge knowledge base which will be useful for
developers working with SQL.

References

[BPL12] Alberto Bacchelli, Luca Ponzanelli, and Michele Lanza. Harnessing Stack Overflow for the IDE. In
Proceedings of the Third International Workshop on Recommendation Systems for Software Engi-

[BTH14] Anton Barua, Stephen W. Thomas, and Ahmed E. Hassan. What are developers talking about?
2014.

[CAG12] Joel Cordeiro, Bruno Antunes, and Paulo Gomes. Context-based recommendation to support prob-
lem solving in software development. In Proceedings of the Third International Workshop on Rec-

[Kar10] Bill Karwin. SQL Antipatterns: Avoiding the Pitfalls of Database Programming. Pragmatic Book-

[LP15] Michele Lanza Luca Ponzanelli, Andrea Mocci. StORMeD: Stack Overflow ready made data. In

[MMM+11] Lena Mamykina, Bella Manoim, Manas Mittal, George Hripcsak, and Björn Hartmann. Design
lessons from the fastest Q&A site in the west. In Proceedings of the SIGCHI Conference on Human

[NMC15] Csaba Nagy, Loup Meurice, and Anthony Cleve. Where was this SQL query executed? a static con-
cept location approach. In Proceedings of the 22nd International Conference on Software Analysis,

[PBL13] Luca Ponzanelli, Alberto Bacchelli, and Michele Lanza. Seahawk: Stack Overflow in the IDE. In
Proceedings of the 2013 International Conference on Software Engineering (ICSE2013), pages

[PMB+14] Luca Ponzanelli, Andrea Mocci, Alberto Bacchelli, Michele Lanza, and David Fullerton. Improving
low quality Stack Overflow post detection. In Proceedings of the 2014 IEEE International Conference

[PMBL14] Luca Ponzanelli, Andrea Mocci, Alberto Bacchelli, and Michele Lanza. Understanding and classi-
fying the quality of technical forum questions. In Proceedings of the 14th International Conference

for recommendation about programming errors and exceptions. In Proceedings of the Software Evolu-
tion Week - IEEE Conference on Software Maintenance, Reengineering and Reverse Engineering

42
Example-driven Model Queries

Carlos Noguera
Software Languages Lab
Vrije Universiteit Brussel, Belgium
cnoguera@vub.ac.be

Viviane Jonckers
Software Languages Lab
Vrije Universiteit Brussel, Belgium
vejoncke@vub.ac.be

Abstract

Model querying is an integral part of Model-Driven Engineering. Developers query models when specifying model transformations, when defining model constraints, or simply when they need to extract some information from the model. Model queries are often specified in a general programming language, with developers just navigating models through their programming interfaces. OCL is the best known model query language, and while successful, it is difficult to express complex structural properties featured by the sought after model elements. In this presentation we describe an example-driven query facility that aims at easing the description of structural features in a query. In our approach, developers can describe their queries in terms of model fragments augmented with variables and special relations. Templates are translated into logic queries which in turn are executed by Ekeko.

1 EMF-Query

Querying models remains an important part of model-driven approaches. Finding relevant elements of a model is instrumental when transforming models, checking constraints, documentation or report generation, etc.

In the case of EMF-based approaches, developers still specify their queries through a general purpose language (Java) although several domain-specific declarative languages exist (OCL [8], EMF Model Query [4] or EMF-IncQuery [1]). Why developers choose to disregard dedicated model query languages is normally rooted in their lack-luster performance, and on the difficulty to express complex queries on them. With our approach we target the second concern. One of the problems with the use of OCL, as identified in [6, 7] is its steep learning curve. In our approach we aim at reducing this learning curve by allowing developers to express interesting features sought as instances of the meta-model under query as templates. Queries in our approach are then composed of model fragments augmented with variables and special relations between model elements. Queries are then translated to a set of logic predicates which are matched against model instances. The matching provides then bindings for every variable, thus providing the results for the query.

The EMF-Query approach we propose is then composed of three main parts: First a logic query language to specify queries over EMF models. A Query meta-model and a model transformation that weaves the query metamodel and the target meta-model so that templates can be expressed. Finally a model-to-text transformation that translates query instances to the logic query language.

Each of these parts is further described and illustrated by means of a running examples in the following sections.

To illustrate EMF-Query we first introduce as running example a simple UML-like class model shown in Fig 1. The meta-model, pictured on the right, defines the concepts of packages, classes, attributes and methods, all of

Copyright © by the paper’s authors. Copying permitted for private and academic purposes.
them extending the concept of a named element. Packages contain classes and classes contain attributes and methods. Attributes have a type, modeled as a class, and classes have a self reference to represent inheritance. In figure 1b, we show an object representation of a simple class model. In it, a package called pack contains four classes: String, Person, Employee and Volunteer. Person contains an attribute name of type String, and Volunteer extends Employee, which in turn extends Person.

1.1 Querying models

The basis of our approach is a logic-based query language. Our language is implemented as an extension to Ekeko [3], an applicative meta-programming language using Clojure’s core.logic library\(^1\).

Our query language offers three basic predicates: eobject ?eo binds ?eo to a model element. eobject-eclass ?eo ?eclass reifies the relation between an eobject and its eClass. has ?eobject ?feature-name ?val reifies the relation between an eobject and the value of one of its efeatures.

From this three predicates, we are able to express queries that structurally navigate models. In our running example, executing

( has ? pack : name "pack")
( has ? pack :content ?eo)
( has ? eo :name "Person")

on the model instance of Fig. 1b will bind the ?eo variable to the Package instance whose name is "pack", and then bind the ?eo variable to an eobject that has an efeature name with value "Person".

While it is possible to write interesting queries with these, and other derived predicates, it is still cumbersome to describe complex relations between elements. For this reason, we build on top of the EMF-Query predicate library to allow the expression of model templates.

1.2 Model Templates

Model templates allow developers to express queries over a target model by using elements from the target model itself, augmented with variables and special relations. The basic idea is to have any model element of the target model conform to the template meta-model. To this end, in order to allow developers to express templates of a given target model, we require two ingredients: A model of the variables and special relations, and a model transformation that will, given the target meta-model produce a template meta-model that describes templates for that model.

Query Meta-Model

Our query meta model (Fig 2a is composed of three main elements: the notion of a Query, Variables contained in that query and special Relations. The Query element keeps a list of EObjects in the query, as well as a list of the variables and relations defined. Each variable has a name, the type of the variable is an EClassifier from the target meta-model. Currently, we define two special relations: Transitive which represents the transitive closure over references (e.g., super in the Simple Class model) and Reachable which closes over all references.

In order to define templates for a given meta-model, we define a new EClass extending each EClass in the target meta-model and Variable from the Query meta-model. This allows templates to substitute a model

\(^1\)For a full description of the Ekeko, we point the interested reader to https://github.com/cderoove/damp.ekeko/
element for its corresponding variable version. Note that this does not require modifications to either the target
metamodel nor the MOF (EMF’s Ecore). Fig. 2b shows the resulting query meta-model for our Simple Class
model running example. Each non-abstract EClass is extended with a Variable version (whose name is prepended
with V).

1.3 Querying with template models

Having defined a query meta-model and a template meta-model for a given target model, the developer can then
construct queries by reusing objects from existing target models to describe the fixed parts of the template, and
variables from the template meta-model to describe what can change in the template.

For our example, the model in Fig. 3a shows a template query that is equivalent to the query expressed in
EkekoEMF of Section 1.1. The model in fig 3b represents a template that searches for direct subclasses of the
class ”Person”, and binds ?sub to the subclass, and ?pack to its containing package. Finally fig 3c shows a
similar query, this time finding all direct and indirect subclasses. In all of the templates in figure 3 the eobject
representing the “Person” class is an instance of the Simple Class Model from Fig 1a.

![Diagram of query meta-model](image)

(a) Query meta-model

![Diagram of query meta-model to query the Simple Class Model](image)

(b) Query meta-model to query the Simple Class Model

Figure 2: Modeling Templates

![Diagram of template queries](image)

(a) ?pack is the package that contains the class "Person"

(b) ?pack is the package that contains the class "Person" and its direct subclasses (?sub)

(c) ?pack is the package that contains the class "Person" and all subclasses (?sub)

Figure 3: Template queries for Simple Class Model

From templates, a model-to-text transformation generates EkekoEMF queries which can then be run over
model instances of the target model.
2 Prototype

We have implemented a prototype tool that uses Xtend [2] to implement the model-to-model transformation that constructs template meta-models out of a target meta model, and the model to text transformation that from instances of a template meta-model produces an EkekoEMF query. In addition to this we are in the process of developing a graphical query editor based on Graphiti [5] to edit and run the template queries.

3 Future Work

We have yet to produce a full validation of our approach, both in terms of expressiveness and of usability. We also intend to enrich the Query model with the inclusion of negated relations and free-form EkekoEMF queries.

References


Weighted Multi-Factor Multi-Layer Identification of Potential Causes for Events of Interest in Software Repositories

Philip Makedonski, Jens Grabowski
Institute of Computer Science, University of Göttingen
Goldschmidtstr. 7, 37077 Göttingen, Germany
{makedonski,grabowski}@cs.uni-goettingen.de

Abstract
Change labeling is a fundamental challenge in software evolution. Certain kinds of changes can be labeled based on directly measurable characteristics. Labels for other kinds of changes, such as changes causing subsequent fixes, need to be estimated retrospectively. In this article we present a weight-based approach for identifying potential causes for events of interest based on a cause-fix graph supporting multiple factors, such as causing a fix or a refactoring, and multiple layers reflecting different levels of granularity, such as project, file, class, method. We outline different strategies that can be employed to refine the weights distribution across the different layers in order to obtain more specific labelling at finer levels of granularity.

1 Introduction
The field of software mining explores different approaches for extracting information from software repositories both in the form of basic facts and in the form of derived knowledge. While software repositories provide a wealth of information related to the development and evolution of software projects, most of it is of empirical nature, that is, describing consequences rather than causes. For example, developers typically describe their development and maintenance activities as fixing issues and problems, improving certain properties, adding features and functionality, and refactoring code. In contrast, during software assessment, we are often more interested in the potential causes for such activities which are typically not explicitly labelled as such due to the fact that such knowledge is usually not available at the time when the corresponding activity was performed.

In this article, we are concerned activities which are associated with (or contributing to causing) various technical risks for undesirable phenomena, such as failures, or difficult to maintain code that needs refactoring. We explore means for the retrospective identification of such activities based on empirical data and different factors contributing to labeling activities as risky. The presented approach can be generalised to labelling activities as potential causes for events of interest with respect to any particular assessment task, regardless of whether it is concerned with a technical risk, or not.

Existing approaches are typically based on some form of origin analysis [GT02], involving line-tracking and annotation graphs [KZPW06], line histories [CC06], line mapping [MHC14], as well as several refinements to
these [WS08, CCDP09] in order to map and track entities across revisions. Different applications for such approaches have been discussed in the literature, ranging from finding fix-inducing changes [SZZ05] and the role of authorship on implicated code [RD11] to defect-insertion circumstance analysis [PP14]. While these are closely related to topic of this article, to our knowledge none of the existing approaches has incorporated weighting of the extent to which a change contributes to a subsequent fix.

This article is structured as follows: In Section 2 we outline the basic notions related to our approach. In Section 3 we discuss the weighting approach and its generalisation for arbitrary factors. In Section 4 we refine the approach to cover multiple levels of abstraction across distinct layers and showcase different strategies for distributing the weights across the layers. Finally, we conclude with a short summary and outlook in Section 5.

2 Causes and Fixes

In this article we are concerned with determining the likely causes for events of interest. Before we proceed, we need to establish what we consider under “events of interest” and other related notions:

Artifact: A generalised notion of an entity $A$ that developers work on at any level of granularity. An artifact may contain other artifacts at finer levels of granularity.

Revision: A generalised notion of a state $R_t$ of entity $A$ at a point in time $t$.

Event of interest: A revision or a state of an artifact which can be described by some directly measurable characteristic, such as the content of a descriptive message associated with the revision, or the value of an attribute of the modified artifact.

Fix: A modification to an existing part of an artifact at a given state $F_t$, that was last modified or created in a previous state $C_{t-n}$. The modification may, but does not strictly need to, relate to fixing a problem.

Cause: A modification of part of an artifact at a given state $C_t$ that was modified in a later state $F_{t+n}$.

Cause-Fix Relationship: A relationship $C \xrightarrow{\text{causes}} F$ between two states $(C, F)$ of an artifact $A$, where a part of $A$ that was modified in $C_{t-n}$ was subsequently modified in a later state $F$, hence $C$ is considered a cause for $F$ ($C \in F^{\text{FIXES}}$ where $F^{\text{FIXES}} = \{ \forall C \in \text{REVISIONS} | C \xrightarrow{\text{causes}} F \}$) and conversely $F$ is considered a fix for $C$ ($F \in C^{\text{CAUSES}}$ where $C^{\text{CAUSES}} = \{ \forall F \in \text{REVISIONS} | C \xrightarrow{\text{causes}} F \}$).

Cause-Fix Graph: A directed graph $G = (N, E)$, where the set of nodes $N$ includes representations for each state of an artifact, and a state may contain other states at finer levels of granularity. The set of directed edges $E$ includes representations for each cause-fix relationship.

A Cause-Fix Graph can be constructed by utilising information extracted from Version Control Systems. This can be accomplished by applying any of the approaches for tracking the location of modified fragments across revisions already described in the literature [WS08, CCDP09]. The resulting graph at the project (or global) level of abstraction represents the cause-fix relationships between states of the whole project. An example for such a graph for three global states ($R_3, R_4, R_5$) is shown on Figure 1.

3 Weights and Factors

A simplified binary classification of nodes in the graph as causes for events of interest presents some limitations. The basic example from Figure 1 already raises two questions related to the significance of the classifications:

- Given that both $R_3$ and $R_4$ are identified as causes for the fix in $R_5$, are they both equally likely causes and thus to be considered of equal importance?
- Given that $R_3$ is identified as causing both $R_4$ and $R_5$, is it then considered a less likely cause for $R_5$, and thus to be considered of less importance?

In order to be able to reason about these questions, we need means to quantify the relationships between fixes and causes. We can establish that cause-fix relationships are many-to-many, that is a revision may be the cause for many subsequent revisions, and a revision may fix multiple previous revisions. A rather intuitive approach to quantifying the degree to which a revision can be considered as the cause for another revision is to think of a fix...
as “removing a weight”. In addition, there may be different types of “weights” based on different characteristics of the fixing revision, e.g. “fixing an issue”, “refactoring code”, etc., reflecting the different kinds of events of interest. In order to accommodate this, we extend the notion to “removing a weight related to a factor factor”. Thus, we speak of a fixing revision F as having removed weight (rw) with respect to factor factor where:

\[ rw(F, \text{factor}) = \begin{cases} 
1 & \text{if factor property holds for } F \\
0 & \text{if factor property does not hold for } F 
\end{cases} \quad (1) \]

Each of the causes C can be regarded as contributing to that weight, thus for each cause-fix relationship C causes F and for each factor factor, we define the notion of contributed weight (cw) of a causing revision C to a fixing revision F with regard to a factor factor as:

\[ cw(C, F, \text{factor}) = rw(F, \text{factor}) \left( \frac{1}{|\text{FIXES}|} \right) \quad (2) \]

For each fix F caused by a cause C, cause C is then said to accumulate a total weight (tw) with regard to factor factor, defined as:

\[ tw(C, \text{factor}) = \sum_{F \in C, \text{causes}} cw(C, F, \text{factor}) \quad (3) \]

For example, if the fix for the factor “fixes” in R5 is removing a weight rw(R5, fixes) = 1, and if there are two revisions R5 \text{FIXES} = \{\forall C \in \text{REVISIONS}[C \text{ causes } R_5] = \{R_3, R_4\} \} identified as causes for this fix, that are considered to be contributing equally to that weight, then each cause-fix relationship contributing a cw(C, R5, fixes) = 0.5. In this case, we speak of R3 and R4 as having a tw(R3, fixes) = tw(R4, fixes) = 0.5. Thus, at first glance it may seem that R3 and R4 can be considered equally important.

In order to reason about the second question, we need to contemplate the inverse relationship. If we consider R3 in the example, it causes both R4 and R5, thus R3 \text{CAUSES} = \{\forall F \in \text{REVISIONS}[R_3 \text{ causes } F] = \{R_4, R_5\}, whereas R4 only causes R5, i.e. R4 \text{CAUSES} = \{R_5\}. To take this into account we define the notion of average weight (aw) with regard to factor factor as:

\[ aw(C, \text{factor}) = \frac{tw(C, \text{factor})}{|C, \text{causes}|} \quad (4) \]

In the example above, this yields aw(R3, fixes) = 0.25 and aw(R4, fixes) = 0.5, respectively. Thus, we can state that while both R3 and R4 can be considered important as causes for the fix in R5 with respect to the
factor “fixes”, since $R_3$ is also a cause for $R_4$, it is less important than $R_4$ as it also caused a “neutral” change with respect to the factor “fixes” in addition to the fixing change. If we consider the “refactors” factor, we observe that the weights are distributed differently since it is $R_4$ where the weight related to that factor is removed ($rw(R_4,\text{refactors}) = 1$) and hence $R_3$ is the only identified cause contributing all the removed weight ($cw(R_3,\text{refactors}) = tw(R_3,\text{refactors}) = aw(R_3,\text{refactors}) = 1$). The corresponding weighting is also shown on Figure 1. The weight-related values are calculated for each factor for each node. Note, that while information about the causing revisions can be considered definitive, information about the fixing revisions is only partially known. Future revisions may still include fixes for existing revisions, thus altering their weights.

4 Layers and Strategies

In the examples discussed so far, only the project level of granularity was considered. In practice, a revision at the project level of granularity can be decomposed to revisions at the file and logical levels of granularity, where multiple related artifacts at these levels are changed together as part of a development activity. In this case, the challenge of transferring weights between the different levels arises. Furthermore, while a set of related artifacts may be changed within a causing revision, only a subset of these artifacts and possibly a set of additional artifacts may be changed within a corresponding fixing revision. Thus, the causes and fixes for a revision of an artifact at a finer level of granularity may be a subset of the causes and fixes for the containing artifact. Consequently, the weight distribution may vary across the different levels of granularity. This raises two fundamental challenges:

- Given a revision that is the cause for a fix, where the cause affects multiple artifacts at a finer level of granularity, are all of these artifacts contributing equally to the cause for the fix?
- Given a revision that is considered a fix, which affects multiple artifacts at a finer level of granularity, are all these artifacts equally important for the fix?

The naive approach is to simply copy the weights from the project level to the file level and to the logical level. As noted above, this can potentially result in a lot of noise since the sets of artifacts at a finer level granularity may vary between the causing and the fixing states at the project level. A more adequate approach is to construct distinct Cause-Fix Graph layers at the file and logical levels of granularity. This enables weights redistribution within the corresponding layers yielding more accurate weighting for each layer.

To address the second challenge we consider distributing the removed weights among the different layers while taking into account different characteristics, such as the type of the affected artifacts, the size of the affected artifacts, or the amount of changes in the affected artifacts in order to reflect the impact of such characteristics on the assigned weights. An example for a simpler strategy is the equal splitting of the removed weights among the artifacts at a finer level of granularity. A more sophisticated strategy takes into account the size of the modifications and favours artifacts with more substantial changes while distributing the removed weights, as these could be considered more costly or requiring more effort to perform. As a consequence, the resulting weights are also distributed differently to reflect the importance of the larger changes needed to perform the fix. The strategies can be further refined to distribute weights only among artifacts of a certain type, e.g. only code-related artifacts, where other artifacts can be ignored during the weight distribution.

5 Conclusion

In this article, we explored a weight-based approach for finding potential causes for events of interest in software repositories. The approach is based on a multi-layer cause-fix graph and can accommodate weighting across multiple factors independently. We outlined different strategies for the distribution of weights across the layers. The proposed weighting approach enables us to incorporate an arbitrary number of factors in order to accommodate different assessment tasks, focusing on different characteristics of contexts that can serve as indicators for potential causes of events of interest. Further characteristics can be used both for additional factors, and for additional weight distribution strategies.

References


Reverse Engineering Tool Requirements for Real Time Embedded Systems

Brice Govin 1,2
Arnaud Monegier du Sorbier 1
Nicolas Anquetil2, Anne Etien2,
Stéphane Ducasse2

1THALES AIR SYSTEMS,
Parc tertiaire SILIC,
3 Avenue Charles Lindberg 94628 Rungis Cedex
2RMod team, Inria Lille Nord Europe,
University Lille 1, CRIStAL, UMR 9189,
59650 Villeneuve d’Ascq, France

Abstract

For more than three decades, reverse engineering has been a major issue in industry wanting to capitalise on legacy systems. Lots of companies have developed reverse engineering tools in order to help developers in their work. However, those tools have been focusing on traditional information systems. Working on a time critical embedded system we found that the solutions available focus either on software behaviour structuring or on data extraction from the system. None of them seem to be clearly using both approaches in a complementary way. In this paper, based on our industrial experiment, we list the requirements that such a tool should fulfil. We also present a short overview of existing reverse engineering tools and their features.

1 Introduction

During their lifetime, systems face constant evolution, for example to fulfil new requirements. To keep useful they have to be maintained. However, maintenance requires a deep understanding of the systems that can be obtained for example using reverse engineering techniques. Some systems are data oriented and are mostly developed using object oriented paradigm. Other systems are behaviour oriented and are typically programmed using procedural paradigm. In this paper, we will focus on this second type of system and more precisely on real-time critical embedded ones.

These systems have specific constraints and characteristics. They are behaviour oriented in order to answer to time requirements, real time and hardware constraints. Data on the contrary were often not structured following domain abstractions, but rather to attend the needs of one functionality or the other. A domain abstraction of the data is nevertheless required to understand and document fully the system.

We tried traditional reverse engineering tools available on the market or in academia on several industrial real time critical embedded systems. However, they are mostly based on structure recovery in the object-oriented paradigm and do not fit in our cases. This paper aims to highlight features that a tool should provide to reverse engineer real time embedded systems.

The next section (§2) presents the features required by a reverse engineering tool dedicated to real time critical embedded systems. Section §3 reviews several existing reverse engineering tool in the light of these needed features. Section §4 draws some conclusions and presents future work.
2 Features for a reverse engineering tool

From a review of literature and based on our experience reverse engineering critical systems with existing tools we identify some features that a reverse engineering tool should provide to fit behavioural systems specificities.

Ducasse and Pollet provide a state of the art on software architecture reconstruction [DP09]. They propose to classify approaches according to five criteria: goal, process, input, technique, and output. Based on these criteria, we identify the following features.

**Process:** Ducasse and Pollet state that “because hybrid processes reconcile the conceptual and concrete architectures, they are frequently used to stop architectural erosion”. This process seems to fit better the case of reverse engineering a live system to keep it evolving and a tool should support that.

**Output:** It is important to keep in mind that despite the goal of extracting an abstract understanding of the system, the source code remains the main manipulated artefact. Thus, reverse engineering tools must provide *source code visualisation* like in a text editor.

According to [DP09], a lot of existing approaches provide graphical representation of system views. This is very common in software engineering at large. Thus, some *graphical visualisation* capability seems essential for any reverse engineering tool.

**Input:** The physical organisation of the system (that we call *concrete code structure*) corresponds to the organisation on the disk in term of files and folders as well as grammatical structure from the programming language. It often reveals structural information and a tool should be able to represent this information, map it to the source code and manipulate it.

In parallel with this, we consider a *concrete data structure*. It corresponds to the data structure extraction defined in [HHH+00]. Data is a very important aspect of any software system. Real time, embedded, systems will typically not offer data structures easily mappable to domain concepts because the data are typically implemented so as to facilitate the implementation of the behaviour.

Human expertise specifying a conceptual architecture is very helpful when it is available [DP09]. This logical structuring of the system can be completely different from the concrete code structuring. We call this the *abstract code structure* and again should be represented, mapped to the system and manipulable. Due to the type of system we target, this structure is typically behaviour oriented.

Similarly, we consider an *abstract data structure*. It is similar to the data structure conceptualisation specified in [HHH+00], and aims to define data structure in abstract terms, typically closer to domain concepts.

*Dependency analysis* aims to analyse dependencies either between data (e.g. sub-typing, reference) or between behavioural entities (e.g. invocation).

The five previous features focus on a single type of artefact either data or behaviour. However, data are manipulated by the behaviour e.g. through arguments in method invocations and as stated earlier, it is important to be able to consider data and behaviour in a complementary way. *Data/behaviour relationships provider* aims to highlight these relationships.

**Techniques:** Manipulating code either directly or through abstraction requires to be able to retrieve a given element by reference or by querying the representation. Therefore a *search* feature is needed.

**Output:** In the conformance output, [DP09] classify the ability to specify conformance rules and to validate them. These rules can be relative to either data, behaviour or both. A reverse engineering tool should therefore have a *rule creator and checker*.

**Goal:** In order to target the co-evolution goal, implementation and abstract representation that evolve at different speeds should be synchronised [DP09]. A tool should offer the possibility to *implementation an abstract structure* by modifying directly the source code according to changes made at higher abstraction level.

Finally it seems desirable that the tool be *open to user extension* and give its users the ability to personalise the tool for example by creating their own queries or visualisation and not only used the provided ones. Another example would be to allow the users to connect the reverse engineering tool to any other that they already use.

3 Reverse Engineering Tools

In this section, we look for the features previously defined on several tools available on the industrial market or in academy: Understand [Sci], Agility [Agi] , CodeCase [Cod] are industrial tools, Rigi [SWM97] is an academic tool and Moose [DLT00] is a meta-tool. Each of them provides code extractor for non object-oriented languages. However, the industrial tools focus on data and are relatively limited concerning behavioural and so also concerning data/behaviour relationship. The two academic tools are more generic, they can be adapted to focus on behaviour. Table 1 lists the features we found in each tool.

---

53
Table 1: Comparison of a few reverse engineering tools. (Agil=Agility, CC=CodeCase, Und=Understand) (“v” means that the tool provides the feature, “∼” that the tool can be adapted to provide the feature and a blank cell that the feature is not provided. “BU” stands for “bottom up”.)

<table>
<thead>
<tr>
<th>Features</th>
<th>Agil.</th>
<th>CC</th>
<th>Und.</th>
<th>Moose</th>
<th>Rigi</th>
</tr>
</thead>
<tbody>
<tr>
<td>Process choice</td>
<td>BU</td>
<td>BU</td>
<td>BU</td>
<td>BU</td>
<td>BU</td>
</tr>
<tr>
<td>Source code visualisation</td>
<td>v</td>
<td>v</td>
<td>v</td>
<td>∼</td>
<td>∼</td>
</tr>
<tr>
<td>Graphical visualisation</td>
<td>v</td>
<td>v</td>
<td>v</td>
<td>v</td>
<td>v</td>
</tr>
<tr>
<td>Concrete code structure</td>
<td>v</td>
<td>v</td>
<td>v</td>
<td>v</td>
<td>v</td>
</tr>
<tr>
<td>Concrete data structure</td>
<td>∼</td>
<td>∼</td>
<td>∼</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Abstract code structure</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Abstract data structure</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dependency analysis</td>
<td>v</td>
<td>v</td>
<td>v</td>
<td>v</td>
<td>v</td>
</tr>
<tr>
<td>Data/beh. relationships provider</td>
<td>v</td>
<td>v</td>
<td>v</td>
<td>v</td>
<td>v</td>
</tr>
<tr>
<td>Search</td>
<td>v</td>
<td>v</td>
<td>v</td>
<td>v</td>
<td>v</td>
</tr>
<tr>
<td>Rule creator and checker</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Implementation of abstract structure</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Open to user extension</td>
<td></td>
<td></td>
<td></td>
<td>v</td>
<td>v</td>
</tr>
</tbody>
</table>

4 Conclusion and future work

Based on literature review and our experience in reverse engineering real, industrial, time critical, embedded systems, we identified required features for a reverse engineering tool dedicated to this type of system. The defined features concern a system’s behaviour, data and the relationship between them. They also correspond to different abstract representation of the code and the way to either view it or query it. Each of these features should be mapped directly or indirectly to the code, since in the end, it remains the main artefact.

Existing industrial and academic tools are then evaluated according to these features. Even if each of these tools provide code extractor for non object oriented languages, none of them provides all features. Our future work is to develop a more complete reverse engineering tool dedicated to real time systems. For this purpose, we will base our work on existing techniques deployed in software architecture reconstruction or in data reverse engineering.

References


On the Evaluation of a DSL for Architectural Consistency Checking

Andrea Caracciolo
SCG, University of Bern, 3012 Bern, Switzerland
http://scg.unibe.ch

Abstract

Software architecture erodes over time and needs to be constantly monitored to be kept consistent with its original intended design. Consistency is rarely monitored using automated techniques. The cost associated to such an activity is typically not considered proportional to its benefits.

To improve this situation, we propose Dicto, a uniform DSL for specifying architectural invariants. This language is designed to reduce the cost of consistency checking by offering a framework in which existing validation tools can be matched to newly-defined language constructs.

In this paper we discuss how such a DSL can be qualitatively and qualitatively evaluated in practice.

1 Architecture erosion

Software architecture is the result of a design effort aimed at ensuring a certain set of quality attributes. The decisions deriving from such an effort are typically constraints on various aspects of an implementation. These may include invariants over structural (e.g., naming conventions, dependencies) or behavioral (e.g., communication) aspects of the system. Even though explicitly specified, these constraints are rarely checked automatically. In a previous study [Cara14], we show that only 40% of software architects formally specify and automatically test such constraints. This situation can be explained by analyzing the limitations associated with the tools currently available on the market. In fact, tools suffer from the following drawbacks:

- Scattered Functionality: Most tools are specialized in a narrow domain and are typically capable of evaluating only limited types of constraints.
- Specification Language Heterogeneity: Current tools are based on different specification languages that differ in both syntax and semantics.
- Specification Language Understandability: Tools often force the user to express constraints in a typically overly technical and verbose form.

To improve this situation we built Dicto, a DSL (Domain Specific Language) for the specification of architectural constraints.
2 Dictō

Dictō is a language that aims at supporting software architects in formalizing and testing prescriptive assertions on functional and non-functional aspects of a software system [Cara15]. Instead of dealing with multiple tool-specific formalisms, one can define several types of architectural constraints using one uniform, highly-readable, formal language (as shown in the example below).

```
Test = Package with name:"com.app.Test"
View = Package with name:"com.app.View"
Model = Package with name:"com.app.Model"
Controller = Package with name:"com.app.Controller"

Test, View can only depend on Model, Controller
Model cannot contain cycles
only Test can contain dead methods
```

The language is based on a plugin framework in which new language constructs can be defined along with the logic required to validate the concepts they are expressing. Developers can create a new rule template (e.g., Method must be executed in < Integer ms) by implementing a set of pre-defined data transformers for a given target tool. These transformers must be capable of (1) generating an input specification that is consistent with the user specified invariants; (2) interpreting the results produced by the tool. The advantages of this language are the following:

- Separation of concerns: conceptual design (specification of constraints) and technical effort (rule evaluation) are managed separately.
- Support for communication: a specification encoding valuable architectural knowledge should be accessible and readable by multiple parties, including stakeholders that do not have the skills necessary to operate the tool used to verify the expressed constraints.
- Reduce overall specification and validation effort: the time and effort involved in the process of writing and testing rules should be minimized as much as possible. To achieve this goal, we built a solution that offers reusable validation functionality, a uniform language syntax and a simple extension mechanism for the integration of new tools.

3 Evaluation

In this paper we discuss how Dictō, a DSL designed for encoding architectural constraints, can be exhaustively evaluated in an industrial context. Based on preliminary considerations, we would like to analyze the following properties by answering the following questions:

- **Impact on the product**: Does the solution improve code quality in any measurable way?
- **Impact on developers**: Does the solution increase architectural awareness?
- **Impact on process**: How is the solution integrated into the process? Are there any conflicts with pre-existing practices?
- **Specification Expressivity**: Does Dictō fit the specification needs of practitioners?
- **Specification Usability**: Are constraints easy to read and write? Are the results actionable?
- **Ease of adoption**: Is the effort required to support new requirements sustainable and cost-effective?

To answer these questions, we are currently running 3 different case studies with 3 different industrial partners. Our approach is to analyze their needs, encode their constraints using our DSL and analyze the effect Dictō has on the project. We plan to analyze the initial stages of the integration process, taking note on unsatisfiable expectations and similarities with pre-existent solutions. We will ask the study subjects to customize the rules
and adapt them to emerging requirements. We also plan to involve stakeholders with different background (e.g., developers, analysts, technical managers) and ask them to explain their understanding of the formalized rules. The general impact of the solution will also be measured by observing if reported violations are actually taken into consideration and fixed, and if this has some other beneficial effect on other quality indicators (e.g., coupling, test coverage, bug resolution time). Each constraint expressed in Dictö will be also compared with pre-existing coding guidelines, and implicitly known rules. We also aim at reporting on the potential reuse of developed tool adaptors (is a standard set of analyzers sufficient to express and test quality concerns across organizations?). By collaborating with multiple industrial partners, we hope to see how our solution is accepted in different contexts.

4 Conclusion

In this paper we discuss several best practices for the evaluation of a quality assessment tool in an industrial context. Our goal is to outline a concrete strategy for identifying the practical limitations of a prototypical solution. This is done by measuring the impact and applicability of a tool together with practitioners. The results of such an effort should help reaching a deeper understanding of the domain and generate new ideas for research.

5 Acknowledgements

We gratefully acknowledge the financial support of the Swiss National Science Foundation for the project “Agile Software Assessment” (SNSF project Np. 200020-144126/1, Jan 1, 2013 - Dec. 30, 2015).

References


A Search-based Approach for Generalizing and Refining Source Code Templates

Tim Molderez
Software Languages Lab
Vrije Universiteit Brussel, Belgium
tmoldere@vub.ac.be

Coen De Roover
Software Languages Lab
Vrije Universiteit Brussel, Belgium
cderoove@vub.ac.be

Abstract

Code templates are a convenient means to search and transform source code. However, such templates may still be difficult to specify, as they can produce too few or too many matches. To assist the users of our EKEKO/X program transformation tool, we have provided a suite of mutation operators for Java code templates and have designed a genetic search algorithm to recommend modifications to such templates.

Code templates are often used in source code search and transformation tools. Matching such a template results in a number of source code fragments of interest. Despite the low learning curve, as templates are specified in terms of concrete source code, it is not always easy to produce only the desired matches. A template may be too general and produces false positives, or too specific and produces false negatives.

The Ekeko/X [1] Eclipse plug-in we have developed allows users to search and transform Java programs by means of code templates. To assist our users in specifying these templates, we introduced a suite of several different kinds of template mutation operators: Some operators modify the template itself, e.g. by removing or inserting AST nodes in the code of a template, or by replacing nodes with wildcards or metavariables. Some operators will add directives to an AST node, which either add additional constraints to that node, or relax them. For example, an "invokes" directive attached to a method call indicates that it must call a given method body. Finally, there also are composite operators that can act on multiple nodes. For example, the "generalize-types" operator abstracts away the name of all occurrences of a particular type in a template.

Using this suite of operators, we have designed a single-objective genetic search algorithm [2] that recommends a series of modifications to an EKEKO/X template such that it will only match with a set of desired source code fragments. The fitness of a template is then gauged by considering how close its matches correspond to only the desired fragments. This algorithm is evaluated on the problem of assisting users in producing a template that matches similar source code fragments, e.g. instances of a bug, code clones or instances of a design pattern.

References


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


1Ekeko/X is available for download at https://github.com/cderoove/damp.ekeko.snippets
BibSLEIGH: Bibliography of
Software Language Engineering
in Generated Hypertext

Vadim Zaytsev
vadim@grammarware.net
Universiteit van Amsterdam, The Netherlands

Abstract

The body of research contributions is vast and full of papers. Projects like DBLP help us navigate through it and relate authors to papers and papers to venues. Relating papers to papers is already consider-
ably harder, and projects like Google Scholar do their best to battle all
the variations in citations. Relating papers to topics is an open problem
with some automated methods under development but mostly manual
contributions. Relating papers to concepts and especially concepts to
concepts is impossible without expert intervention and sometimes re-
quires years of research to accomplish in a convincing manner.

BibSLEIGH has started in 2014 as a personal project for pretty-
printing, normalising and eventually annotating bibliographic items. It
quickly started displaying the properties typical for model repositories,
both good and bad, linked to data extraction, consistency management,
synchronisation, analysis, etc. At SATToSE 2015 I would like to show
the project’s current state and discuss its potential in a broader scope.

Motivation

After noticing how much time that could have been spent writing papers, is wasted on reformatting the bibliogra-
phy, polishing it towards consistency, choosing the right abbreviations and making similar last minute decorative
adjustments, I have started a project called BibSLEIGH for collecting and managing my bibTeX database. The
main reasons were as follows:

- It is too late to start using advanced software like Mendeley: properly tagging several thousands papers and
  books is an impossibly massive investment.
- Obtaining individual bibTex files (downloading from publishers’ websites, copying from DBLP, searching
  in your old papers, crafting manually) takes time and provides no overview.
- BibTex providers commit to questionable conventions: DBLP separates proceedings from inproceedings;
  ACM abbreviates venues; Elsevier includes abstracts; IEEE adds keywords; etc.
- Venue names have too many variations, including obviously unacceptable ones with the status (proceedings,
  post-proceedings, pre-proceedings, revised papers, special issue, etc) at the end or lacking, but location and
dating information included in excruciating details.
- Different formats are needed in various circumstances: internal reports may have longer and more precise
  items; in-community links are contracted first to save space, etc.

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

Figure 1: The workflow of BibSLEIGH with concrete examples. On the left, in red, data sources: DBLP pages and its XML dump, bare bones noncontextualised BibTeX entries, Google Scholar-like hubs and publishers’ websites. In the centre, the main BibSLEIGH repository in the heart of all related activities. On the right, in green, possible uses of the database if we assume it exists and allows for easy access, querying, bundling, annotating and representing. The yellow (in monochrome, light frameless slanted text) are examples of mappers that import content from various sources, improve existing content based on heuristics or enhance it with additional data.

Solution

A lightweight solution was infeasible: most bibTeX providers resist crawling. At the same time, their data dumps are gigabyte long XML documents ruling out any free XPath processor. However, an iterative event-based parser was used to extract and sanitise interesting fragments — the initial case study covers 20 venues out of 6500+ available at DBLP with 25000+ papers. It was revealed that some of the irregularities are too sudden and impossible to detect, so the chosen solution is semi-automatic and produces a persistent intermediate representation which is stored separately and can be adjusted in any desirable way. A hypertext frontend is generated statically from this corpus of collected bibliographical data: it provides overviews, generates the actual bibTeX with desired conventions and links back to the editable corpus. Some examples of its activities are given below.

Incremental updates are possible with tools like dblp2json.py that slowly crawl a DBLP page of an event, collect all XML bibliographic files referenced from it, tries to figure out which one is the main entry (the issue of a journal or a volume of proceedings) and which ones are its elements, produces JSON entries accordingly and connects them to the rest of BibSLEIGH. Other such importers exist, and more can be written.

Automated transformation. When a venue which is already covered by the corpus, gains a new edition of a conference, it can be imported automatically. Subsequent sanitization might be necessary to polish some details, but the initial porting is a matter of a simple filtering and mapping of structured data. The main difference between this and the previous item is the necessity of having synchronising traits — i.e., matching previously imported content even if it evolved and import only new entries, — to make the project viable in the long run.

Engineering” (the year and the abbreviation is available from other fields, and the exact geographical location is irrelevant after the conference took place and tickers were reimbursed). This change itself is quick if done manually but its propagation from the proceedings entry to all the papers it contains, can be automated. There are many other situations when we need to intentionally disrupt and then automatically restore the consistency of BibSLEIGH.

**Heuristic-based transformation.** Page information is partially missing from some DBLP data: only the starting page of each paper is available. However, if we know the starting page of the next paper of the same volume, the ending page is easy to determine — and the heuristic is very realistic since modern CS/SE publications rarely contain advertisements between papers and rarely combine several papers on one page.

This section gives an overview of several activities and artefacts that are or can be involved in the workflow of BibSLEIGH. Many boxes on the figure are mere examples, the system can be extended considerably.

**Future work**

What makes BibSLEIGH become more than a glorified wrapper for DBLP is harvesting its domain specificity and community specificity. While keeping the automated, semi-automated and heuristic-based transformations as maintenance activities, we can continue ingraining the bibliographic entities and their groups with information relating them to one another, as well as to concepts, methods, frameworks, approaches and toolkits. Annotating them manually or automatically with information about topics can aid automated clustering and linking beyond traditional methods depending on the citation information. We see this as another step towards the construction of a body of knowledge for the domain of software language engineering (SLEBoK).

Software language engineering, besides being a subdomain of software engineering, is known to be a bridging area of research, where a fair share of activities is devoted to seeking similarities between technologies and technical spaces, and to developing techniques with wide and cross-space applicability. However, even within one space reaching a point of soundly relating concepts can take substantial time and effort — consider laying relations between attribute grammars and affix grammars [10] or between object algebras to attribute grammars [12]. Can BibSLEIGH facilitate this?

At the same time, providing interactive access to the curated annotated corpus of academic papers on programming language theory, compiler construction, metaprogramming, software evolution and analytics, refactoring and other related topics can serve as an entrance point into the research domain as well as the foundation for some metaresearch activities. Its relevance and usefulness for the SLE community will be vastly determined by the direction the project will take, but in general all these questions are currently counted among the open problems of SLE [2]. The main questions the project is currently facing is whether to proceed with it beyond \texttt{bibtex} pretty-printing and curation and if yes, then in which direction. Which features, if any, would be in demand within our (SATToSE+) community?

**Related work**

There exists related work in at least four different directions: model repositories; community support; annotated bibliographies; taxonomies. Model repositories such as FMI (Free Model Initiative) [13], ReMoDD (Repository for Model Driven Development) [8], CDO (Connected Data Objects)\(^1\), Atlantic Metamodel Zoo\(^2\), Grammar Zoo [16], GenMyModel\(^3\), etc on a quest of collecting models for various purposes. We can find quite a number of initiatives related to community management and facilitation: DBLP [11], Reengineering wiki [6], Researchr [14], Research 2.0 [1], SL(E)BOK, etc, including ones specifically aimed at community analysis, such as metaScience\(^4\). They usually combine requirements elicitation with experience reports with calls to arms. Annotated bibliographies also exist on different topics, like domain specific languages [5], reverse engineering [3], metrics [15], software (language) engineering\(^5\), parsing [9], some of them covering up to 1700 papers! Such bibliographies and are usually constructed by senior experts and can require years of research to accomplish in a convincing manner. Taxonomies go even one step further and tie key publications together with key concepts and relations between them: examples exist for taxonomies of domain specific aspect languages [7], reverse engineering [4], (un)parsing [17].

\(^1\)https://eclipse.org/cdo/
\(^2\)http://www.emn.fr/z-info/atlanmod/index.php/Zoos
\(^3\)https://repository.genmymodel.com
\(^4\)http://atlanmod.github.io/metaScience/
\(^5\)https://github.com/slebok/yabib
References


Resources

- http://bibtex.github.io — web front end
- http://github.com/slebok/bibsleigh — partially curated JSON data
- http://github.com/bibtex/bibsleigh — JSON refactorings and visualisations
Pretty printers: measured effects on productivity of teams of developers

Carlos U. Cirello F.
carlos.cirellofilho@student.uva.nl
Faculteit der Natuurwetenschappen, Wiskunde en Informatica
Universiteit van Amsterdam

Abstract

Production of software involves writing source code. This process takes several forms, among them the disposition of text so it becomes easy for human eyes to read and understand what is being done. Programmers diverge not only on what can be considered right code, but also what are the styles which are comfortable to work with. Aside the debate of coding style choices, there is also the impact of merge conflicts in versioning systems when changed source code possesses both cosmetic and content changes, therefore halting merging from long forked pieces of code.

1 Introduction

Source code is byproduct of programmer’s effort comprising two aspects: the merit and solution to the problem set at hand, but also personal opinions and biases. The formative years of a developer mold the vision of what is right and comfortable to work with - and when confronted with alternatives, the debate quickly derails into flame wars [Min 2008].

Aside the debate of coding style choices, there is also the impact of merge conflicts in versioning systems when changed source code possesses both cosmetic and content changes, therefore halting merging from long forked pieces of code.

Thus this research is meant to investigate the following questions:

1. May a code base written with different, or inconsistent, coding style formats lead to sooner, rather than later, merge conflicts? Does the outlook of merge conflicts changes when consistent style is applied?

2. What would happen if Source Code Management system had a pretty printer filtering its input and output?

This investigation is relevant for the domains of peopleware, by the aspect of quality assurance and productivity; revision control, while interested in keeping control of raw text is unable to solve cosmetic conflicts; and pretty printing: which acts as a normalizing force to source code as it does not communicate directly with revision control systems, except by its final product - the source code.

Copyright © by the paper’s authors. Copying permitted for private and academic purposes.
2 Anatomy of Pretty Printers

Pretty Printers are a class of program which takes valid source code as input, and produces as output valid and prettified code. The process which transform the input into output can be done in several ways, but it is a complicated effort:

Pretty-printing is complicated because the layout of a node cannot just be inferred from its form. [...] A pretty-printer must keep track of much contextual information. [Hughes 1995].

Therefore, some contextual representation of the source is necessary in order to avoid bad printing, or worse, wrong printing. From a mega-model comprising textual, structured and graphical representation of source [Zaytsev & Bagge 2014], it is possible to infer the transformation back and forth between Typed Tokens and String, and also AST and String.

Figure 1: Zaytsev & Bagge mega-model of parsing - note the possible transformations between AST, Typed Tokens and String, therefore offering more than one way to achieve pretty-printing.

2.1 Objects of research

3 Measuring effects on perceived productivity

Output of pretty printers affects perceived productivity by automating the chores of applying a coding style. A single style within a project is said to ensure consistency when desired [Zend 2014], to spread familiarity [Straker 1991], restrict the use of language to some subset [Shields 2014] and prevent a set of mistakes [Shields 2014].

Another important effect is that the automation itself can be used as an advantage to productivity by pegging the pretty printer onto one’s editor [Cirello phpfmt 2014] [Walters et al. 2014]. As of December 2014, at Tiobe Index [TIOBE Nov 2014] among top 10 languages, all of them have pretty printers available [Arceneaux et al. 2008] [Jindent] [Hattori et al. 2010], which shows a community effort in offering them.

3.1 Merge Conflict Problem

When two or more developers try to modify the same file at the same point, a conflict arises. Achieving behavior change, however, is not the only topic in the developer’s agenda — they have their own vision about how source code should look like aesthetically; so even if they make the same behavior change, they piggyback a cosmetic modification.
One tactic is solving this aesthetic problem is to establish a coding format standard and enforce through a code sniffer, however it has some problems: (a) code sniffers may only act after the file is saved, as they need the whole content to produce a full report of the mistakes; (b) they are usually configured to act post-hoc, meaning they cannot prevent the pollution of versioning systems with codes which violate certain style standard.

On the other hand, pretty printers can act on source code in several levels, and they can act as a filter to the file storing process because they transform, as a stream, the input into output, keeping control of the state necessary to produce the changes, pretty much like a compiler.

### 3.2 Investigation and mitigation of the Merge Conflict Problem

Implement a simulation of working environment, in which a program (from now on named as “Chaos Maker”) introduces random syntactically valid changes into sample piece of code (PHP) and commits to a centralized repository. Two instances of this program would be running, and creating opportunities for change conflict. In another experiment, this would be replayed, but both instances would pass the code through an automated code style formatter, and then the conflict metric of both experiments would be compared.

This simulation will be configured to accommodate the following scenarios:

1. Small code bases, with developers changing all possible files randomly;
2. Large code bases, with developers changing all possible files randomly;
3. Large code bases, with developers changing a subset of files (simulating concerted efforts into a single module of a bigger system);

The gathered metrics are: (a) number of merge conflicts caused by cosmetic changes, meaning when all non-whitespace tokens, when compared are the same; (b) mean distance between first commit and first merge conflict.

Another set of simulation shall take place through the application of a filter before all commits activities, in order to reproduce the concept of a semantic-aware SCM [PlasticSCM 2014] and replay the scenarios above.

### 3.3 Expected results

If the literature is correct, it is expected following null hypotheses: (a) tendency towards nil of merge conflicts stemmed from cosmetic changes; or (b) the decrease of overall number of conflicts in the lifetime of a versioned source code repository.

Also the creation of a language-aware source code management tool or front-end plugin for some existing one, namely *git*.

### References


Interactive User-Oriented Views for Better Understanding Software Systems

Truong Ho-Quang, Michel R.V. Chaudron
Department of Computer Science and Engineering
Chalmers University of Technology and Gothenburg University
Gothenburg, Sweden
{truongh, chaudron}@chalmers.se

Abstract

Understanding about software artefacts is a crucial task for people who want to join in any software development process. However, because of the large amount of detailed and scattered information in software artefacts, understanding them is usually very time-consuming and vulnerable to human errors and subjectivities. A system that aids practitioners to investigate understanding about software artefacts could reduce the vulnerabilities and speed up software development/maintenance process. Our research focuses on building a comprehensive view of software system in order for developers to achieve the two goals: (i) to save the time spending on searching and navigating on source code; AND (ii) to gain better understanding about software artefacts regarding to domain-specific tasks. To achieve these goals, we propose an empirical approach which has been conducted and received preliminary results.

1 Introduction

Software artefacts are created, maintained and evolved as part of a software development project. Understanding about software artefacts is a crucial task for every person who wants to join in any phase of software development life cycle [26]. However, because of the large amount of detailed and scattered information in software artefacts, understanding them is usually very time-consuming and vulnerable to human errors and subjectivities [9][20]. The task becomes even more difficult when it comes to large-size software projects, which contain a huge amount of code, designs and documentation.

Recently, a significant number of research and tools has been conducted in order to investigate better understanding of software artefacts. It can be listed as reverse engineering [3], feature location [8], document summarization [ref], etc. However, there is lack of automatic approaches were proposed. With regards to those that can automatically perform the task, the accuracy is not high [12]. Soh et.al, with an observation on 2408 developers interaction logs, has pointed out that 62% of files explored during the implementation of a task are not significantly relevant to the final implementation of the task [20].

In addition, the approaches are usually applied on a single software artefact (e.g. source code, revision history), resulting in a single view(navigation or search results). This fact, at some points, limits developers ability to obtain the overview of the whole (OR a part of) system, which is an essential part of understanding, with
regards to the development/maintenance task. Thus, creating of a comprehensible view which can automatically navigate and generate suitable views on different software artefacts would be very beneficial.

To this end, our research has been focusing on the visualisation and the generation of high-level design and architectural views from source code and design documentations. The research could unlimitedly be extended to different views on a various range of software artefacts (such as requirements, use-cases, test-cases, revision logs [6]).

![Figure 1: A comprehensible view for task "Maintain a feature"](image)

Figure 1 shows a prototype design of such the view. Given a maintenance task, the sub-views show the task-related parts on different software artefacts. Sub-view Architecture Overview shows an overview of the system with highlights on the task-related components. Sub-view Editor locates to the relevant part of source code. Sub-view Chat shows the list of the responsible developers and the historical chat regarding the observed source code. Sub-views are linkable between themselves and automatically or manually updated. We take (industrial and educational) developers/programmers as the main audience of our research. The following example reveals how the developers perform the understanding task using the view. We consider it as the motivation in our research.

**Motivation example.** Developer X has to conduct a task: Maintain Feature A. Software artefacts are stored in projects database. X starts by logging into his work space (e.g. an IDE), then performs searching for the Feature A by key words. Sub-view Editor will automatically address the related parts of source code which could potentially be changed during the maintenance work. Sub-views Architecture Overview and Sequence Diagram will provide the developer with an overview about the code structure and probably a suggestion about which parts could be subsequently changed. The developer can ask for recommendations from responsible persons through Chat space.

Using such the view could allow the developer to better investigate understanding about the task and the system, and to reduce the implementation and maintenance time.

2 Problem Statement and Research Questions

2.1 Problem Statement

**Lack of empirical research on developers cognitive task during software maintenance phase.** Despite the large body of work on software maintenance [26][20][2], there are very few studies that empirically investigated how developers achieve the understanding of software artefacts during software development/maintenance activities. Ko et.al [13][14] has revealed a number of issues that cause developers more time on navigation between source files. The authors have suggested ideas for tools that help developers seek, relate, and collect information in a more effective and explicit manner. On the other hand, it seems that there is a huge gap between state-of-the-art research and practice in software comprehension. An observational study by Roehm et.al shows that no one in the 28 professional developers (from seven software companies) observes any use of state-of-the-art comprehensive tools [18].

**Hard to collect relevant task-based information effectively and automatically.** For most tasks, developers begin by searching then navigating by search results. However, traditional searching methods seem not very effective. A. J. Ko et.al have revealed that an average of 88 percent (±11) of developers searches led to nothing of later use in the task. Those failed searches were at least partially responsible for approximately 36 percent of their time spent on inspecting irrelevant code [14]. Recently, a number of task-specific searching methods (such as features location, program slicing, UML slicing, etc.) has been introduced. However, they are
not easy to apply and sensitive to inputs quality [8]. Thus, developing an easy-to-use solution could improve searching and navigating efficiencies.

**Lack of visualisation of relevant information in understandable manners.** Apart from searching and navigation, visualisation of software artefacts is widely used in the areas of software maintenance, reverse engineering, and re-engineering, where typically large amounts of complex data need to be understood and a high degree of interaction between software engineers and automatic analyses is required. Over the past few years, software visualisation has greatly evolved. However, despite the fact that software visualisation tools have a great potential, when it comes to contextual information, finding a suitable solution is not an easy work [21][10] (e.g. UML design layouting, personalised view, etc.).

2.2 Research Questions

Our research focuses on the visualisation and the generation of high-level design and architectural views from source code and design documentations. In order to come up with a systematic answer for the question, it’s necessary to find out what is practitioner’s mind when performing the understanding task. Thus, we would think about the following research questions:

*RQ1.* What are practitioners needs in order to understand a part of system with regards to a specific task?

*RQ2.* How to generate and present the information by an effective way?

3 Research Approaches

In order to investigate the two research questions, we conduct two research activities as shown in the Figure 2. In Research Activity 1, we use both qualitative and quantitative approaches to learn practitioners needs and strategies during the understanding phase. In particular, by conducting interviews with industrial and academic practitioners, we could achieve better understanding on what is their cognitive thinking and possibly the strategy that was used to understand the system. By logging practitioners activities and analysing the logging file, we could statistically investigate their unconscious behaviors and the difficulties performing the understanding task. This approach is discussed in detail in subsection A.

Research Activity 2 aims at answering RQ2 with a focus on generating high-level abstraction of design and architectural views from source code. We have been studying sense-making and software architectural visualisation. The approaches are discussed in subsection B.

Activity 1 and Activity 2 are concurrently performed. On one hand, outcomes of Activity 1 can be considered as requirements for Activity 2. On the other hand, research ideas and the views that are generated from Activity 2 will be introduced to practitioners. Validation will be made during the iterations of the two activities.

3.1 An exploratory study of practitioners

3.1.1 Conduct interviews

Developers are often not up-to-date with state-of-the-art comprehension tools. On the other hand, academia has a limited knowledge about industrial practitioners. For example, when it comes to questions like: *How could we understand a (part of a) software system?* Referring to software design seems to be an obvious answer. However, none of the observed research has mentioned the use of architecture design as part of the understanding process.

*Therefore, semi-structured interviews* will be used to get a better understanding of software practitioners. We consider both academic and industrial developers as targeted interviewees. We split them into groups by several ways: level of software comprehension expertise; familiar with a specific software system/software
maintenance task. Shedding some lights in the differences between groups in understanding software systems could be beneficial for us in order to generate suitable views for each group. We take our colleagues and Software Engineering students at the University of Gothenburg and Chalmers University of Technology as academic candidates. We have been inviting a number of local companies (such as Volvo Cars, Ericsson) and out-of-border companies (which locate in Vietnam, The Netherland) to involve in this research.

3.1.2 Process Mining

In the quest for knowledge about strategies and struggles that practitioners have found during the understanding phase, process mining is possible research tool. Process mining techniques make use of historical data to graphically represent and analyse a particular process [23].

Blikstein reported of the use of a logging module for programming tasks [1] and identified student strategies that could help lecturers identify student problems in an early stage of the task. Ko et al. conducted a study in which they used the combination of a logging file and a visual interpretation tool to analyse the behaviour of software developers during a maintenance task [14]. They successfully identified different strategies the developers used. Claes, Pingerra et.al [4][17] logged students events during business process modeling sessions. They used visual analysis [24] and found different styles and related them to model quality.

By using a process mining approach, we have conducted an exploratory study on students strategies performing software modeling tasks [5]. We found out that students use different strategies for solving the tasks. We categorised these strategies into four main strategies: Depthless, Depth First, Breadth First and Ad-Hoc. From our results Depth First indicates to support better layout and richness (detail). We wanted to examine our insights by conducting this experiment on a bigger sample size of students, and possibly on industrial side.

3.2 Data generation and development of the views

3.2.1 Sense-making on source code

Sense-making, as described by Weick [7], literally means making sense of events. According A. von Mayrhauser, sensemaking is a term used to refer to humans capability to actively comprehend the significance of ambiguous events and data [25]. To our point of view, sense-making is considered as a process where software artefacts are manipulated and presented in a higher level of abstraction. With the focus on high-level design and architectural concepts from source code, we take software reverse engineering (RE) and natural language processing (NLP) as the main drivers for the sense-making process.

Reverse Engineering. Reverse engineering aims to analyse the source code of a system and create design representations of the system [3]. Open source and commercial tools have been developed to generate software design from source code. However, the reverse engineered presentation often contains too much details. When a RE class diagram becomes too large, it provides little benefit towards program. We have been working on possible solutions to present the RE diagrams in a more informative way.

We take the prior research done by Osman et.al as inspiration. The authors have proposed a supervised machine learning approach to condense RE class diagrams into another class diagram that is close to forward design diagram [15][16]. The authors compute values of a number of design metrics from source code and use those to predict classes as important or not. The condensed diagram is then constructed from the reverse-engineered diagram by keeping the important classes and eliminating unimportant ones. Thung et.al have extended Osmans work by using networks metrics as predictors [22]. This work could be extended by considering more predictor features, i.e. dynamic metrics (from execution traces), text mining metrics (from use cases, requirements).

Natural Language Processing. NLP can be considered as a process of extracting information from human or natural language inputs. By adapting NLP to source code analysis, one could be able to extract semantically-related parts of source code, which in the end results in the reducing of maintenance cost. Shepherd et.al has introduced a Find-Concept search process which makes use of NLP analysis that captures the relations between actions (verbs) and the objects (nouns) that these actions act upon [19]. Hill et.al propose a technique and a tool to score method relevance with respect to natural language descriptions of a specific maintenance task [11]. Starting with a seed method and a natural language description of the bug to be fixed or feature to be added, the tool automatically generate and show a reduced version of the call-graph of the method by pruning irrelevant structure edges from consideration. We have been working on automatic recognition of class roles (such as UI, security, persistence, etc.) by using text-analysis on source code. The result could then be displayed in a role-based view of reverse engineering design.
3.2.2 Data visualisation

Visualisation depends on target audience and its information needs which are not available at the first phases of the research. Therefore, visualisation is not our main focus at current time. So far, we have been working on the two main directions: 1) With regards to the visualisation of RE diagrams: We are considering different visualisation strategies for class’s roles; 2) Regarding presentation of activity logging data: We have developed the tool LogViz which is capable of showing multiple logging files and filtering the logging by activities and architectural elements [5]. In the time to come, we tend to contact with companies for validating our approaches.

References


Collaboration Networks in Software Development: Perspectives from Applying different Granularity Levels using Social Network Analysis - Research in Progress

Miguel Ángel Fernández, Gregorio Robles
GSyC/LibreSoft
Universidad Rey Juan Carlos
{ma.fernandezsa@alumnos., gregorio.robles@}urjc.es

Jesús M. González Barahona
Bitergia
jgb@bitergia.com

Abstract
This paper shows research in progress in the analysis of collaboration networks found in software development projects. Traditionally, collaboration networks are obtained by analyzing collaboration in the same file or module/directory; when two developers perform modifications on the same entity during a given time period it is assumed that they are at least implicitly collaborating. In our research, we want to study how the granularity of the software artifact affects the research output of collaboration graphs. In this regard, we obtain traditional graphs based on collaboration in files and augment it with information of collaboration at the function/method level. In the future we want to include developer affiliation information to perform a collaboration analysis at the company level.

1 Introduction/Motivation
The development of large software systems is a collaborative task where many developers, sometimes up to thousands of them, are involved. In such scenario, software engineering research has been long looking after understanding how these collaborations arise, and how they evolve over time [MM07, Sin10, SLL10, HZ09].

In order to identify collaboration, many scholars have used techniques such as social network analysis, where two developers (nodes) are connected if they have collaborated together [MFT02, LFRGB+04]. In most social network studies the resulting network is based on file-based or module-based data of interactions; if there has been a collaboration between two developers in a file or a module, these developers are connected.

Our research is concerned with the fact that the resulting network graph depends heavily on the granularity level that is selected [HWC12]. When there are tens of files in a module/directory or thousands of lines in a file, did collaboration really exist?

Therefore, in addition to the existing collaboration graphs, we have been working to obtain a new one that takes collaboration at the function/method level into account. In this type of graph, two developers collaborate if they have modified the same function in a given time period. We think that, although there might be exceptions with large functions/methods, this provides a new, still unknown level of granularity in the analysis that can help to obtain a better global picture.
2 Methodology
Our methodology studies registered changes made to a given repository tracked by a versioning system (in our case git\(^1\)). From that repository, it extracts the log for a specified date range. Using that data, the program iterates for each commit, performs a checkout and uses ctags with each file to identify function/method information in each of those files.

Next, matches between the commits information and ctags data (still for each checkout) are searched for, to identify those methods that have changed. By now, changes in methods are only tracked if the method has not changed its name. Future versions of our tool will include heuristics to analyze as well those functions whose name change.

While changes to functions/methods are extracted, the developers who have performed these changes are attached to the change. As by now, we do not apply developer merging algorithms, so that a developer with different aliases will only appear once, but we plan to do it in the future. This data is aggregated and offered in two CVS-formatted files, one for collaborations occurring in the same file and another one with collaborations in the same function/method. This output can be used by traditional programs to obtain a social network graph, such as Gephi, or to calculate social network measures and properties.

3 Case study
We have used the program to study the evolution of the GNOME-text editor gedit\(^2\). The considered date range for this study goes from the very beginning of the project (at least, from the moment a first commit has been found in the log), which is April 15, 1998 until April 15, 2015. Within this time range, we have chosen time-lapses of six months so we can handle and understand better the resulting data and its evolution over time. The chosen time period is consistent with the release period of GNOME (at least from 2005 onwards).

In this paper we show the results for two selected timeframes: from January 1, 2001 to May 31, 2001 and from June 1, 2014 to December 31, 2014. For both ranges we offer a graphic representation of collaboration networks between developers. Each node represents a developer, and edges represents interactions between them. We have two different graphs for each date range: an in-file collaboration network (developers who have modified same file) and an in-method collaboration network (developers who have modified same method).

Figure 1 shows the different graphs (in-file and in-method data) for the first half of 2001. As expected the number of collaborations (only those developers who collaborate are shown in the graph) is larger in the in-file network than in the in-method one.

Figure 2 shows graphs (in-file and in-method data) for the second half of 2014. We see that the number of contributors to gedit has grown, and as such its network size and number of collaborations.

\(^1\)The tool is available on-line as free software in a GitHub repository: https://github.com/LibreSoftTeam/R-SNA
\(^2\)The repository of gedit used in this analysis is https://github.com/GNOME/gedit
We have computed a numerical representation, the betweenness centrality of a node, that reflects the amount of control that this node exerts over the interactions of other nodes in the network. The values of betweenness given in Tables 1 and Table 2 are normalized, and only nodes that have at least one non-zero value are shown.

As it can be seen the values are very different when considering the two types of collaboration networks, suggesting that developers that with traditional techniques (i.e., collaboration in files) may not be that central if collaboration at the function/method level is considered. This can be observed by ranking developers by their betweenness value for both networks; those who are highly ranked at the file level have not to be the ones that have the higher values at the function level.

Table 1: Normalized betweenness centrality for the first half of 2001.

<table>
<thead>
<tr>
<th>Node</th>
<th>Files</th>
<th>Methods/Functions</th>
<th>Rank Files</th>
<th>Rank Methods/Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gediminas Paulauskas</td>
<td>0.08305</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Chema Celorio</td>
<td>0.19019</td>
<td>0.02140</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>Paolo Maggi</td>
<td>0.12273</td>
<td>0.02140</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>Pablo Saratxaga</td>
<td>0.10483</td>
<td>0.23965</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>Jacob Leach</td>
<td>0.00159</td>
<td>0.00737</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>Jason Leach</td>
<td>0.00159</td>
<td>0.06807</td>
<td>5</td>
<td>2</td>
</tr>
</tbody>
</table>

4 Future work

As future work, we plan to improve our tool with state-of-the-art functionality by (1) including algorithms to track function name changes [GZ05], and (2) integrating algorithms to merge developer aliases [KVSvdB12].

Once this is done we want to perform several analysis to compare the collaboration graphs obtained by analyzing at the file vs. the function/method level, and to see the implications of doing the analysis at a finer level of granularity. Therefore we want to reproduce some of the studies done in the past that have been done at the file level.

In addition, we plan to augment the analysis with developer affiliation information [GBICMR13]. This will allow to gain further understanding of emerging communities such as OpenStack or WebKit, that are mainly composed of industrial partners.

Acknowledgments

This work has been funded in part by the Spanish Gov. under SobreSale (TIN2011-28110) and by the Comunidad de Madrid under “eMadrid - Investigación y Desarrollo de tecnologías para el e-learning en la Comunidad de Madrid” (S2013/ICE-2715).
Table 2: Normalized betweenness centrality for second half of 2014

<table>
<thead>
<tr>
<th>Node</th>
<th>Files</th>
<th>Methods/Functions</th>
<th>Rank Files</th>
<th>Rank Methods/Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sebastien Wilmet</td>
<td>0.14896</td>
<td>0.00842</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>Sebastien Lafargue</td>
<td>0.05861</td>
<td>0.03879</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Paolo Borelli</td>
<td>0.04775</td>
<td>0.01284</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>Robert Roth</td>
<td>0.03033</td>
<td>0.00410</td>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td>Jesse van den Kieboom</td>
<td>0.02957</td>
<td>0.01990</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>Piotr Drg</td>
<td>0.02797</td>
<td>0</td>
<td>6</td>
<td>-</td>
</tr>
<tr>
<td>Ignacio Casal Quinteiro</td>
<td>0.02728</td>
<td>0.00134</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>Boris Egorov</td>
<td>0.00645</td>
<td>0</td>
<td>8</td>
<td>-</td>
</tr>
<tr>
<td>Matthias Clasen</td>
<td>0.00832</td>
<td>0.02218</td>
<td>9</td>
<td>2</td>
</tr>
<tr>
<td>Alexandre Franke</td>
<td>0.00061</td>
<td>0</td>
<td>10</td>
<td>-</td>
</tr>
<tr>
<td>Igor Gnatenkov</td>
<td>0.00027</td>
<td>0</td>
<td>11</td>
<td>-</td>
</tr>
</tbody>
</table>

References


Controlled Experiment to Assess a Test-Coverage Visualization: Lesson Learnt
– Experience Report, work in progress –

Alexandre Bergel¹, Vanessa Peña-Araya¹, Tobias Kuhn²

¹PLEIAD Lab, Department of Computer Science (DCC), University of Chile
²Department of Humanities, Social and Political Sciences, ETH Zurich

http://bergel.eu
vpena@dcc.uchile.cl
kuhntobias@gmail.com

June 29, 2015

Abstract

Evaluating a software visualization is a difficult and error-prone activity. In this short paper, we report our experience when conducting a controlled experiment to evaluate test blueprint, a visualization to assess the test coverage. Our experiment went through two iterations. The first iteration was unfortunately inconclusive, due to some decisions we took that we are now considering as mistakes. After revising our experiment, we obtained exploitable results, which are also matching our intuition.

1 Visually Assessing Test Coverage

Motivation. Test coverage is about assessing the relevance of unit tests against the tested application. It is widely acknowledged that software with a good test coverage is more robust against unanticipated execution, thus lowering the maintenance cost. However, ensuring good quality coverage is challenging, especially since most of the available test coverage tools do not discriminate between software components that require strong coverage from the components that require less attention from the unit tests.

Visualization. Test blueprint is an innovative visualization for test coverage [1]. It employs an effective and intuitive graphical representation to visually assess the quality of the coverage. A combination of appropriate metrics and relations...
visually shape methods and classes, which indicates to the programmer whether more effort on testing is required.

Encapsulating boxes represents classes (e.g., C1, C2). Inheritance is indicated with an edge between classes. Subclasses are below their superclass (C1 is the superclass of C2). Inner boxes represent methods. C3 defines six methods. Each method is represented as a small box, visually defined with five dimensions:

- **height** is the cyclomatic complexity of the method [2]. As the method may take different paths at execution time, the higher the box will be.

- **width** is the number of different methods that call the method when running the tests. A wide method (w) means the method has been executed by many different methods. A thin method means the method has been executed a few times.
• gray intensity reflects the number of times the method has been executed. A dark method has been executed many times. A light-toned method has been executed a few times.

• a red border color (light gray on a B&W printout) means the method has not been executed. A blue border indicates abstract methods. A green border indicates that the method is a test method, defined in a unit test. Note that a unit test may contain methods that are not test methods; utility methods for example.

• the call-flow on the self variable is indicated with edges between methods. This happens if the body of a method method1 contains the expression self method2, meaning that the message method2 is sent to self. Note that we are focusing on the call-flow instead of the control-flow. The call-flow is scoped to the class. Call-flow is statically determined from the abstract syntax tree of the method.

**Evaluating test blueprint.** Measuring the impact of the test blueprint on developers has been the topic of a long effort. The remaining of the paper describes the two attempts we carried out. The two hypotheses we are interested in are:

H1 - *Test blueprint helps identifying the method to test in order to maximize the coverage increase.*

H2 - *Test blueprint helps assessing the difficulty to test a class.*

### 2 First Attempt

**Controlled experiment design.** As a first attempt to evaluate test blueprint, we conducted a controlled experimented, designed as follows:

• As a base line, we took the visualization produced by EclEmma\(^1\), which we consider as a standard test coverage tool for Java. Figure 2 illustrates this visualization.

• Instead of comparing the tools themselves, we solely focused on the visualization. Evaluating the tools would require more effort, especially since EclEmma has been developed and maintained for a long time period. In addition, evaluating the tools would introduce biases related to the programming language (Java vs Pharo) and to the IDE (Eclipse vs the Pharo IDE). Test blueprint is implemented in the Pharo programming language\(^2\).

\(^1\)http://www.eclemma.org

\(^2\)http://pharo.org
• The Pharo open source community has multiple times expressed the need of a robust test coverage tool. Since we strongly felt test blueprint was aiming at addressing this need, we decided to directly survey the community. For that purpose, we designated a small web questionnaire for the community members to fill in.

• Participants were asked a number of questions on two sets of classes, one shown with test blueprint and the other with EclEmma. Questions were divided in two categories: (i) classes characterization regarding their easiness to test and (ii) methods having the highest potential to increase the coverage. These two questions are directly related to the two hypotheses. Category (i) contained 8 questions and category (ii) has 6 questions.

Results. The experiment has been carried out with care. Unfortunately, the experiment was not concluding. In particular, no tendencies could be drawn and the hypotheses have been left unverified. We believe this is the result of a number of suboptimal decisions we took:

• Classes that were given to the participants to assess were small. This was made on purpose to avoid measurement bias that would stem from large windows (e.g., participants would need to scroll through the visualization). It is known that visualization helps facing scalability (especially when compared with textual listing). However we did not exploit this.

• Too many questions where asked to the participants. These questions were also imprecise and not directly linked to the hypothesis we wished to evaluate.

• Participants have a natural tendency to be resistant to proposals to improve their environment. Although we were not able to measure it, we had a clear impression that the participants’ answers were indicating that they wish no changes.

3 Second Attempt

As a second attempt, we improved the experiment design as follow:

• 12 participants have been selected from the University of Chile and from a local company that is known to use unit tests in their production.

• participations were surveyed on paper and not via a website.

• Only two questions were asked to each participant.

• We have removed some variables in our design. In particular we do not make a distinction between small and large classes and between the inner structure of classes. The reason for this is that 12 participants was not enough to have a representative data set.
This revised experiment provided better results. The controlled experiment indicates that the hypothesis H1 is not verified, despite a better average score of test blueprint against EclEmma. The hypothesis H2 is verified: test blueprint significantly outperforms coverage report listing to indicate the difficulty to test a class.

4 Conclusion

This short paper describes the two iterations we have carried out on evaluating a test coverage visualization. We hope the experience we are reporting will be beneficial to other researchers.

References


![Figure 2: EclEmma’s test coverage output](image-url)
Predicting Software Quality through Network Analysis

Giulio Concas
concas@diee.unica.it
Michele Marchesi
michele@diee.unica.it
Cristina Monni
cristina.monni@diee.unica.it
Matteo Orrú
matteo.orru@diee.unica.it
Roberto Tonelli
roberto.tonelli@dsf.unica.it

Department of Electrical and Electronic Engineering (DIEE)
University of Cagliari
Piazza D’Armi, 09123 Cagliari (Italy)

Abstract

We used a complex network approach to study the evolution of a large software system, Eclipse, with the aim of statistically characterize software defectiveness along the time. We studied the software networks associated to several releases of the system, focusing our attention specifically on their community structure, modularity and clustering coefficient. We found that the maximum average defect density is related to two different metrics: the number of detected communities inside a software network and the clustering coefficient. These two relationships both follow a power-law distribution which leads to a linear correlation between clustering coefficient and number of communities. These results can be useful to make predictions about the evolution of software systems, especially with respect to their defectiveness.

1 Introduction

Modern software systems are large and complex products, built according to a modular structure, where modules (like classes in Object Oriented systems) are connected with each other to enable software reuse, encapsulation, information hiding, maintainability and so on. Software modularization is acknowledged as a good programming practice [Par72, BC99, SM96] and a certain emphasis is put on the prescription that design software with low coupling and high cohesion would increase its quality [CK94]. In this work we present a study on the relationships between software systems quality and their modular structure. Depending on the fact that software systems are inherently complex, the best model to represent them is by retrieving their associated networks and the related topological properties [Mye03, ŠB11, WKD07, BB0]. In a software network nodes are associated to software modules (e.g. classes) and edges are associated to connection between software modules (e.g. inheritance, collaboration relationships). We investigated the software modular structure - and its impact in software quality - by studying specific network properties: community structure, modularity and clustering coefficient. A community inside a network is a subnetwork of densely connected nodes when compared to nodes outside the community [GN01]. Modularity is a function that measures how marked is a community structure (namely the way the nodes are arranged in communities) [NG04]. The clustering coefficient represents a measure of how much nodes are connected inside a network [New03].

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

82
We studied several releases of a large software system, Eclipse, performing a longitudinal analysis on the relationship between community structure, clustering coefficient and software quality. Our aim is to figure out if the studied metrics can be used to predict software quality of future releases.

This paper is organized as follows. In Section 2 we illustrate the methodology. In Section 3, we present and discuss some of our results and finally, in Section 4, we report our conclusions.

2 Methodology

In this work we aim at analysing the structure of a software system using its associated software network. In order to build the associated software network we parsed software’s source code, retrieved from the corresponding Software Control Managers (SCM). During this procedure, we associate network nodes to classes and network edges to the various relationships between classes (inheritance, composition, etc.). We consider the number of defects (bugs) as a main indicator of software quality. To exploit this we collected data about the bugs of a software system by mining its Bug Tracking Systems (BTS). In order to associate each bug to the corresponding classes we mined the commits on the software SCM to figure out which classes a bug fix intervention is related to. At the end of this process we obtained a network where to each node is associated the number of bugs of the corresponding class. We collected the source code and analysed 5 releases of Eclipse, whose main feature are presented in Table 1.

<table>
<thead>
<tr>
<th>Release</th>
<th>Eclipse 2.1</th>
<th>Eclipse 3.0</th>
<th>Eclipse 3.1</th>
<th>Eclipse 3.2</th>
<th>Eclipse 3.3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size</td>
<td>8257</td>
<td>11406</td>
<td>13413</td>
<td>16013</td>
<td>17517</td>
</tr>
<tr>
<td>Sub-Projects n.</td>
<td>49</td>
<td>66</td>
<td>70</td>
<td>86</td>
<td>104</td>
</tr>
<tr>
<td>N. of defects</td>
<td>47788</td>
<td>59804</td>
<td>69900</td>
<td>80149</td>
<td>95337</td>
</tr>
</tbody>
</table>

Table 1: Main features of the analysed releases of Eclipse (EC): size (number of classes), number of sub-projects (sub-networks), and total number of defects.

Each release is structured in almost independent sub-projects. The total number of sub-projects analysed amounts at 375, with more than 60000 nodes (classes) and more than 350000 defects.

We detected the associated community structure using the algorithm devised by Clauset et al. [CNM04]. This is an agglomerative clustering algorithm that performs a greedy optimization of the Modularity (Q) [New03]. At the end we retrieved the number of communities in which the network is structured, the corresponding value Q of the modularity, and the nodes associated to each community. We performed the computation of the clustering coefficient using the implementation included in the IGraph software [GC06]. To study the system’s evolution we used the following approach. We first carried out the analysis for each release, and then we assembled together different releases, according to a temporal evolution. More precisely, for the 5 releases of our dataset, we studied the evolution of the system by cumulating the first and the second releases in a single set, then adding the third release to this first set to obtain a second set and so on. This way we were able to make predictions about the next release starting from those cumulated in the previous assembly.

3 Results

Figures 1a and 1b show the distributions of the average bugs number (ABN, Fig. 1a) and of clustering coefficient (CC, Fig. 1b) with respect to the number of communities (NOC) for all the sub-projects of all the releases. Although the scatterplots for the relationship between NOC and other metrics are sparse, the reported scatterplots show the existence of a power-law-like relationship between the maximum values of the mentioned metrics. This led us to hypothesize a linear relationship between the maximum values of CC and the ABN. In Tab. 2, on the left, the power law exponents, the correlation coefficient, the $\chi^2$ and the degrees of freedom (dof) for the best fitting in log-log scale are reported. They refer to different “cumulated” releases for the relationship between CC and NOC. Table 2 shows how the power laws parameters do not change significantly from one cumulated release to another. This suggests the existence of a progressively more stable behaviour during software evolution, where the fitting with a power law becomes more accurate and tends to a fixed value as new releases are added in the cumulated dataset. The same considerations can be applied to the relationship between maximum ABN and NOC. The scatterplot portrayed in Fig. 2 shows the relationship between the maximum defect density versus the maximum clustering coefficient, for all the cumulated releases, along with the best fitting straight line. We investigated if, starting with a dataset of $N$ releases, the best fitting curve for the cumulated $N-1$ releases
Figure 1: Scatterplot of the relationships between the studied metrics.

(a) Average Bug Number vs. Number of Communities
(b) Clustering Coefficient vs. Number of Communities

<table>
<thead>
<tr>
<th>Eclipse</th>
<th>2.1 - 3.0</th>
<th>2.1 - 3.1</th>
<th>2.1 - 3.2</th>
<th>2.1 - 3.3</th>
</tr>
</thead>
<tbody>
<tr>
<td>α</td>
<td>-1.010</td>
<td>-0.917</td>
<td>-0.977</td>
<td>-0.986</td>
</tr>
<tr>
<td>r</td>
<td>-0.654</td>
<td>-0.667</td>
<td>-0.715</td>
<td>-0.712</td>
</tr>
<tr>
<td>(\chi^2)</td>
<td>0.075</td>
<td>0.057</td>
<td>0.087</td>
<td>0.119</td>
</tr>
<tr>
<td>dof</td>
<td>16</td>
<td>17</td>
<td>20</td>
<td>21</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Eclipse</th>
<th>max ADD</th>
<th>max CC</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\chi^2 / \text{dof})</td>
<td>0.361</td>
<td>1.005</td>
</tr>
</tbody>
</table>

Table 2: Left: Data regarding maximum Clustering Coefficient vs Number of communities for Eclipse: exponent \(\alpha\), correlation coefficient \((r)\), value of Chi Squared \((\chi^2)\) and number of degrees of freedom \((\text{dof})\). Right: fit data for the maximum defect density and maximum clustering coefficient versus the number of communities: correlation coefficient \((r)\), normalized Chi squared \((\chi^2)\), and number of degrees of freedom \((\text{dof})\).

could also be a good fit for the \(Nth\) release. In order to measure the forecast accuracy we adopted a \(\chi^2\) test. Table 3 reports the results of the best fitting for the relationship between CC and ABN showing that the linear correlation is not very high. Nonetheless, the \(\chi^2\) test returns an high level of significance. Table 2 reports the results of the analysis on the forecast for software quality. We computed the ratio between the \(\chi^2\) and the degrees of freedom. According to the results reported on Table 2, on the right, the \(\chi^2\) values are close to 1, meaning that for the given degrees of freedom the fits are good.

Table 3: Fit data for the maximum defect density vs maximum clustering coefficient: correlation coefficient \((r)\), normalized Chi squared \((\chi^2)\), and number of degrees of freedom \((\text{dof})\).
4 Conclusions

In this work we presented a longitudinal analysis on the evolution of a large software system with a focus on software defectiveness. Through a complex network approach we were able to study the structure of the system by retrieving the community structure of the associated network. After retrieving the number of defects associated to the software network classes, we performed a topological analysis detecting the community structure. We found a power law relationship between the maximum values of the clustering coefficient, the average bug number and the division in communities of the software network. This lead to a linear relationship between the maximum values of clustering coefficient and of average bug number. We show that such relationship can in principle be used as a predictor for the maximum value of the average bug number in future releases.

References


[BB0] LOVRO UBELJ, SLAVKO ITNIK, NELI BLAGUS, and MARKO BAJEC. Node mixing and group structure of complex software networks. Advances in Complex Systems, 0(0):1450022, 0.
Null Check Analysis

Haidar Osman
Software Composition Group
University of Bern
osman@iam.unibe.ch

Abstract
Null dereferencing is one of the most frequent bugs in Java systems causing programs to crash due to the uncaught NullPointerException. Developers often fix this bug by introducing a guard (i.e., null check) on the potentially-null objects before using them.

In this paper we investigate the null checks in 717 open-source Java systems to understand when and why developers introduce null checks. We find that 35% of the if-statements are null checks. A deeper investigation shows that 71% of the checked-for-null objects are returned from method calls. This indicates that null checks have a serious impact on performance and that developers introduce null checks when they use methods that return null.

1 Introduction
In a previous work [1], we analyzed the bug-fix code changes in a software corpora that consists of 717 popular (> 5 stars) and large (> 100 KB) Java projects cloned from Github [1]. We found that a considerable number of the bug fixes are recurrent, confirming literature findings [2][3][4]. However, an interesting finding was that missing null check is the most recurrent bug arising across the projects in both corpora.

In this paper, we analyze the null checks in Java systems to gain a better understanding of the missing null check bug pattern. Surprisingly, our study shows that 35% of the conditional statements in our corpus are null checks. This result suggests that the null checks have a serious negative impact on the performance of Java systems. Also, we find that 25% of the checked objects are member variables, 24% are parameters, and 51% are local variables. Unsurprisingly, 71% of the checked objects come from method calls. In other words, developers mostly insert null checks when they use methods that return null.

2 Extracting Syntactic Bug-Fix Patterns
We analyzed the evolution history of each project in our corpora to extract bug-fix code changes. Our approach follows these steps:
1. Extract the bug fixing commits.
2. Extract method bodies before and after the commit.
3. Diff the text of every changed method.
4. Anonymise the changed code.

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

86
(a) Every number is replaced by N.
(b) Words are anonymized using numbered letters (T0, T1, T2, etc.).
(c) Every whitespace is removed.

For example, following these steps, we can end up with a bug-fix code change such as:

```java
((View) getParent(1)).invalidate() ---> View parent = (View) getParent(1);
  if (parent != null) parent.invalidate();
```

Anonymizing the previous bug-fix code change results in the bug-fix syntactic pattern:

```java
((T0)T1(N)).T2() ---> T0T3 = (T0)T1(N);
  if(T3!=null)T3.T2();
```

After investigating the results, we found that “missing null check” is the most frequent bug category [1]. This bug is usually fixed by adding a null check like in the following patterns:

```java
[
] ---> if(T0==null) return;
[
] ---> if(T0==null) return null;
[
] ---> if(T0!=null)T0.T1();
[
] ---> if(T0==null) throw new T1();
[
] ---> if(T0==null) return T0;
```

3 Null Check Analysis

The severity of the missing null check bug pattern comes from the fact that it causes software systems to crash due to the often uncaught NullPointerException in Java. A manual inspection of 200 instance of the missing null check pattern shows that 70% of the checked-for-null objects come from method calls. In other words, when a method “may” return null, the returned object should be checked before it is dereferenced.

In this study we aim to better understand the checked-for-null objects. More specifically, we want to understand the types of these objects and how they are usually initialized. So, we developed a tool that analyzes Java source code and extract information about the null checks and the checked-for-null objects. Our tool analyze each Java source file as follows:

1. It parses the Java source file and extracts the null check (conditional statements).
2. It extracts the expression that is checked against null.
3. It parses the expression and determines its type (name expression, method call expression, array access expression, etc.).
4. When the expression is a name expression, the tool determines its type (member variable, local variable, or parameter).
5. When the expression is a name expression, the tool also extract all the assignment statements that appear textually before the null check and parses them to extract the type of the assigned expression (method call, value, etc.)

4 Where Does This Null Come From?

We ran the null check analysis on our corpus. We found that 35% of the conditional statements are null checks. Similarly, but from another point of view, Kimura et al. found that the density of null checks are from one to four per 100 lines of code [5]. These results combined pose serious doubts regarding the effect of the null checks on performance, but this is outside the scope of this paper.

We also found that 25% of the checked-for-null objects are member variables, 24% are parameters, and 51% are local variables. The analysis of the assignments of these objects show that 71% of the time they are assigned the results of method invocations.
5 Implications

The results of this study imply that null checks are mostly applied on objects coming from method invocations. In other words, when methods possibly return null, they tend to cause NullPointerExceptions in the invoking methods forcing developers to add null checks. This motivates the development of a tool that checks whether the returned object is checked before it is put in use, when the invoked method may return null.

Also the fact that parameters get checked against null often, indicates poor API design. We argue that null should not be passed as a parameter to methods. Method overloading should be used to prevent such scenarios.

References


Detecting Violations of CSS Code Conventions

Boryana Goncharenko
boryana.goncharenko@gmail.com
University of Amsterdam, The Netherlands

Abstract

Code conventions are used to preserve code base consistency and express preference of a particular programming style. Often, code conventions are described in natural language and developers need to apply them manually. Existing tools typically offer a predefined set of rules that cannot be customized. This project aims at allowing CSS developers to express an arbitrary set of code conventions and detect their violations automatically. The solution requires designing a domain specific language capable of expressing existing conventions and implementing its interpreter to automatically detect their violations.

1 Introduction

Coding conventions put constraints on how code should be written in the context of a project, company or programming language. Style guides can comprise conventions that refer to whitespacing, indentation, code layout, preference of syntactic structures, code patterns and anti-patterns. They are mainly used to achieve code consistency, which in turn improves the readability, understandability and maintainability of the code [2].

Style guides are often designed in an ad hoc manner. Coding conventions typically live in documents that contain a description of each rule in natural language accompanied by code examples. This is the case with the style guidelines of Mozilla [8], Google [6], GitHub [10], WordPress [11] and Drupal [4]. To apply the conventions, developers first need to read and interpret them. Such an approach introduces a number of issues. Using natural language can make guidelines incorrect, ambiguous, implicit or too general. Another problem is that developers apply conventions manually, which increases the chances of introducing violations involuntarily. There are tools that check for compliance with guidelines, however, they are often hard to customize or limited to one type of violations, e.g. only whitespacing.

The core idea behind the project is to provide a solution that lets developers express an arbitrary set of coding conventions and detect their violations automatically in an IDE. Writing conventions in an executable form could assist authors in detecting incorrect, ambiguous or inconsistent guidelines. Automatic detection of violations could minimize the effort required by developers to write code that complies to the guidelines. To meet the constraints of a Master’s project, the implementation is limited to the domain of Cascading Style Sheets (CSS).

The project requires determining the need for CSS code conventions in organizations, collecting and analyzing available style guides, and providing a way to express conventions. Specifically, the project attempts to answer the following set of questions:

• Do we need code conventions in CSS?
• What CSS code conventions exist?
• How to detect violations of code conventions?
2 The Need for CSS Code Conventions

Despite the new features added in the second [1] and third [5] versions of CSS, the language has obvious limitations, e.g. lack of variables. A number of preprocessors have evolved to tackle the downsides of CSS. Solutions such as SASS [3], LESS [9] and Stylus [7] offer enhanced or even different syntax and translate it to CSS. preprocessors are not only ubiquitous but are widely adopted in practice. Obviously, using such solutions avoids the need for CSS code conventions because the code is generated and not maintained directly. Thus, CSS code conventions make sense only if developers handcraft the CSS files.

To determine whether CSS is written and maintained as opposed to being generated, all commits to open source repositories hosted on GitHub for 2015 (up to April) are being analyzed. If the commit contains a file with extension .scss, .sass, .less or .styl, it is considered preprocessor maintenance. In case the commit contains files with the .css extension and no preprocessor extensions, it is considered maintenance of CSS. To exclude cases in which developers commit third-party CSS, only commits that modify CSS files are taken into consideration. Commits that add or delete CSS files are ignored.

Preliminary results show that from all commits that include some form of CSS, half of the commits use preprocessors file and the other half — plain CSS.

3 Existing CSS Code Conventions

The CSS community has produced a pool of recommendations, best practices, and style guides, but how to choose among them? Since the primary organization responsible for the specification of CSS has not recommended code conventions, any selection strategy based on the author of the conventions could be considered cherry picking.

To determine the set of code conventions, two searches with the keywords “CSS code conventions” have been made using the search engines http://duckduckgo.com and http://google.com. The first 50 results of each search have been analyzed. From each result only conventions about pure CSS are taken into account and guidelines for CSS preprocessors are ignored. In case the result is an online magazine or a blog post that links to other resources, these references are considered as results and analyzed separately.

Results of the two searches include the CSS coding guidelines of CSS professionals as well as leading companies, e.g. Google, Mozilla, GitHub, Wordpress. The accumulated corpus consists of 165 unique coding conventions.

4 Expressing CSS Code Conventions

Analysis of convention corpus shows that every convention could be defined as a combination of the constraints it enforces and the context in which these constraints should be applied. Consider the convention “Class names should be lowercase”. Its context are the nodes of the CSS code that represent classes and its constraint is that the name property of class nodes should be lowercase.

Conventions can have more complex contexts that describe not only one node, but also a combination of nodes. For example, the guideline “Do not put quotes in URL declarations” requires selecting the argument node of a URL function node. A context can be seen as a description of CSS nodes and their relations, i.e. a description of a code pattern.

Each convention can impose a variety of constraints. They can refer to the type, attribute values or context of nodes. Constraints often require nodes to possess knowledge from the CSS domain. Such an example is a convention that needs to find vendor-specific properties.

Style guidelines could specify relations between two or more conventions. Often a given convention is followed by a number of exceptions, for example: “Use single quotes for strings. However, if the string is in a charset, use double quotes”. In such cases the context of the exception must describe patterns that are a subset of the patterns described by the context of the original convention, i.e. the exception should have a more specific context than the general convention.

All conventions in the corpus are further categorized based on the type of constraints they specify (sublists provide examples of conventions that fall in each category):

**Whitespace and Layout** category contains rules that constrain the overall layout of the code. It includes conventions related to whitespace, indentation and comments. Conventions in this category are highly specific and their violations can be refactored automatically. Examples include:

- Use four tabs for indentation
• Put one blank line between rulesets
• Disallow spaces at the end of the line

Syntax Preference category comprises conventions that express preference of a particular syntax. Note that rules in this category do not aim at ensuring CSS validity, but choose between syntactic alternatives. For example, both single and double quote strings are valid in CSS and a convention may narrow down the choice of the developer to single quotes. Conventions in this category are specific and their violations can be refactored automatically. Examples include:

• Use lowercase for id and class names
• Require a semicolon at the end of the last declaration
• Use strings with single quotes

Programming Style category consists of conventions that put constraints on how CSS constructs are used to achieve a certain goal. They express good and bad practices in the CSS domain and are used to improve maintenance and performance, or to avoid a bug in a particular implementation. Violations of such conventions cannot be refactored automatically. Examples are:

• Do not use the universal selector
• Avoid using !important

5 Conclusion
The topic of this presentation is a work in progress corresponding to a graduate project (UvA MSc Software Engineering). The main objectives of the project are: (1) to find whether developers use pure CSS or prefer CSS preprocessors such as Less/SASS/Stylus and (2) to aggregate a set of CSS code conventions used in practice. Possibly, the main contribution of the project will be a domain specific language for expressing existing conventions and its interpreter for automatic detection of their violations.

References
Design by Contract and Modular Reasoning in Aspect-Oriented Languages

Tim Molderez
Software Languages Lab
Vrije Universiteit Brussel, Belgium
tmoldere@vub.ac.be

Dirk Janssens
Ansymo
Universiteit Antwerpen, Belgium
dirk.janssens@uantwerpen.be

Abstract

Aspect-oriented programming aims to separate crosscutting concerns into their own modules, called aspects. While aspects can achieve this at a syntactic level, this is done at the expense of modular reasoning: Whenever a method call is made, all aspects should be inspected to determine whether or not that call’s behaviour will be affected by an aspect. To restore modular reasoning, we present a two-part approach that does not restrict the programming language itself, but rather governs how to write contracts for aspect-oriented programs.

1 Context

This is a summary of the journal paper "Modular Reasoning in Aspect-Oriented Languages From a Substitution Perspective" [6]. The goal of aspect-oriented programming (AOP) is to separate so-called crosscutting concerns, which are those concerns that are inherently tangled with one or more other concerns. Common examples include logging, profiling or authentication. AOP, while not intrinsically related to OOP, is often implemented as an extension to an object-oriented language. One of the key additions of AOP is the pointcut-advice mechanism, which allows the developer to implement the behaviour of crosscutting concerns into syntactically separate modules called aspects. In short, this is possible because the pointcut-advice mechanism is able to implicitly alter the behaviour of multiple other modules. While such a mechanism is quite powerful, it should also be wielded with care: if the behaviour of classes can be altered implicitly, it is possible that clients of those classes can perceive unexpected, or even undesired, behaviour.

In general, whenever a module is used, every single aspect should be inspected as well to be certain whether or not its behaviour will be implicitly modified by an aspect. This goes against the grain of modular reasoning, i.e. the ability to understand a module by only examining the module itself, and those modules it explicitly refers to. Not only is this notion of modular reasoning important for program understanding, but it is necessary as well to be able to safely extend a program without causing surprising behaviour in the existing code.

In order to restore modular reasoning, we present a two-part approach from a design by contract [5] perspective. Rather than restricting the programming language itself to enforce modular reasoning, the approach describes how contracts on methods and advice, focusing on pre- and postconditions, can be written to retain modular reasoning. This approach considers various subtleties of AOP languages: call and execution pointcuts, pointcuts that can only be determined at runtime, higher-order advice (advice on advice), (non-)proceeding around advice, overriding advice and first-class aspects.

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

2 ContractAJ

For the purpose of studying modular reasoning in AOP languages, we have developed a minimal aspect-oriented language called ContractAJ, which adds AOP support to the ContractJava [2] language. Both the syntax and operational semantics of ContractAJ have been formally specified. ContractAJ is representative for a wide range of aspect-oriented languages, which is reflected in a number of design decisions that unify aspect-oriented concepts with object-oriented ones, keeping the language small but expressive.

To illustrate ContractAJ’s syntax, consider the small example program in Fig. 1. The ContractAJ language is quite AspectJ-like [3], but note that we do not have an explicit aspect keyword to indicate modules representing a crosscutting concern. Instead, similar to EOS-U [7], the notion of aspects and classes is unified: aspects are plain classes, which are allowed to contain pointcuts and advice. The example of Fig. 1 implements a caching crosscutting concern as a separate Cache class. The last line in this example is an advice that expresses: Whenever the List.set method is called, execute the given body of code (shown as ellipses) instead, which implements the cache update behaviour. Another difference with AspectJ is that every advice has a name, store in this example, which makes it possible to override advice, and adds more fine-grained control for higher-order advice.

The pointcut language, expressing when an advice body should be executed, is minimal but representative: The call and execution of methods (and advice) can be intercepted, and can be combined with an if pointcut construct to add runtime conditions to a pointcut. Contracts of methods and advice are specified using the @requires and @ensures annotations, respectively representing the pre- and postconditions of methods/advice. Finally, the proc keyword can only be used in the contracts of advice. When used in a pre/postcondition, it is a shorthand for the pre/postcondition of the method (or advice) being intercepted/advised.

```java
class List{
    @requires i>=0 && i<this.length()
    @ensures this.get(i)==val
    void set(int i, Object val) {...} 
}

class Cache {
    @requires proc
    @ensures proc && this.isCached(i, val)
    around store: call(void List.set(int i, Object val)) {...} 
}
```

Figure 1: Example of an ASP-compliant advice

3 The advice substitution principle

As mentioned before, an aspect is allowed to implicitly modify the behaviour of other modules. However, to allow for modular reasoning, such modifications should not cause any unexpected behaviour. The caching example of Fig. 1 is quite harmless in this regard, as a cache will not interfere with the expected behaviour of the List class. While not all aspects will be as harmless, there are several useful cases that are harmless, e.g. logging, caching, profiling, contract enforcement, ... To make this notion of harmlessness more precise, we have defined the advice substitution principle (ASP), which is the AOP analogue to behavioural subtyping [4] in OOP. The ASP is defined as follows for around advice:

ASP for around advice. Consider an around advice `a` in type `t` that is applied to join point `u.x`, representing a method call or an advice execution. If `x` is a method, `u` is the static type of the receiver. If `x` is an advice, `u` is the class containing `x`. The around advice satisfies the ASP if and only if, for all objects of type `t`:

- **The precondition of `t.a` must be equal to or weaker than the precondition of `u.x`.**
- **The postcondition of `t.a` must be equal to or stronger than the postcondition of `u.x`, if the precondition of `u.x` held in the pre-state.**
- **The invariant of `u` should be preserved in `t`.**

In short, this principle states that each advice should take into account the contracts of each method call (or advice execution) it advises. There also are variants of the ASP defined for before and after advice, where
the advice body is executed before or after the advised method, rather than replacing its execution. The main difference is that these variants must take into account the effects of an implicit proceed call, as the effects of the method/advice body being intercepted are not part of the pre/postconditions of the advice body. What is important to consider as well is that the ASP is defined in terms of a join point, i.e., a well-defined point in time where an advice can intercept. This implies that the ASP must hold at all join points to which the advice is applied. As a join point is a runtime concept, and because contracts should be understandable without any runtime information, the developer should essentially make an overapproximation and ensure that the ASP holds at all join point shadows.

4 The @advisedBy clause

The example of Fig. 1 is one that satisfies the rules of the ASP. However, let us now consider the example of Fig. 2, which implements authentication and authorization crosscutting concerns in the Security class. The preconditions of the authenticate advice state that they are the same as the preconditions of the join point being advised. The same holds for postconditions, but only if the user is currently logged in. In this example, the ASP rule regarding postconditions does not hold, because nothing is ensured when the user is not logged in, which clearly is a weaker postcondition than the one of the join point being advised. In other words, this advice inherently breaks the ASP, as its very purpose is to not execute the desired operation if the user is not authenticated.

```java
class Account {
    @requires this.getAmount() >= m && m>0
    @ensures this.getAmount() == old(this.getAmount()) - m
    @advisedBy Security.authenticate, Security.authorize
    int withdraw(int m) {...} ...
}
```

```java
class Security {
    @requires proc
    @ensures if(isLoggedIn()) {proc}else{true}
    around authenticate: call{void Account.withdraw(int m)} {...}

    @requires proc
    @ensures if(isAuthorised()) {proc}else{true}
    around authorize: call{void Account.withdraw(int m)}
        && if(isEnabled()) {...} ...
}
```

Figure 2: Example usage of the @advisedBy clause

If we stopped here, then modular reasoning is no longer possible and the ASP serves no purpose. However, it is possible to restore modular reasoning by means of a simple annotation. The problem at hand is that the advice may produce unexpected behaviour, as any clients of e.g. Account.withdraw may not be aware that the user had to be authenticated. To correct this situation, these clients need to become aware. This is done by adding the @advisedBy annotation/clause to the method bodies being advised, Account.withdraw in this example. This annotation lists those advice that a method body is expecting to be advised by. The ordering of this list also matters: In this case, we expect that the authenticate advice is executed first, which then proceeds to executing the authorize advice body, which finally proceeds to the withdraw method. Due to the addition of the @advisedBy clause, we should also refine the meaning of the proc keyword: It refers to the pre/postconditions of the body that we expect (i.e., without having runtime knowledge) to be executed in a proceed call. In turn, this implies that an advice should be aware of all locations where it is mentioned in an @advisedBy clause, as it should know if there is another advice that we expect to execute on a proceed call. This is similar to the discussion where an advice that satisfies the ASP should be aware of all its join points.

At first glance, using the @advisedBy clause appears to be quite restrictive in terms of modularity, as the clause expects certain advice to be present. However, any advice mentioned in an @advisedBy clause still remains open for extension. Similar to methods, the behaviour of an advice can still be extended by overriding it. That is, if an @advisedBy clause expects a certain advice to be executed, that expectation may be fulfilled just as well by an overriding advice. As such, any aspects mentioned in an @advisedBy still satisfy the open-closed principle; they remain open for extension, but closed for modification.

Finally, it is important to consider that adding an @advisedBy clause to a method effectively changes its pre/postconditions towards any clients of that method. That is, the pre/postconditions of the method

94
body itself needs to be composed with those of the advice in the @advisedBy clause to obtain the "effective" pre/postconditions. While the definition of effective pre-and postconditions is not shown here, we should note that this definition takes into account the fact that pointcuts may include runtime conditions. This is done by simply including those conditions into the effective pre/postcondition itself, such that clients are aware of these conditions as well.

5 Modular reasoning in ContractAJ

To conclude, we have presented an approach to modular reasoning for the ContractAJ language, being representative for a wide range of aspect-oriented languages. The actual usage of this approach to ensure modular reasoning can be summarised as follows: Like object-oriented programs, all classes should still adhere to the strong behavioural subtyping [1] rules. If a class has no advice, then these are the only rules that it must satisfy, which is positive considering that the majority of classes in most AOP systems do not contain any advice.

When writing contracts for an advice, a choice must be made: If it is desired to remain unaware/oblivious of an advice, then it must adhere to the ASP. If it is however desired or necessary to become aware of an advice, then the @advisedBy clause must be used. Finally, in case both ASP-compliant advice and advice in an @advisedBy clause could advise the same join point, then the ASP-compliant advice must get a lower precedence.

In the full journal paper, there also is a proof which shows that, if this approach is used correctly, then the property of modular reasoning will be ensured. We also assist developers in using this approach by means of a runtime contract enforcement algorithm, which adds contract enforcement aspects that will produce an error whenever a contract is violated, either because the implementation of a body does not correspond to its contracts, or because our approach was used incorrectly. An implementation of this algorithm, called adbc, also is available for AspectJ1.

References


1The adbc contract enforcement library is available for download at: https://github.com/timmolderez/adbc
An empirical study of identical function clones in CRAN

Maëlick Claes
Software Engineering Lab
maelick.claes@umons.ac.be
University of Mons

Abstract

Code clone analysis is a very active subject of study, and research on inter-project code clones is starting to emerge. This talk presents an empirical study of identical function clones in the CRAN package archive network, in order to understand the extent of this practice in the R community. Depending on too many packages may hamper maintainability as unexpected conflicts may arise during package updates, but duplicating functions from other packages may also reduce package maintainability. We study how the characteristics of cloned functions in CRAN snapshots evolve over time, and classify these clones depending on what has prevented package developers to rely on dependencies instead.

1 Introduction

Analysing the impact of inter-project code cloning is an emerging topic of research in the code cloning community [4, 5]. When depending upon a third-party library, errors may be introduced inadvertently when upgrading to a newer version of the library one depends upon. Duplicating code instead is an alternative but may be detrimental to the maintainability in the long run.

This presentation summaries our findings [2] on the practice of cloning function code between packages contained in a large open source package archive maintained by a specific community of developers. The case study that we have chosen is CRAN, the official repository of packages of the statistical project R. The R developer community is part of a rather specific ecosystem that includes many non-programmers such as statisticians, biologists and economists. They generally don’t have a software engineering background as strong as other programming language communities.

Identical cloned functions across packages appear to be omnipresent in CRAN. With our study, we aim to understand why this is the case, and how this practice evolves over time. In particular, we want to understand to which extent functions are identical clones across packages, why R package developers clone functions, and if those clones could be avoided by the introduction of explicit dependencies.

2 About CRAN

CRAN is the official R package repository. It dates back from 1997 and contains over 6,000 packages and 9 million lines of R code. While other researchers have studied the evolution of this ecosystem [3], we are not aware of any study that focuses on the presence of function clones in CRAN. CRAN has a rather strict policy to ensure package quality. Packages need to pass a series of tests to be accepted in the repository. These tests are rerun daily and package maintainers must ensure that their package still passes these tests to avoid their package being archived. This policy puts a heavy burden on package maintainers, especially if the package no
longer passes the tests due to an update of some dependent package over which the maintainer has no control [1].

We hypothesise that R package maintainers sometimes resort to copy-and-paste reuse to reduce the extent of this problem. Indeed, copying the code of a function they want to reuse from another package requires less effort at the short term than explicitly depending on that package and take the risk that future changes in that package may break their own one. The other way around, we also suspect that some function clones between R packages exist because the declaration of an explicit package dependency does not allow to access the required function. This is for example the case if the package is local or is not exported in the namespace.

To analyse the history of clones in CRAN we first parsed the source code of each version of each CRAN package using built-in R functions. Working with abstract syntax trees (AST) allows us to ignore all differences in code format of otherwise identical function bodies. For each function body’s AST we compute its SHA-1 hash algorithm. Two functions that have the same hash value can be considered as identical with a negligible probability of false positives.

For the purpose of the empirical analysis we excluded all functions whose body contains less than 6 lines of code because this value allows us to avoid most of the small code fragments leading to “accidental clones”. We also excluded all intra-package clones, i.e., clones that appear within the same package.

3 Observed Clone Cases
Before delving into an empirical analysis, we focus on a subset of CRAN packages showing an interesting cloning behaviour.

Coexisting package versions In some CRAN snapshots, two different “versions” of the same package may coexist. While these packages have a different names, one of them can be regarded as the new version of the other. A valid reason for this clone case is to allow existing packages to continue to depend on the old version, while already exposing the new version with extended functionality.

An occurrence are packages lme and nlme that have coexisted for some time. Both packages provide statistical model functions. nlme adds non-linear models to lme and actually replaced it in later snapshots of CRAN. They share more than 600 identical function clones totalling over 7,000 lines of code.

A second example is the pair of packages np and npRmpi. The latter package is a version of np that uses MPI (Message Passing Interface). Both are currently available on CRAN, are maintained by the same person and share more than 10,000 lines of code.

The fork package Related to the previous case, forked packages continue to coexist with the package they have forked from. Rcmdr offers a graphical interface to use R statistical functions. Package QCAGUI provides a graphical interface for the QCA package, and can be considered as a fork of Rcmdr with most of the statistical related features removed. Rcmdr and QCAGUI share more than 8000 lines of code.

The frequently cloned package For some packages, most functions have been cloned by other packages. distr, contains 182 lines of code, and all its global functions have been cloned by different packages.

The utility package An utility package bundles together a lot of functions cloned from many other packages. DescTools, which gathers functions for basic statistics that are scattered across different packages.

The popular package A popular package contains specific functions that are cloned by a lot of other packages. An example of such a package is MASS, a well-known and widely reused statistical package. It has 16 functions that have been cloned by 16 different packages for a total of 180 code lines.

The popular function A popular function is a function that is cloned by a lot of different packages, while the other functions of the same package are not. An example is the package combinat, whose function permn of 151 lines of code is cloned by 7 different packages.

4 How prevalent are clones in CRAN?
Figure 1 shows that the number of packages containing clones increases over time, up to 2,000 packages containing clones today. This corresponds to 24.2% of all packages. The trend follows the overall exponential growth trend of the number of available CRAN packages. It also shows that while the number of lines of cloned code is relatively much smaller than the total number of lines of code, the packages concerned by these clones are the biggest ones in CRAN.

From these findings we can conclude that

• The cloning phenomenon in CRAN impacts quite a lot of packages (up to 25%).

97
The ratio of packages impacted by cloning appears to have stabilised.

While cloning impacts very few lines of code compared to the overall size of CRAN, it still impacts more than 250,000 code lines. Moreover, those lines are included in packages that represent around 50% of all code lines in CRAN.

5 Why did clones appear?

Our goal is to understand the reason of existence for those clones and whether these clones could have been avoided. To fulfil this goal we study in more detail the CRAN snapshot to date 2014-12-01.

We counted 6,253 non-archived packages and 3,184 clone sets, involving 7,366 function clones in 1,409 distinct packages. In total, this amounts to 162,327 lines of cloned code out of 8,383,417 in total (i.e., <2%).

In order to understand why these clones exist we studied origin(\(C\)), the origin function of each identified clone set \(C\). The origin corresponds to the function with the oldest date, implicitly assuming that all other clones belonging to the same clone set were copied from it. We found exactly 1 origin package for all 3,184 considered clone sets.

Is the origin function anonymous (i.e., not stored in any variable)? and declared locally (i.e., declared inside another function)?

Among the 3,184 considered origin functions (one for each clone set), we identified 796 functions (i.e., 25%) that were either anonymous or local. 250 of these were both anonymous and local.

Does the origin still exist in the most recently available package version?

For the 2,388 origin functions that were globally visible (i.e., not local) in the origin package version, 202 (8.45% of all global origin functions) were no longer available in the latest considered version of the same package, either because the function has been removed or changed somehow over time.

Is the origin available as a public function to the package users?

The current CRAN policy requires packages to define a “NAMESPACE” file that lists which functions or objects are exported by the package. Those exported functions are all the functions that the package user is allowed to use. Because NAMESPACE files can use regular expressions and because package environments can be modified
dynamically, we extracted the list of exported objects by loading each package in a virtual machine containing a snapshot of CRAN corresponding to the release date of the package.

Out of the 2,186 origin functions that still exist today, we weren’t able to retrieve the list of exported functions for 287 of them (i.e., 13%). Of the remaining 1,899 origin functions, 673 were exported while 1,226 were not.

In summary, it turns out that, for the considered snapshot, cloning cannot be avoided for the majority of clones in each identified clone set. Of the 3,184 origin clones, 25% (796) of them were local functions that cannot be depended upon no matter whether they are exported or not by their containing package. Of the remaining 1,899 global functions, only 35% (673) were public ones.

6 Is it possible to remove clones and how?

We have seen that it was only possible for a small fraction of the clone sets to remove identical clones by adding a dependency to the origin function. However, we still need to check whether this dependency already exists or if it could be added. This dependency cannot be added if the package containing the origin (directly or indirectly) depends on the package containing the clone, since otherwise a cyclic dependency would be introduced.

From the previously identified 673 public origin functions there are 49 copies where there is an existing direct dependency to the origin package. In the opposite direction we found 20 packages for which their origin package directly depends upon them and only one for which the origin package indirectly depends upon it.

While the majority of clones cannot be removed by depending upon the package that contains the origin function, perhaps a dependency can be added to another package containing a clone of the origin. From the 3,458 non-origin clones of the 2,511 clone sets for which the origin is not refactorable by adding a dependency, only 250 have a public copy of the function. This gives a total of 317 clones that could potentially be removed by adding a dependency to the package with this public clone. 31 already have this dependency and 18 have a reverse dependency to the package declaring the public clone, making it impossible to add the dependency.

7 Conclusion

We studied the problem of inter-project software clones across R packages contained in the CRAN package repository over a 15-year time period. While identical cloned functions appear to be present in a rather small portion of the code of all packages, they still represent hundred of thousands of lines of code. We were able to identify an important number of clones that could theoretically have been avoided by introducing explicit dependencies to another package containing the function clone. Only in very few cases it was not possible to add such a dependency because it would give rise to a cyclic dependency.

We also found valid reasons why cloned functions appeared. In the majority of cases, cloning could not be avoided because the original function being cloned was local or private. This made it technically impossible to reuse the function by simply depending upon the package defining it. Nevertheless, we believe that R package maintainers still lack information about, and could benefit from, feedback on the presence of clones in their package and dedicated tools to help them deal with it.

References


Driving the Evolution of Cloud Software
towards Energy Awareness

Christophe Ponsard, Jean-Christophe Deprez, and Dimitri Durieux
Software and System Engineering Deparment
email: {cp, jcd, ddu}@cetic.be
CETIC Research Centre, Gosselies, Belgium

Abstract

ICT energy efficiency is a growing concern. Large effort has already been spent making hardware energy aware and improving hardware energy efficiency, and further progress in this area requires to consider evolving the software layer to more energy awareness. Although specific work is devoted in areas like embedded/mobile systems, much remains to be done at software level for Cloud applications. Software developers need an energy aware Cloud infrastructure as well as code development support to make informed decisions about energy efficiency and compromise with other important non-functional requirements like performance. This evolution to energy awareness impacts a number of artifacts including requirements, design, code and tests.

In the scope of this paper, we focus on the evolution of analysis phase (requirements, design and test load specification) with the limited goal of enabling the collection energy measurement data that can be used for a design time evolution and will later enable dynamically adaption scenarios. In order to help Cloud application developers, we propose a framework composed of (1) a Goal-Question-Metric analysis of energy goals, (2) a UML profile for relating energy requirements and associated KPI metrics to application design and deployment elements, and (3) an automated Cloud deployment of energy probes able to monitor those KPI and aggregate them back to questions and goals.

1 Introduction

The expansion of ICT both at professional and personal levels induces increasingly larger amounts of data exchanged (high resolution pictures, videos) and processed (Big Data), increasing connectivity of all devices (mobile devices, Internet of Things) and higher penetration (on business domains, emerging countries). This evolution raises the energy required to run ICT to a level that would be dramatic if ICT energy efficiency was not improving simultaneously. Between 2007 and 2012, global ICT consumption raised from 3.4% to 4.6% of the overall energy consumption and the ratio for data centres also grew from 1% to 1.3% of the global energy consumption hence a 30% increase [Int13].

Copyright © by the paper’s authors. Copying permitted for private and academic purposes.
To reach another stage of energy saving, it is required to consider the software layer. While several initiatives have already studied how to reduce energy consumption of mobile or embedded devices, little has been done for improving the energy performance of the service-side computation part of application, in particular, in the context of Cloud computing. To bootstrap the process of energy based pricing model for Cloud service at infrastructure, platform or application level, it is necessary to develop a Cloud stack capable to record energy consumption at each layer and to facilitate negotiations between a customer and a provider where energy consumption is one of the factors. In a second step, self-adaptation capabilities in the Cloud stack middleware or in the Cloud applications can then enable dynamic energy savings. An number of Green Architectural Tactics have been described in [PLL14] they refine the above energy monitoring and self-adaptation with more detailed tactics like scaling down, workload scheduling, energy brokering and service adaptation.

Not to limit self-adaptation to be implemented in Cloud stack middleware, tools are needed to help development teams to learn how their applications consumes energy and how to refactor these application to achieve additional energy savings. Several refactoring have been described in a recent paper [PSNC15]. A relevant refactoring include the correction of energy hotspot in the code. In order to enable self-adaptation, another useful refactoring is to insertion variability point that can be controlled in order to enable more energy efficient reconfiguration.

These development tools must therefore encompass all development phases from requirement and design to construction of Cloud application and workload test development. Work in this direction is actively progressing in the scope of several EU projects, for instance, FP7 projects such as [ASC13], [ECO12], or [ENT13]. In the scope of this work we rely on the ASCETiC architecture which is illustrated in a simplified form in Figure 1. The architecture covers the three Cloud layers: SaaS, PaaS, and IaaS. The developer tools are located at the higher SaaS layer and interact with the PaaS to deploy energy collection probes. Those take benefits of specific data collection and estimation components at the PaaS and SaaS layer that will efficiently collect the required energy data and make them available through an application monitor API at the PaaS layer.

In this short paper, we focus the evolution of analysis phase (requirements, design and test load specification) with the limited goal of enabling the collection energy measurement data that can be used for a design time evolution and will later enable dynamically adaption scenarios. More details are available in our papers [DRP12, DP14, PD15].

2 Augmenting Cloud Requirements, Specifications and Deployment for Energy Awareness

Based on the previously described energy-aware stack, it is necessary to help developers to learn how much energy is consumed by their application on the server-side. Unlike certain performance or security characteristics already understood by users and developers, energy consumption behavior of server-side components is often completely unknown. Rare are those who could state quantifiable requirements on the energy consumption behavior on the server-side of particular features of their application. Our approach is to provide the developer with tools that will ease the following key steps to move to an energy-aware cloud application development:
At requirements level - To structure the approach, the Goal-Question-Metric (GQM) paradigm is used [BCR94]. In particular, it can be used to propose generic goals and questions that developers will often want answers to in order to gain a more precise knowledge of the energy consumed by various features or components of their application. It also makes the link with a number of already identified energy-related metrics [BGL13].

At design level - To capture the information on how to measure energy consumption of a feature or component in a way that follows the traditional modelling approach used by analysts and eases further processing, the design model language must be augmented with annotations to connect design element to energy requirements (goals and questions). This will enable the automated deployment of measurement probes to monitor the specified KPI and report them in terms of the questions and goals of the GQM identified at requirements level.

At runtime level - Probes collect the specified data and report them to a monitoring infrastructure part of the energy-aware Cloud stack. This monitoring itself is efficient in terms of data collection strategy. Application monitoring occurs at the SaaS level but relies on data from the lower PaaS and IaaS layers, for example, for collecting Watt-hour of a blade or CPU percentage time of a process running in a virtual machine (VM).

For easing adoption by developers, it is also very important to propose a practical tool that will seamlessly integrate with current development habits and mainstream development environments. In this respect the Unified Modelling Language (UML) is now universally known by developers and supported by development environments [OMG97]. Standard extensions mechanisms, based on stereotypes, are available to enrich the existing diagramming notations, in our case with energy-related information. The profile can be "plugged" to simply evolve an existing application model. If no model exists, a simplified energy-oriented model can be retro-engineered. It will naturally help to capture all the relevant energy-related elements and in a possible later step be used as a basis to drive the application refactoring.

3 Overview of the Implementation and Validation

Our reference implementation was developed on Papyrus, an Open Source Eclipse-based UML tool. Papyrus supports the definition of profiles through .profile projects that can be specified with the tool itself. They can then be applied to normal UML projects which will then benefit from the specified extension. Papyrus automatically generates all the input forms required to captures the structured and typed information specified in the profile. Subsequently, the Acceleo query technologies was used to retrieve the energy-related information encoded in the instantiated model and to generate the deployment descriptor in OVF format. Finally, the visualisation and reporting currently rely on BIRT.

![Image](image-url)

Figure 2: Annotated Question and related KPI probe in the UML Model

The current toolset is being validated on a "News as an Asset" (NewsAsset) case study which is an end-to-end multimedia cross-channel solution for evolving News Agencies, Broadcasters and Publishers developed by Athens Technology Center (ATC) [ATC]. The global objective is to move to a Cloud-based architecture than can dynamically fit to the need of the agency based on a low entry cost and a pay-per-use model. This requires the application to be migrated to operate in the Cloud.
Figure 2 shows the Papyrus editor displaying a component diagram with the focus on a subcomponent within the NewsAsset server: the Data Reader/Writers. Two entries are visible in the property tab: the KPI list and the technology specifications. A single KPI was defined in our case and is displayed on the dialog on the left part of the figure. It relates to a specific question and the corresponding probe is PerfView which is referenced as a Chef cookbook. It will automatically deploy the probe and configure it based on the parameters specified in the KPIDefInput, i.e. the measurement frequency and the targeted thread. Some visualisation modalities can also be specified. On the technological side, C# is specified as implementation language and will drive the selection of a compatible probe.

4 Perspectives

We are currently working in the following directions. First, we are working on fully automating the deployment of probes and to efficiently collect the relevant energy related data (sampling rate, aggregation strategy,...). Second, we will formalise some part of the profile in order to provide easy mapping on popular automation frameworks such as Chef. Third, we will be further enrich our profile with self-adaptation strategies than can preserve energy goals even when the execution context is evolving due to evolution in load, network, infrastructure costs, etc.

Acknowledgements

This work was partly funded by the European Commission under the FP7 project ASCETiC (nr 610874).

References


On the use of Java database frameworks in Java projects
– A large-scale historical empirical analysis

Mathieu Goeminne, Software Engineering Lab, UMONS, Belgium

Abstract

More and more software systems are becoming data-intensive, and require intensive access to a database. In this paper, we empirically study a large corpus of data-intensive Java projects as well as the database frameworks used in these projects to manage the connection between the source code and the database. We also study how this usage evolves over time. In particular, we study whether certain database frameworks are used simultaneously, whether some database frameworks get replaced over time by others. Using the statistical technique of survival analysis, we also study the survival of database frameworks in the Java projects in which they are used, as well as the survival of these projects themselves.

1 Introduction

Data-intensive software systems (DISS) are software systems that heavily rely on a database: many services provided by these systems require a connection to a database that ensures the persistency of manipulated data. As any software system, DISS have to evolve over time in order to accommodate new user requirements, fix bugs, increase quality, and adapt to new technologies. Several database frameworks have been developed to automate the time-consuming and error-prone task of creating SQL queries and adapting them to the changes occurring in the database schema and the software that exploits it.

In this paper, we focus on database frameworks for open source Java projects using relational database technology. We rely on open source data since we need full access to the source code development history of the studied projects. We have chosen for Java because it is one of the most popular programming language today. We restrict ourselves to relational databases, because they are omnipresent in current-day software projects and because historical data is available to analyse their usage in data-intensive software projects.

We empirically study the presence and survival of popular database frameworks in these projects. In particular, we explore to which extent: multiple database frameworks are used simultaneously; some database frameworks tend to replace other ones; the survival of projects is positively or negatively affected by the presence or absence of specific database frameworks.

2 An overview of Java Database Frameworks

We considered 20 Java database frameworks as potential candidates for our empirical study. These candidate frameworks were selected by skimming recent scientific publications, Stack Exchange and blog posts. Despite the fact that the list is therefore not exhaustive, it covers the most frequently cited frameworks.

As an additional constraint, since our goal is to study the evolution over time of the database framework usage in existing Java projects, we only consider database frameworks that are at least 3 years old. These frameworks need to have a direct means for accessing the database. Therefore, we excluded web frameworks, such as Play and JavaServer Faces, that mainly aim to facilitate the development of web applications.

As a baseline, we also included JDBC in our study. Unlike other considered frameworks, it doesn’t provide any abstraction of a database schema and forces the developers the write raw SQL queries. JDBC
is still heavily used because such a low level connection allows to submit complex queries that would be difficult or impossible to express with a higher level database framework.

We determined the presence of these frameworks by analysing the imports statements in Java files. The presence of some frameworks using specific configuration files is also detected by looking for these files.

3 Data Extraction Process

We extracted all 14,765 Java projects taken from the Github Java Corpus proposed in [1]. Among these, 13,328 (90.3%) projects still have a Git repository on 24 March 2015. These Java projects are potential candidates for our empirical analysis, so we created a local clone of each of them. Figure 1c shows how many of these projects used at least one of the considered database frameworks. We observed that the Rife and sqlo2 frameworks were only encountered once, while the Sormula and Jorm frameworks were never used. Therefore, we excluded these 4 frameworks from our empirical analysis.

After this preprocessing phase, 3,820 GitHub Java repositories remained that each used at least one database framework. For each commit of each considered repository, we identified activities related to the considered frameworks. To speed up the analysis we used a map-reduce based approach instead of creating a local copy of each of the commits stored in the repositories.

4 Research Methodology

Our empirical study focuses on the following research questions:

RQ1 Which of the considered database frameworks are most frequently used?

RQ2 How long does a database framework “survive” in the projects in which it is used?

RQ3 How many database frameworks do software projects tend to use over time?

RQ4 Does the introduction of a new database framework in a project influence the survivability of an existing one?

We use survival analysis models [5, 4] for describing the time needed for an event to occur in a considered population. If the event didn’t occur before the last observation of a project, we can not determine if, and a fortiori when, the event will occur after this observation. Survival analysis models take into account such a right-censored information and estimate the time of occurrence. More precisely, the Kaplan-Meier estimator [3] builds survival analysis models that represent the probability to observe the disappearance of a database framework in a Java project, or the death of the project itself. Kaplan-Meier curves visually represent this probability. These curves start at value 1, since all the considered individuals are, by definition, alive at the beginning of their observed life period. The curves decrease over time, and reach the value 0 if the considered event occurred in all the considered individuals during the observation period. To test a difference with statistical significance between two survival distributions we used the a Mantel-Haenszel test [6, 7]. All survival analysis results produced in this paper were obtained using the R package survival\(^1\).

5 Preliminary results

This section presents preliminary results of our study. They consist of visualisations and tests of statistical hypotheses, and provide initial evidence for our research questions.

W.r.t. RQ1, Fig. 1a shows in logarithmic scale the number of projects using the considered database frameworks. Their popularity is strongly unbalanced and only 7 frameworks have been used in more than 100 projects.

\(^1\)http://cran.r-project.org/web/packages/survival

\[^2\]
To answer RQ2, Fig. 1b shows the survival curves of database frameworks used in at least 100 projects. Survival models were computed using the Kaplan-Meier estimation. After they have been introduced, all the frameworks remain used in more than 45% of the projects. Nevertheless, we observe different trends in their survivability. For example, in 19.2% of all cases Avaje is removed two weeks after its introduction. In the same time interval, Spring is only removed from 4.8% of the projects. Three years after its introduction, Avaje disappears from 48.1% of all projects. This lack of survival compared to the other frameworks probably makes Avaje a not recommandable database framework candidate. A detailed statistical comparison of the survival curves of the considered database frameworks is part of our future work.

As a preliminary explorative visualisation for RQ3, Fig. 1c presents in logarithmic scale the number of projects in which a given number of database frameworks have been introduced. The distribution of the number of frameworks used by the 3,820 remaining projects seems to follow a power law.

All the considered Java database frameworks basically provide the same service: allowing the source code to exploit a database. Intuitively, two frameworks used in the same project are therefore in competition. In order to answer RQ4, focusing on the question if the introduction of a new framework causes the eviction of a previously used one in the same project, we used Mantel-Haenszel tests to check for a difference in the survival rates of a framework depending on the presence of other frameworks. Table 1 shows the p-values of these tests. A low value ($\leq 0.05$) reveals a significant difference. Significant test results are marked with ‘+’ if the introduction of another framework increases the survival rate, or with ‘-’ if the introduction decreases the survival rate. Indecisive cases are marked with a ‘?’.

6 Conclusion

This paper presents the preliminary results of a large scope study of Java database framework usage. We observed an important disparity in the popularity of the considered frameworks. The survival analyses revealed that a framework introduced in a project has between 45% and 65% of chances to stay in this project as long as the project remains active. However, some frameworks tend to be removed shortly after
Table 1: p-values of Mantel-Haenszel tests for differences of survivability, with and without the introduction of another framework.

<table>
<thead>
<tr>
<th>Tested introduced</th>
<th>Raw SQL</th>
<th>Spring</th>
<th>JPA</th>
<th>Vaadin-GWT</th>
<th>Hibernate</th>
<th>Wicket</th>
<th>Avaje</th>
</tr>
</thead>
<tbody>
<tr>
<td>JDBC</td>
<td>—</td>
<td>0.045*</td>
<td>0.015*</td>
<td>0.198</td>
<td>0.016*</td>
<td>0.239</td>
<td>0.006*</td>
</tr>
<tr>
<td>Spring</td>
<td>0.300</td>
<td>—</td>
<td>0.021*</td>
<td>0.678</td>
<td>0.006*</td>
<td>0.255</td>
<td>0</td>
</tr>
<tr>
<td>JPA</td>
<td>0.020*</td>
<td>0.972</td>
<td>—</td>
<td>0.339</td>
<td>0.094*</td>
<td>0.268</td>
<td>0.956</td>
</tr>
<tr>
<td>Vaadin-GWT</td>
<td>0.507</td>
<td>0.845</td>
<td>0.299</td>
<td>—</td>
<td>0.462</td>
<td>0.405</td>
<td>NA</td>
</tr>
<tr>
<td>Hibernate</td>
<td>0.110</td>
<td>0.835</td>
<td>0.913</td>
<td>0.921</td>
<td>—</td>
<td>0.832</td>
<td>0</td>
</tr>
<tr>
<td>Wicket</td>
<td>0.053*</td>
<td>0.781</td>
<td>0.489</td>
<td>0.123</td>
<td>0.800</td>
<td>—</td>
<td>NA</td>
</tr>
<tr>
<td>Avaje</td>
<td>0.713</td>
<td>0.000*</td>
<td>0.449</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>—</td>
</tr>
</tbody>
</table>

their introduction, while others have a higher chance of survival. Despite its lack of advanced services and its lower level of abstraction, JDBC is very popular and has a higher survival rate than more sophisticated frameworks such as Hibernate and Spring. At the opposite side of the spectrum, Avaje tends to be the most often and most quickly abandoned framework.

In the future, we intend to extend and improve our analyses by using Proportional Hazards regression models for statistically determining if the introduction of a second database framework leads to an improvement or a deterioration of the survival rate of a framework. We plan to study the ease and the cost of replacing a framework by another one. We intend to determine if some frameworks are frequently or seldomly used together. Finally, we intend to determine the main factors that influence the survivability of database frameworks. Among others, we would consider the project size, its purpose, the number developers involved in the project and the number of files accessing to the database as criteria for a sub-group analysis.

References


Mutation Testing: An Industrial Experiment

Ali Parsai, Quinten David Soetens, and Serge Demeyer
University of Antwerp
Middelheimlaan 1, 2020 Antwerp
ali.parsai@student.uantwerpen.be
{quinten.soetens, serge.demeyer}@uantwerpen.be

Abstract

To assess the ability of a test suite to catch faults, an accurate quality metric is needed. Mutation testing is a reliable and repeatable approach that provides such metric. However, because of its computationally intensive nature and the difficulties in applying such a technique to complex systems, it has not been widely adopted in industry. This study aims to determine if the information gathered by this method is worth the performance costs in an industrial case.

1 Introduction

One of the major goals in the software development process is quality assurance. Testing software systems is one way to reduce the chance of unintended faults during this process. It is necessary to measure the quality of the test code in order to maintain a high quality test suite, which in turn, assures sufficient testing of the software. There are simple metrics to estimate this quality, such as Branch coverage. Such metrics provide an overview of the quality of the test suite in an easily attainable manner; yet, they are inadequate in their purpose of estimating the effectiveness of the test suite in catching faults [WMO12, IH14]. Building confidence on the final product based on their results increases the probability of faults not being caught [Mar91]. Hence, there is a need for more powerful metrics to address this problem.

Mutation testing offers another metric by introducing a repeatable and scientific method of measuring the quality of the test suites. This metric is demonstrated to simulate the faults realistically [ABL05]. Because this metric relies on a fault model, it often produces more accurate results than simple coverage criteria if the model used to produce the faults is close to reality [ABLN06]. Moreover, because every mutant is instantly traceable to the part of the code it has changed, it is easy to obtain a detailed map of where the uncaught faults are. However, there are downsides of using mutation testing. One that has limited its functionality in industrial practices is its computationally intensive nature [FWH97]. The issue that we try to investigate is to answer if it is worth it to use mutation testing instead of simple metrics in the system. Most of the industries rely on branch coverage as the quality metric of choice [GZ13]; because it can be calculated quickly during the build process.

Hence, the question we try to answer is as follows:

- RQ. What are the trade-offs involved in mutation testing vs. branch coverage for determining test suite quality?

To answer this question, Agfa HealthCare Segmentation component was chosen as the target project. This software system is a fair representation of the challenges in the industry for mutation testing due to the complexity of its architecture and testing framework. Because of this, none of the available mutation testing tools could be used to perform the analysis; therefore, a new tool called LittleDarwin\(^1\) was developed for this purpose.

By answering this question, we try to gather practical evidence on the benefits and disadvantages of mutation testing in comparison to branch coverage in delivering detailed information about the quality of the test suite.

\(^1\)http://littledarwin.parsai.net
2 Background
This section describes the background information related to mutation testing. In addition, it provides an overview of the implementation of mutation testing in LittleDarwin.

2.1 Mutation Testing
Mutation testing is the process of injecting faults into software, and counting the number of intentional faults which make at least one test fail. The idea of mutation testing was first mentioned in a class paper by Lipton (reported in [OU01]) and later developed by DeMillo, Lipton and Sayward [DLS78]. This procedure is executed in the following manner: First, faulty versions of the software are created by introducing a single fault into the system (Mutation). This is done by applying a known transformation on a certain part of the code (Mutation Operator or Mutator). After generating the faulty versions of the software (Mutants), the test suite is executed on each one of these mutants. If there is an error or failure during the execution of the test suite, the mutant is regarded as killed. On the other hand, if all tests pass, it means that the mutant has survived. Mutation coverage is percentage of the killed mutants to all mutants. An overview of this procedure can be observed in Figure 1.

2.2 LittleDarwin
LittleDarwin is a mutation testing tool created by the first author. This tool is designed to offer an alternative framework for those who need to apply mutation testing to complex systems. LittleDarwin is designed to be independent from the testing structure. As a result, LittleDarwin demands much less compatibility from the target system in order to perform its analysis, and it can be run on any build structure no matter how complex it is given following conditions: The build process must be able to run the test suite, return non-zero if any tests fail, and zero if it succeeds, and be sufficiently fast in order to keep the total run time practical.

In its current version, LittleDarwin supports mutation testing of Java programs. In total, there are 9 mutation operators implemented in LittleDarwin. These mutators are collectively known as the minimal set. In 1996, Offutt et al [OLR+96] determined that this selection of mutation operators are enough to produce the same coverage information with a four-fold reduction of the number of mutants.

3 Experimental Setup
Due to the popularity of branch coverage as a quality metric in the industry, it is not sensible to assess the usefulness of mutation coverage without a comparison. During the build process, JaCoCo\(^2\) acts as a Maven\(^3\) plugin which calculates branch coverage for Segmentation component. The data for the branch coverage can be acquired by executing the build system — thus executing the test suite — once. Mutation coverage is calculated by using LittleDarwin on Segmentation component.

3.1 Comparison Criteria
The following categories are used to describe the results of this experiment:

- **Category 1.** Classes with similar branch and mutation coverage.
- **Category 2.** Classes with low branch coverage and high mutation coverage.
- **Category 3.** Classes with high branch coverage and low mutation coverage.
- **Category 4.** Classes which do not have a branch coverage but have a mutation coverage.
- **Category 5.** Classes which have a branch coverage but do not have a mutation coverage.

Considering \(m\) as mutation coverage percentage, and \(b\) as the branch coverage percentage for a class, the classes with non-zero scores with the difference of \(t\%\) or without either coverage are considered to be in Category 1 \((|m - b| \leq t \land m, b > 0) \lor m = b = 0\). Any classes with non-zero scores for both metrics that are not in Category 1, are in either Category 2 \((m - b > t \land m, b > 0)\) or Category 3 \((b - m > t \land m, b > 0)\). Those that are not part of the first three categories, are part of either Category 4 \((m > 0 \land b = 0)\) or Category 5 \((b > 0 \land m = 0)\).

\(^2\)http://www.eclemma.org/jacoco/
\(^3\)http://maven.apache.org/
The value of $t$ determines the threshold we set in the difference of the results of two metrics for a single class to consider them as similar. This threshold does not have any effect on the number of classes in Categories 4 and 5, and only changes the values in Categories 1, 2, and 3.

By counting the number of the classes in each category we can determine the usefulness of each metric for our purpose. For classes in Category 1, the information can be acquired using either metric, meaning that, at least without any deeper analysis, the difference between two metrics regarding these classes is minimal. For classes in Categories 2 and 3, both metrics should be used in conjunction to acquire the insights described above; meaning that neither metric on its own provides any extra information. Information about classes in Categories 4 and 5 are, however, only attainable by one of the metrics; therefore, showing a distinct advantage for each metric in gathering information.

### 4 Analysis of the Results

In this section, we try to answer our research question using the results of our experiment:

- **RQ.** What are the trade-offs involved in mutation testing vs. branch coverage for determining test suite quality?

In total, 12825 mutants were generated for 211 source files of Segmentation component. 4955 of these mutants were killed by the test suite, resulting in 38.6% overall mutation coverage.

In Figure 2, the horizontal axis are the classes sorted by their mutation coverage, and the vertical axis is the coverage score by percentage. The blue bars are the representation of branch coverage and the red line is the calculated mutation coverage.

As seen in Figure 2, there are a lot of classes which lack a branch coverage metric, but mutation coverage is calculated for them (Category 4). In total, 90 classes are classified as Category 4. After manual inspection of several classes, it was clear that mutation coverage is in fact providing the correct analysis, and the calculation method used by JaCoCo is at fault. The fault lies in execution traces being lost during test execution when the involved classes use external resources. The complicated structure of the system is partially to blame for this phenomenon; however, such complexities are unavoidable in many industrial cases. This lack of information about existing test coverage is crucial in preparing a plan to improve the quality of the test suite. This information allows major efforts to be focused on the parts of the test suite which is in need of service, rather than wasting time on developing redundant tests.

More interesting cases could be found in the areas where mutation coverage was low and branch coverage was high (Category 3). In one example, branch coverage was 88% and mutation coverage was 41%. This shows that the tests covering this class are of poor quality, and even though they cover most of the control structures, they are unable to catch the potential bugs.

In Table 1, a summary of the results of the experiment is provided. A total number of 90 classes out of 212 were classified as Category 4, while only 4 classes were classified as Category 5. The high number of classes in category 4 suggests that the results of this experiment support the fact that mutation testing can provide...
Table 1: Summary of experiment results

<table>
<thead>
<tr>
<th>Experiment</th>
<th>Total #</th>
<th>Cat.1</th>
<th>Cat.2</th>
<th>Cat.3</th>
<th>Cat.4</th>
<th>Cat.5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Segmentation</td>
<td>212</td>
<td>102</td>
<td>8</td>
<td>8</td>
<td>90</td>
<td>4</td>
</tr>
</tbody>
</table>

extra information which branch coverage does not provide. However, the test suite for Segmentation component includes lots of black-box unit tests which are made according to the requirements of the software. These tests only consider the final results of the algorithms they test. There were 8 classes that were classified as Category 2, and 8 that were Category 3. Such a low number of classes in these two categories suggests that the classes with high mutability areas are either tested throughly, or not being tested at all. The rest of the classes (102) were classified as Category 1, for which the mutation coverage and branch coverage calculated roughly the same values. These results show that mutation testing provides valuable extra information at the cost of longer computation time, making it a suitable analysis method for quality-critical systems with relaxed time constraints.

5 Conclusion and Future Work

To answer our research question, we compared the results from LittleDarwin to branch coverage information produced by JaCoCo. Mutation testing revealed test coverage for parts of Segmentation component that remained undetected by JaCoCo. Based on these results, we can argue that using mutation testing can reduce the workload in refactoring and developing the test suite by pinpointing where the weaknesses are, and where there is enough coverage already; making it a viable choice for long-term strategic planning. However, to replace branch coverage as a quality metric, mutation coverage needs to be calculated in comparable time frames.

To improve the performance, it is possible to use incremental mutation testing in a continuous integration framework, effectively replacing simple metrics such as branch coverage. It is also possible to use distributed computing techniques to run the analysis in parallel, thus reducing the time needed to complete the operation.

References


