Decision Modules in Models and Implementations

(Co-)Evolution in MDSE ecosystems

J.G.M. Mengerink
Eindhoven University of Technology, The Netherlands
Email: j.g.m.mengerink@tue.nl

INTRODUCTION

In model driven software engineering (MDSE) [10], model-transformations are central artifacts [14]. They depend on meta-models for their structure and relate the different models in the ecosystem. However, meta-models evolve, for instance because of new insights in the systems they model. A pressing issue in industry, is that maintaining model-transformations with respect to meta-model evolution is very costly [3] in both a time-related and skill-related sense. To this end, it is desirable to automate this co-evolution of transformations, with respect to meta-model evolution, to the furthest extent possible. Although for meta-model/model co-evolution, a variety of tools exist [15], for meta-model/model-transformation co-evolution, most tools remain in prototype [2], [4], [12]. The methods and techniques of these prototypes are promising. However, the prototypes are all aimed towards specific use-cases and only offer support that is sufficient for their specific use-cases. When one requires to evolve artifacts that are not in-line with the artifacts in those case-studies, these prototype are not yet mature enough.

In this extended abstract we sketch the envisioned direction of the PhD research addressing the (co-)evolution challenge in MDSE ecosystems. The research is to be conducted in 2014–2018.

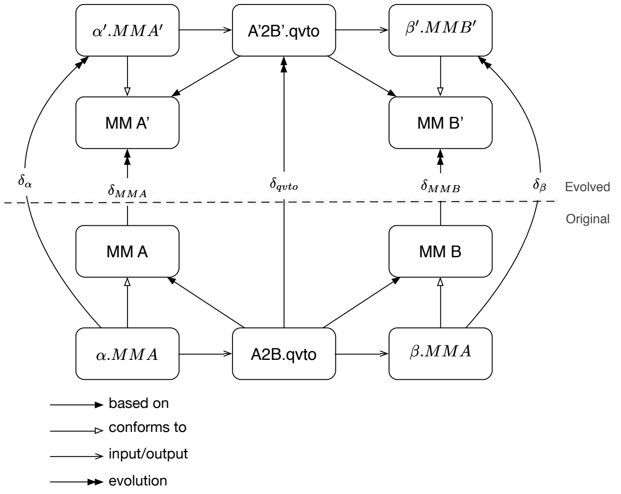


Fig. 1: Abstract representation of evolution in an MDSE ecosystem, extended form the non-evolutionary variant in [8]

INDUSTRIAL CONTEXT

Our research takes place at ASML, the leading provider of complex lithography systems. Here we have access to an industrial repository containing a large MDSE ecosystem with version history going back up to three years. Our ecosystem can be represented similarly to that of Jouault and Kurtev [8]. However, we are more interested in the evolutionary axis through such a system, as is illustrated in Figure 1. As in the non-evolution version [8], our representation shows two models (α.MMA and β.MMB) relating to meta-model MMA and MMB respectively. To incorporate evolution, we include the evolved versions of MMA and MMB (MMA′ and MMB′ respectively), to which evolved models α'.MMA′ and β'.MMB′ conform. Lastly, our model-transformation A2B.qvto should co-evolve to support the new models, leading to A′2B′.qvto.

RESEARCH QUESTION

The main question that we aim to solve is how to specify the differences between difference versions of our modeling artifacts (meta-models, models, and model-transformations). That is: in what way can we specify, for example, $\delta_{\texttt{MMA}}$, such that we have enough information to co-evolve the related models and model-transformations. This specification can take place either before, or after evolution of the primary artifacts (i.e. the meta-models). If one was to provide such a specification a-priori, it could be used to perform evolution on both the primary, and the secondary artifacts (i.e. the model-transformations). Alternatively, this specification could be created after evolution of he primary artifacts (potentially in an automated way), and used solely for the evolution of secondary artifacts.

RELATED WORK

In literature, a number of different approaches into specifying evolution have been addressed. ***State-based*** approaches attempt to calculate the difference between two versions of a meta-model ($\delta_{\texttt{MMA}}$), then adapt the related artifacts (A2B.qvto and α.MMA). Often, these approaches attempt to aggregate smaller changes into higher order transformations (HOTs). [1], [5], [17]

Generation approaches aim to fully generate model-transformation, rather than evolving them from previous versions. By-example techniques can be employed, letting the user specify relations between model instances (i.e. between α'.MMA′ and β'.MMB′) [9]. Using this information, A′2B′.qvto is generated, rather than evolved from A2B.qvto. Other approaches include regenerating from a shared ontology of concepts [16].

Operator-based approaches define a set of operators which the developer can use. These operators affect both the meta-model and artifacts, while preserving conformance during the evolution. Rather than compute $\delta_{\mathtt{MMA}}$, the user creates it by the successive applications of these operators. While an extensive set of these operators exists for model co-evolution [6], only a very restricted set is available for transformation co-evolution [11].

An example of an operator-based language is one by Luo [13]. However, it focuses on refinement, only allows for additive changes, and does not consider subtractive changes [7] (i.e. removal of elements). Furthermore, this approach specifies change at a fine-grained level of detail. To effectively co-evolve artifacts, it is desirable that changes are specified at a higher, more coarse, level. For example, specifying change in terms of adding and deleting model elements, provides little information about the intent of the user. However, if one were to specify change in terms of higher-order operations such as `Extract Superclass` or `Flatten Hierarchy`, additional information with respect to the evolution process can be obtained (i.e. to what end is the user adding/removing a certain element?). Using this additional information, artifacts can be co-evolved more precisely, such that the result is closer to the end-result desired by the user.

In order to extend such a language with subtractive and updative (e.g. renaming an element) operations [7], the different operations (either low-level or high-level) need to be categorized with respect to the context in which they operate. For instance, extending a meta-model with an optional element, does not require conforming models to be update, so $\alpha.\mathtt{MMA} = \alpha'.\mathtt{MMA}'$. We wonder whether we can discover, and use these properties to facilitate co-evolution.

Envisioned Approach

Given the large amount of available work for operator-based (co-)evolution of models [6], we feel research in to operator-based (co-)evoltion of model-transformations will be the most fruitful. The first aim of our study will be to increase the available operators for model-transformations, by looking at the available operators of models. In this way, we aim to specify the difference between two meta-model versions in terms of these operators. An added benefit to this approach is that such a sequence of operators should immediately give us a specification for co-evolution of model-transformations. However, rather than creating these operators in just a, traditional, bottom-up fashion, additionally we will attempt use extract operators from the ASML repositories. Secondly, our research will focus on semi-automatic reconstruction of operator-sequences from a difference specification between meta-model versions. The latter should close the gap between state-based and operator-based approaches.

References

[1] Antonio Cicchetti, Davide Di Ruscio, Romina Eramo, and Alfonso Pierantonio. Automating Co-evolution in Model-Driven Engineering. *2008 12th International IEEE Enterprise Distributed Object Computing Conference*, pages 222–231, September 2008.

[2] Juri Di Rocco, Ludovico Iovino, and Alfonso Pierantonio. Bridging state-based differencing and co-evolution. In *Proceedings of the 6th International Workshop on Models and Evolution*, ME '12, pages 15–20, New York, NY, USA, 2012. ACM.

[3] Marcos Didonet Del Fabro and Patrick Valduriez. Towards the efficient development of model transformations using model weaving and matching transformations. *Software & Systems Modeling*, 8(3):305–324, 2009.

[4] Jokin Garcia, Oscar Diaz, and Maider Azanza. Model transformation co-evolution: A semi-automatic approach. In Krzysztof Czarnecki and Gorel Hedin, editors, *SLE*, volume 7745 of *LNCS*, pages 144–163. Springer, 2013.

[5] Boris Gruschko, Dimitrios Kolovos, and Richard Paige. Towards synchronizing models with evolving metamodels. In *Proceedings of the International Workshop on Model-Driven Software Evolution*, 2007.

[6] Markus Herrmannsdoerfer, Sander D. Vermolen, and Guido Wachsmuth. An extensive catalog of operators for the coupled evolution of metamodels and models. In *SLE*, pages 163–182. Springer, 2011.

[7] Ludovico Iovino. *Coupled Evolution in metamodeling ecosystems*. PhD thesis, Unversita di Laquila, Via Vetoio, I-67100 LAquila, Italy, April 2013.

[8] Frédéric Jouault and Ivan Kurtev. On the architectural alignment of atl and qvt. In *Proceedings of the 2006 ACM Symposium on Applied Computing*, SAC '06, pages 1188–1195, New York, NY, USA, 2006. ACM.

[9] Gerti Kappel, Philip Langer, Werner Retschitzegger, Wieland Schwinger, and Manuel Wimmer. *Model Transformation By-Example: A Survey of the First Wave*, volume 7260 of *LNCS*, pages 197–215. Springer, 2012.

[10] Stuart Kent. Model driven engineering. In Michael Butler, Luigia Petre, and Kaisa Sere, editors, *Integrated Formal Methods*, volume 2335 of *LNCS*, pages 286–298. Springer, 2002.

[11] Steffen Kruse. On the use of operators for the co-evolution of metamodels and transformations. In *International Workshop on Models and Evolution*, 2011.

[12] Tihamer Levendovszky, Daniel Balasubramanian, Anantha Narayanan, and Gabor Karsai. A novel approach to semi-automated evolution of dsml model transformation. In Mark van den Brand, Dragan Gasevic, and Jeff Gray, editors, *SLE*, volume 5969 of *LNCS*, pages 23–41. Springer, 2010.

[13] Yaping Luo, Mark van den Brand, Luc Engelen, and Martijn Klabbers. From conceptual models to safety assurance. In *Conceptual Modeling*, pages 195–208. Springer, 2014.

[14] Tom Mens and Pieter Van Gorp. A taxonomy of model transformation. *Electronic Notes in Theoretical Computer Science*, 152(1-2):125–142, March 2006.

[15] Louis M. Rose, Markus Herrmannsdoerfer, James R. Williams, Dimitrios S. Kolovos, Kelly Garces, Richard F. Paige, and Fiona A.C. Polack. A comparison of model migration tools. In *Model Driven Engineering Languages and Systems*, pages 61–75. Springer, 2010.

[16] Stephan Roser and Bernhard Bauer. Automatic generation and evolution of model transformations using ontology engineering space. In Stefano Spaccapietra, JeffZ. Pan, Philippe Thiran, Terry Halpin, Steffen Staab, Vojtech Svatek, Pavel Shvaiko, and John Roddick, editors, *Journal on Data Semantics XI*, volume 5383 of *LNCS*, pages 32–64. Springer, 2008.

[17] Guido Wachsmuth. *Metamodel Adaptation and Model Co-adaptation*, volume 4609 of *LNCS*, pages 600–624. Springer, 2007.

Evolving Languages with Object Algebras

Pablo Inostroza
Centrum Wiskunde & Informatica (CWI)
Amsterdam, The Netherlands
Email: pvaldera@cwi.nl

Tijs van der Storm
Centrum Wiskunde & Informatica (CWI)
Amsterdam, The Netherlands
Email: storm@cwi.nl

***Abstract*—Object Algebras are a programming technique for the extensible implementation of recursive data types. This extended abstract introduces Object Algebras and shows how they could be used to develop highly evolvable software languages. The paper is concluded with a discussin of directions for further research.**

I. Introduction

Object Algebras [4] are a solution to the expression problem [7]. This means they support the extension of a data type along two dimensions: data type variants and the operations over the data type. Since the abstract syntax of a language is naturally described using recursive data types, this suggests that Object Algebras are a viable technique for implementing extensible language implementations.

Using Object Algebras, the abstract syntax of a language is described using generic factory interfaces. The following interface describes a simple language for expressions:

```
interface ExpAlg<E> {
  E lit(int n);
  E add(E l, E r);
}
```

Operations over the abstract syntax are represented by implementations of such generic interfaces, where the type parameter (e.g., `E`) is bound to the type of the operation. For instance, evaluation of expressions can be realized as follows (using Java 8 closures):

```
interface IEval { int eval(); }

class Eval implements ExpAlg<IEval> {
  IEval lit(int n) { return () -> n; }
  IEval add(IEval l, IEval r) {
    return () -> l.eval() + r.eval();
  }
}
```

The functional interface `IEval` captures the type of the operation we're defining. The class `Eval` is a factory for interpreters of expressions.

To evaluate an expression it should be created using the `Eval` factory. As an example, the following generic method creates the expression "1 + 2" over any algebra `alg`:

```
<X> X make(ExpAlg<X> alg) {
  return alg.add(alg.lit(1), alg.lit(2));
}
```

To create evaluable expressions, one would call this method with an instance of `Eval`.

II. Adding Syntax

Any language operation is realized by (re)implementing the generic factory interface. To add another language construct, however, we need to extend the generic factory interface itself first. For instance, the following interface could represent the extension with multiplicative expressions:

```
interface MulAlg<E> extends ExpAlg<E> {
  E mul(E l, E r);
}
```

Existing operations can then be extended by implementing the extended interface and subclassing the class representing the base operation. For instance, to extend the evaluator of expressions to support multiplication, one would write the following class:

```
class EvalMul extends Eval implements MulAlg<IEval> {
  IEval mul(IEval l, IEval r) {
    return () -> l.eval() * r.eval();
  }
}
```

Expressions should now be created over the extended interface `IMulAlg` using the factory `EvalMul`.

III. Changing Semantics

Sometimes we do not want to add a new language construct, but change the semantics of an existing construct, for instance, to fix a bug. This can be achieved using plain inheritance.

Consider the contrived example of changing the behavior of `add` to perform subtraction instead of addition. This can be expressed by overriding the constructor method `add` of `EvalMul`:

```
class SubIsTheNewAdd extends EvalMul {
  IEval add(IEval l, IEval r) {
    return () -> l.eval() - r.eval();
  }
}
```

IV. Advice

If it's not needed to completely replace the semantics of a construct, we can inherit from an operation and call **`super`** to selectively add "advice" to language constructs, as a simple form of Aspect-Oriented Programming (AOP) [3]. .

For instance, let's say we decide that the `add` construct is deprecated. In this case we want to keep the original behavior[1] but issue warning message to the user:

[1] "Extend and deprecate" is a common language evolution pattern.

```
class DeprecateAdd extends SubIsTheNewAdd {
  IEval add(IEval l, IEval r) {
    System.err.println("WARNING: + is deprecated");
    return super.add(l, r);
  }
}
```

Note that it is equally possible to additionally wrap `l` and `r`, and capture the result of calling `super`.

V. Desugaring

A common strategy to extend a language with a new construct is by "desugaring" it to a combination of existing constructs. Object Algebras in combination with Java 8 default methods provide a natural way of specifying such extensions.

Consider the addition of unary negative expressions `-x`. Semantically, this is equivalent to `-1 * x`:

```
interface NegAlg<E> extends MulAlg<E> {
  default E neg(E e) {
    return mul(lit(-1), e);
  }
}
```

The default method provides a default implementation of the `neg` constructor. This works for every operation implemented over `NegAlg` (i.e., for every binding of `E`). If the desugaring is undesired, for instance when pretty printing, `neg` can always be overridden in the concrete implementation of the operation.

VI. Concrete Syntax

The extension examples shown above all involved the abstract syntax of a language as modeled by generic factory interfaces. In a realistic language implementation, however, the concrete syntax should be accounted for as well. A pragmatic solution to this problem was presented in [2]. This solution is based on using Java annotations on factory methods to specify syntax productions. Using reflection all productions can be collected and used to generate a parser for a concrete parser generator (e.g., ANTLR4).

The syntax for the basic expression language is then specified as follows:

```
interface ExpAlg<E> {
  @Syntax("exp = NUM");
  E lit(int n);

  @Syntax("exp = exp '+' exp") @Level(10)
  E add(E l, E r);
}
```

The `@Level` annotation specifies the expression's precedence level. This information can also be used to guide pretty printing.

VII. Discussion

This extended abstract introduced Object Algebras as a technique for evolving language implementations. However, Object Algebras are very young; there's not much experience yet on how to apply them in realistic case studies (but see [2]). In this section we discuss ongoing research and sketch out directions for future work.

a) Morphing Operations: Object Algebras support the extension of an operation to accomodate a new language constructs. However, it seems impossible to change the type of operations themselves. For instance, we cannot change the return type or parameter list of `eval` in `IEval`. Such changes would require reimplementing the evaluator for all cases. Additional parameters can be sometimes simulated using side effects in fields of in the concrete algebras. However, this might require maintaining a stack to simulate parameter passing. This is both tedious and error-prone. This problem is particular relevant when a language is extended with additional effects (e.g., state, continuations, backtracking, etc.).

b) Program Analysis: Object Algebras map syntactic constructs to denotations (objects) that represent the desired semantics. Often the result is simply an interpreter. Further research is needed how to specify complex program analyses (e.g., name resolution, type inference, data flow analysis, etc) as Object Algebras in an extensible way. Recent work on the relation between Object Algebras and attribute grammars also shows promise in this regard [6]. Another approach to investigate is the framework of abstract interpretation [1]. In this case a the syntactic constructs are not mapped to a concrete semantics, but to an abstract semantics (e.g., representing types).

c) Cross-cutting Concerns: Language implementation is full of cross-cutting concerns. Examples are: tracing, profiling, unique identities for name analysis, origin tracking for error messages, inserting hooks for debuggers. First steps towards generic advice in the context of object algebras is presented in [5]. Dynamic proxies are expected to be very valuable here, since they allow us to implement any operation interface (e.g., `IEval`) dynamically. However, their applicability is hampered if the signature of the operations needs to change (cf. previous point).

VIII. Conclusion

Object Algebras show a lot of promise for the implementation of evolvable languages. However, more research is needed to make their essential ingredients more modular and composable. Finally, real-life case studies are required to go beyond the stage of simple expression-oriented languages.

References

[1] P. Cousot. Abstract interpretation. *ACM Comput. Surv.*, 28(2):324–328, 1996.

[2] M. Gouseti, C. Peters, and T. van der Storm. Extensible language implementation with Object Algebras (short paper). In *GPCE'14*, pages 25–28. ACM, 2014.

[3] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. *Aspect-oriented programming*. Springer, 1997.

[4] B. C. d. S. Oliveira and W. R. Cook. Extensibility for the masses: practical extensibility with Object Algebras. In *ECOOP'12*, pages 2–27. Springer, 2012.

[5] B. C. d. S. Oliveira, T. van der Storm, A. Loh, and W. R. Cook. Feature-oriented programming with object algebras. In *ECOOP'13*, pages 27–51. Springer, 2013.

[6] T. Rendel, J. I. Brachthäuser, and K. Ostermann. From Object Algebras to Attribute Grammars. In *OOPSLA'14*, pages 377–395. ACM, 2014.

[7] P. Wadler. The expression problem. Online, November 1998. http://www.daimi.au.dk/~madst/tool/papers/expression.txt.

A recommendation system for generalizing and refining source code templates

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

Code templates are ubiquitous among source code search and transformation tools. There, the matches for a template are source code fragments of interest to the tool. Despite their ubiquity, code templates can prove difficult to specify. A template that is too relaxed will have unwanted code among its matches. Conversely, a template that is too strict will not match some of the wanted code. The process of generalizing and refining templates until they match the correct code has been the source of many headaches for the users of our own *Ekeko/X* [1] program transformation tool.

To support our users in specifying *Ekeko/X* templates, we introduce a suite of template mutation operators. While some operators change a single component of the template, others change multiple components of the template in a systematic manner. Operator *introduce-variable* is an example of the former. It changes one component of the template into a placeholder for a component of the match. Operator *generalize-aliases*, on the other hand, is an example of the latter. It introduces placeholders for all references to the same variable within a code template. In addition, it requires their matches to alias at run-time.

Using this suite of template mutation operators, we formulate a genetic search algorithm [2] that recommends modifications of an *Ekeko/X* template such that the template matches all of a given set of desired and none of a given set of undesired code fragments. We evaluate our algorithm on the problem of assisting users in specifying a code template that matches similar, but non-identical source code fragments.

[1] C. De Roover and K. Inoue. The Ekeko/X program transformation tool. International Working Conference on Source Code Analysis and Manipulation (SCAM14), 2014.

[2] M. Harman, S. A. Mansouri, and Y. Zhang. Search-based software engineering: Trends, techniques and applications. ACM Computing Surveys, 45(1), Dec. 2012.

A Longitudinal Study of an Industrial FLOSS project using Social Network Analysis*

Gregorio Robles*, José Texeira†, Jesus M. Gonzalez-Barahona*‡

* GSyC/LibreSoft, Universidad Rey Juan Carlos (Madrid, Spain) † School of Economics (TSE), University of Turku (Finland) ‡ Bitergia S.L. (Madrid, Spain)

1 Introduction

Free/Libre/Open Source projects were originally developed by a community of volunteers. In the late 1990s, companies began to collaborate with those communities, some of them by contributing to existing projects (Linux is the best example) or by releasing new ones (such as IBM with Eclipse, or once SUN with OpenOffice.org, etc.). *Hybrid* development communities, where community and industry collaborated, became frequent [3]. In recent years, a new type of FLOSS projects has emerged, driven by several companies who employ most, if not all of the *core* developers and with a very small amount of contributions from volunteers. We call this type of projects *industrial* FLOSS projects. This is the case for instance of WebKit (a web browser layout engine developed in cooperation by Apple, Nokia, Google, Samsung, Intel, RIM among others) or OpenStack (a cloud infrastructure project joint-developed by over one hundred companies). In this scenario, traditional Social Network Analysis (SNA) performed on developers [4, 2] can be augmented with information on the affiliation of the developers in order to address a range of new questions, such as:

- Are developers affiliated with different firms collaborating with each other in the project? How does the collaboration evolve over time? How does the collaboration get affected by exogenous events in the market?
- How do developers cluster into different groups? Do developer clusters correspond to firms?
- Do firms that compete in the same revenue model collaborate less in the ecosystem?
- Is the development process fair? Do contributions get included in the source code base because of their quality or is the process easier depending

*This research has been funded by the Region of Madrid under the project "Investigación y desarrollo de tecnologías educativas en la Comunidad de Madrid" (S2013/ICE-2715).

on the affiliation of the developer? Are supervision and decision processes (such as reviewing, code owning, etc.) neutral in regard to the affiliation of the developers?

2 Current state of research

We have performed a longitudinal study covering more than fours years of the OpenStack project, considering the time, pace and sequence in the study of an ecosystem [1]. We therefore extract commit activity from the versioning system repositories and interfere that two developers have collaborated if they have modified at least a file in common during a certain timespan. Our results point out that combining qualitative ethnographic material with social network analysis techniques gained by means of mining software repositories, we obtain rich longitudinal descriptions that allow to understand the competitive and collaborative aspects that are present simultaneously in such type of large and complex software ecosystems. By calculating collaborative network properties, we found out the hyper-collaborative nature of OpenStack with a surprising low degree of homophily, meaning that developers do not tend to work with developers from their own companies, in code collaboration. On the other hand, we have found that subcommunities in the OpenStack project are very heterogeneous, as they tend to include developers from many different firms. We plan to extend our study with data from the reviewing process, which is of significant importance in the OpenStack project.

References

[1] Rahul C. Basole. Visualization of interfirm relations in a converging mobile ecosystem. *Journal of Information Technology*, 24(2):144–159, 2009.

[2] Christian Bird, David Pattison, Raissa D'Souza, Vladimir Filkov, and Premkumar Devanbu. Latent social structure in open source projects. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, pages 24–35. ACM, 2008.

[3] Brian Fitzgerald. The transformation of open source software. *Mis Quarterly*, pages 587–598, 2006.

[4] Luis López, Jesús M. González-Barahona, and Gregorio Robles. Applying social network analysis to the information in CVS repositories. In *Proceedings of the International Workshop on Mining Software Repositories*, pages 101–105, Edinburg, UK, 2004.

[5] Jose Teixeira and Tingting Lin. Collaboration in the open-source arena: the webkit case. In *Proceedings of the 52nd ACM conference on Computers and people research*, pages 121–129. ACM, 2014.

An Empirical Study into Social Success Factors for Agile Software Development

extended abstract

Evelyn van Kelle*†, Aske Plaat‡, Per van der Wijst†, and Joost Visser*§
*Software Improvement Group, Amsterdam, The Netherlands
e.vankelle@sig.eu, j.visser@sig.eu
†Tilburg University, Tilburg, The Netherlands
Per.vanderWijst@uvt.nl
‡Leiden University, Leiden, The Netherlands
a.plaat@liacs.leidenuniv.nl
§Radboud University Nijmegen, Nijmegen, The Netherlands

***Abstract*—Though many warn that Agile at large scale is problematic or at least more challenging than in smaller projects, Agile software development seems to become the norm, also for large and complex projects.**

Based on literature, we constructed a conceptual model of social factors that may be of influence on the success of software development projects in general, and of Agile projects in particular. We also included project size as a candidate factor.

We tested the model on a set of 40 projects from 19 organisations, comprising a total of 141 project members, Scrum Masters, and product owners.

We found that project size does not determine Agile project success. Rather, value congruence, degree of adoption of agile practices, and transformational leadership proved to be the most important predictors for Agile project success.

I. Background

Agile Software Development methods are originally applied by, and considered successful for, small teams and projects, and scaling up these methods is challenging [1]–[4]. However, larger organizations are also facing the challenges that Agile methodologies address [1]. Since most projects do not fail due to technology, but due to social and organizational problems, and a lack of effective communication [5], it is important to gain understanding about which social factors are of significant influence on Agile project success. Specifically, we are also interested in project success at larger scale.

II. Goals and Methods

The aim of our study was (1) to independently verify earlier identified success factors; and (2) to develop and validate a new, more comprehensive conceptual model by examining relationships between various candidate success factors and Agile project success. Hypotheses regarding these relationships were tested using data from 141 team members, Scrum Masters and product owners from 40 projects from 19 Dutch organizations. A conceptual model was developed based on existing literature and on explorative interviews that were held with practitioners involved in successful (large) Agile development projects. The model includes five candidate success factors: (1) transformational leadership; (2) communication style; (3) value congruence; (4) degree of agility; and (5) project size. Subsequently, this conceptual model was empirically tested and refined. Full details in the study can be found elsewhere [6].

III. Results

Results from regression- and mediation analyses showed that value congruence, degree of agility and transformational leadership were the most important predictors for project success in this model. Value congruence was a mediating factor between candidate success factors and project success. Project size was not found to influence project success, suggesting Agile methodologies could be applied successfully on larger scale as long as there is high value congruence, high degree of agility and transformational leadership.

IV. Discussion

This study contributes to the empirical identification of (new) communication-related success factors in Agile Software Development, by providing a validated conceptual model. The model provides insights into which social factors contribute to Agile project success. We also find that project size does not play a role. This implies that the focus of managers should be on increasing value congruence, agility and transformational leadership. The result that Agile methods can indeed work for large project, is a surprising outcome, since Agile puts so much emphasis on small teams and short sprints. More research is needed to verify and analyse our findings. Future research should be conducted on a larger scale, over a longer period of time in order to validate the model (in other domains).

Acknowledgements

The authors thank Martijn Goudbeek (Tilburg University) for assistance with the statistical data analysis.

References

[1] L. Cao, K. Mohan, P. Xu, and B. Ramesh, "How extreme does extreme programming have to be? adapting xp practices to large-scale projects," in *System Sciences, 2004. Proceedings of the 37th Annual Hawaii International Conference on.* IEEE, 2004, pp. 10–pp.

[2] K. Beck, *Extreme programming explained: embrace change*. Addison-Wesley Professional, 2000.

[3] D. J. Reifer, F. Maurer, and H. Erdogmus, “Scaling agile methods,” *Software, IEEE*, vol. 20, no. 4, pp. 12–14, 2003.

[4] B. Boehm, “Get ready for agile methods, with care,” *Computer*, vol. 35, no. 1, pp. 64–69, 2002.

[5] J. Eckstein, *Agile software development in the large: Diving into the deep*. Addison-Wesley, 2013.

[6] E. v. Kelle, “Social factors of agile development success,” Master’s thesis, Tilburg University, the Netherlands, 2014, to appear.

Pricing via Functional Size – A Case Study of 77 Outsourced Projects

Hennie Huijgens, TU Delft

This story is about a company that experiences two problems in its software engineering outsourcing. First, a worsening trend is seen in productivity, indicating that the organization does not learn from historic projects. Second, much time and energy is spent on preparation and review of fixed price project proposals. Our case study explores whether a new project pricing method helps to solve these problems.

To arrive at a price that is acceptable for both parties involved, most companies rely heavily on expert judgment; where the advice of knowledgeable staff is solicited. Usually this is performed as a bottom up approach, where component tasks are identified and sized and then these individual estimates are aggregated to produce an overall estimate.

Yet, in practice effort and/or schedule overruns are business-as-usual, despite involvement of experts. Software development is characterized by high cost and schedule overruns. Estimation errors are reported to be essential causes of poor management, due to lack of a solid baseline of size.

An alternative method for software project estimation is based on algorithmic cost models (COCOMO 2 is a well-known example) which take cost drivers representing certain characteristics of the target system and the implementation environment and use them to predict estimated effort. In many of these statistic approaches size is assumed to be a key factor to estimate project cost. Usually size of software engineering projects is measured with a formal Functional Size Measurement (FSM) standard. FSM is a method to measure size of software engineering activities by means of the functionality delivered to users, which lays the foundation for a statistical method of project pricing based on functional size.

Advantages of such a statistical method are that this will help to improve transparency of estimations and that it can be a good instrument to create continuous improvement of project performance.

However, our observation is that a purely statistic method is almost never used. If statistical analysis is used, this is usually supplementary to an expert judgment-based approach. And practice shows that in most cases the expert opinion – in many cases supported by reasoning by analogy – is leading when it comes to decision making.

The goal for our research is to answer the question whether a purely statistical approach to pricing is effective in an outsourcing context. We define an approach to be effective when a so-called win-win situation is achieved: meaning that both involved parties are satisfied. The supplier delivers a service for a price that is higher than the cost, and the customer gets higher value than the paid price. In addition to that the outsourcing context asks for a long-term (5 year) relation.

Based on this we focus on transparency as the factor we need to measure to determine success: transparency when setting the price and transparency when finalized. Transparency is important for ‘next’ projects; when pricing of actual projects is transparent, this can be re-used in future projects.

A long-term relation asks, in an outsourcing context, for in-tent of continuous improvement. When a supplier becomes more efficient and effective, the price can go down without a negative effect on the supplier's margin of profit. More value for the same amount of money represents a win-win situation for both the customer and the supplier.

For this purpose we define three research questions:

RQ1: To what extent are both parties involved in an out-sourcing contract satisfied with FSM-pricing?
RQ2: To what extent does FSM-pricing help to improve transparency of project proposals?
RQ3: To what extent does FSM-pricing help to create continuous improvement?

We studied the implementation of single FSM-based pricing of supplier proposals in a Belgian telecom company that outsourced its IT to an Indian supplier. In order to create more transparency in the supplier proposal process a pilot was started on Functional Size Measurement pricing (FSM-pricing). In our research we evaluate the implementation of FSM-pricing in the software engineering domain of the company, as an instrument useful in the context of software management and supplier proposal pricing.

For this purpose we use a mixed methods methodology, as we are examining a phenomenon with multiple (qualitative and quantitative) tools. We perform a single-case, holistic case study that involves two instruments; a survey consisting of open and closed questions, and a quantitative analysis of actual project data. One of the instruments we use for the analysis is a so-called cost/duration matrix, a model that is intended for analysis and benchmark purposes. The model is based on classification of all software projects in a repository into four quadrants:

1. Cost over Time; projects that performed better than average with regard to cost, but worse than average with regard to duration;
2. Good Practice; projects that performed better than average with regard to both cost and duration;
3. Time over Cost; projects that performed better than average with regard to duration and worse than average with regard to cost;
4. Bad Practice; projects that performed worse than average with regard to both cost and duration.

We found that a statistical, empirical, evidence-based pricing approach for software engineering, as a single instrument (without a connection with expert judgment), can be used in distributed environments to create cost transparency and performance management of software project portfolios.

This abstract is based on a paper 'Pricing via Functional Size – A Case Study of 77 Outsourced Projects' by Hennie Huijgens, Georgios Gousios and Arie van Deursen (TU Delft) that is submitted to ICSE 2015 and partly on a paper 'How to Build a Good Practice Software Project Portfolio' by Hennie Huijgens, Rini van Solingen en Arie van Deursen that was accepted for ICSE (SEIP) 2014.

Hennie Huijgens works as a software economics expert for Dutch and Belgium banks and telecom companies. Besides his work he is a PhD candidate (subject: evidence-based software portfolio management) with Arie van Deursen en Rini van Solingen at TU Delft.

Continuous integration in GitHub: Experiences with Travis-CI

Bogdan Vasilescu, Stef van Schuylenburg, Jules Wulms, Alexander Serebrenik, Mark G. J. van den Brand
Eindhoven University of Technology, Den Dolech 2, P.O. Box 513, 5600 MB Eindhoven, The Netherlands
{*b.n.vasilescu, a.serebrenik, m.g.j.v.d.brand*}*@tue.nl*, {*s.b.v.schuylenburg, j.j.h.m.wulms*}*@student.tue.nl*

I. Introduction

Continuous integration (CI) is a software engineering practice of frequently merging all developer working copies with a shared main branch [1], e.g., several times a day, or with every commit. This continuous application of quality control checks aims to speed up the development process and to ultimately improve software quality, by reducing the integration problems occurring between team members that develop software collaboratively [1].

With the advent of social media in (OSS) software development, recent years have witnessed many changes to how software is developed, and how developers collaborate, communicate, and learn [2]. One such prominent change is the emergence of the pull-based development model [3], made popular by the "social coding" platform GitHub. In this model one can distinguish between *direct* contributions to a project, coming from a typically small group of developers with write access to the main project repository, and *indirect* ones, coming from developers who fork the main repository, update their copies locally, and submit pull requests for review and merger.

GitHub's implementation of the pull-based development model enables anyone with an account to submit changes to any repository with only a few clicks. This represents an unprecedented low barrier to entry for potential contributors, but it also impacts testing behavior [4]: GitHub project owners reported scalability challenges when integrating outside contributions, driving them towards automated tests. Automated CI services, such as Travis-CI[1]—integrated with GitHub itself—or Jenkins[2], facilitate this process: whenever a commit is recorded or a pull request is received, the contribution is merged automatically into a testing branch, the existing test suite is run, and the contribution author and project owner are notified of the results.

This extended abstract summarizes the finding of our study of Travis-CI, arguably the most popular CI service on GitHub [5].[3] We quantitatively explore to what extent GitHub developers use the Travis-CI, and whether the contribution type (direct or indirect) or project characteristics are associated with the success of the automatic builds.

II. Methodology

To understand usage of the Travis-CI service in GitHub projects, we extracted and integrated data from two repositories: (i) GHTorrent [6], a service collecting and making available metadata for all public projects available on GitHub; and (ii) the Travis-CI API[4].

Due to limitations of querying the Travis-CI API, we focus on a sample of large and active GitHub projects. Using the GHTorrent web interface[5], we selected all GitHub repositories that: (i) are not forks of other repositories; (ii) have not been deleted; (iii) are at least one year old; (iv) receive both commits and pull requests; (v) have been developed in Java, Python or Ruby; (vi) had at least 10 changes (commits or pull requests) during the last month; and (vii) have at least 10 contributors. We choose projects that receive both commits and pull requests, since we want to understand whether the way modification has been submitted (commit or pull request) can be associated with the build success. Our choice of the programming languages has been motivated by the history of Travis-CI: Travis-CI started as a service to the Ruby community in early 2011, while support for Java and Python has been announced one year later. We expect therefore the use of Travis-CI to be more widespread for Ruby than for Java and Python.

The data were extracted on March 30, 2014. After filtering our sample contained 223 GitHub projects, relatively balanced across the three programming languages: 70 (31.4%) in Java, 83 (37.2%) in Python, and 70 (31.4%) in Ruby.

To extract data about the automatic builds, we started by querying the `repos` endpoint of the Travis-CI JSON API to determine whether Travis-CI is configured for a particular project. Then, if the response was not empty, we iteratively queried the builds associated with this project (25 at a time as per the Travis-CI API) from the `builds` endpoint, collecting the `event_type` fields (that distinguish pull requests from pushes) and the `result` fields (that specify whether the build succeeded—0, or failed—1).

III. Results

We start by investigating the preference for direct and indirect contributions among the projects in our sample. The

[1] https://travis-ci.com

[2] http://jenkins-ci.org

[3] As supported, e.g., by the blog entries https://blog.codecentric.de/en/2012/05/travis-ci-or-how-continuous-integration-will-become-fun-again/ and https://blog.futurice.com/tech-pick-of-the-week-travis-ci, acc. 6/2014

[4] http://docs.travis-ci.com/api/

[5] Accessible from http://ghtorrent.org/dblite/

	Prog. lang.			Age (years)			Contributors		
	Java	Python	Ruby	<2	2–4	>4	≤17	17–33	>33
# projects	10	34	40	24	42	18	29	27	28
. . . s.t. $p < 0.05$	3	19	23	9	25	11	18	15	12
H_0	✗	✓	✓	✗	✓	✓	✓	✓	✗
%odds ratio>1	n/a	89	87	n/a	92	82	89	80	n/a

Table I
COMPARISON OF SUBGROUPS OF 84 GITHUB PROJECTS BASED ON THE PROGRAMMING LANGUAGE, AGE AND THE NUMBER OF CONTRIBUTORS.

shared repository model (with contributors having write access to the repository) is more popular among Java projects, while Python and Ruby projects have more contributors submitting pull requests. Overall, similarly to Gousios, Pinzger, and van Deursen [3], we see that direct code modifications are more popular than indirect ones, with only a small number of projects having more pull requests than commits.

Next we observe that an overwhelming majority of the projects are configured to use TRAVIS-CI (206 out of 223 projects, or 92.3%).However, slightly less than half of the 206 projects (93, or 45%) have no associated builds recorded in the TRAVIS-CI database. This shows that while most projects *are ready* to use continuous integration, significantly fewer *actually do*. Moreover, among the projects configured to use TRAVIS-CI but not actually using it, Java projects are overrepresented, while Ruby projects are underrepresented.

We have observed that the median success rate of 79.5% for commits and of 75.9% for pull requests. To obtain a more refined insight in whether the success of a build is independent from the way the modification has been proposed, we focused on projects that had at least 5 failed and at least 5 successful builds for each contribution type, as required by the χ^2 test of independence. Out of 113 GITHUB projects configured to use TRAVIS-CI and actually using it ($206 - 93 = 113$), 84 projects had sufficient data for the χ^2 test. Among the remaining 29 projects, in most cases it was the failed pull requests cell that had insufficient data, i.e., builds fail less frequently when contributions are submitted via pull requests. We believe this is because when a developer does not have commit rights, she will try harder to make sure the change is valid change and it will not break the build. However, when instead a developer has commit rights, she can try out new things more freely, since she also has the power to reverse the change.

The 84 projects subjected to the χ^2 test have been developed in different languages, have different ages and involve different numbers of contributors. Table I summarizes differences between those languages, ages, and numbers of contributors in terms of rejecting the null-hypotheses of the χ^2 test, i.e., independence of the build success from the way the modification has been proposed. We have used the common threshold of 0.05. The thresholds of 17 and 33 contributors correspond to the 33% and 67% percentiles. Performing Stouffer tests for each group led to very small p-values, indicating that results obtained for the majority of individual experiments can be lifted to the group level. Table I indicates that null hypotheses, e.g., can be rejected (✓) for Python and Ruby projects, but cannot be rejected (✗) for Java projects. Finally, all odds ratio tests for projects where the null hypothesis has been rejected (✗) for the group level turned out to be statistically significant ($p < 0.05$) and in almost all cases the odds ratios exceeded 1, i.e., whenever build success depends on the way the modification has been performed, pull requests are much more likely to result in successful builds than direct commits.

IV. CONCLUSIONS

In this paper we have studied a sample of large and active GITHUB projects. We observed that direct code modifications (commits) are more popular than indirect code modifications (pull requests). Next, we have seen that although most GITHUB projects in our sample are configured to use the TRAVIS-CI continuous integration service, less than half actually do. In terms of languages, Ruby projects are among the early adopters of TRAVIS-CI, while Java projects are late to use continuous integration. For those projects that actually use TRAVIS-CI, we conclude that pull requests are much more likely to result in successful builds than direct commits. However, we observe differences for projects developed in different programming languages, of different ages, and involving different numbers of contributors.

ACKNOWLEDGEMENTS

Special thanks to Mathias Meyer and the Travis CI team for helping us query their API. Bogdan Vasilescu gratefully acknowledges support from the Dutch Science Foundation (NWO) through the NWO 600.065.120.10N235 project.

REFERENCES

[1] P. M. Duvall, S. Matyas, and A. Glover, *Continuous integration: improving software quality and reducing risk.* Pearson Education, 2007.

[2] B. Vasilescu, A. Serebrenik, P. T. Devanbu, and V. Filkov, "How social Q&A sites are changing knowledge sharing in open source software communities," in *CSCW*. ACM, 2014, pp. 342–354.

[3] G. Gousios, M. Pinzger, and A. van Deursen, "An exploratory study of the pull-based software development model." in *ICSE*, 2014, pp. 345–355.

[4] R. Pham, L. Singer, O. Liskin, K. Schneider *et al.*, "Creating a shared understanding of testing culture on a social coding site," in *ICSE*. IEEE, 2013, pp. 112–121.

[5] B. Vasilescu, S. van Schuylenburg, J. Wulms, A. Serebrenik, and M. G. J. van den Brand, "Continuous integration in a social-coding world: Empirical evidence from GitHub," in *ICSM*. IEEE, 2014, pp. 401–405.

[6] G. Gousios, B. Vasilescu, A. Serebrenik, and A. Zaidman, "Lean GHTorrent: GitHub data on demand," in *MSR*, 2014, pp. 384–387.

Inferring Bad Smell Removal Recipes from Repository Mining: an initial exploration

Luis Mayorga[*], Javier Pérez[†]

November 3, 2014

As a software system evolves, it may grow in size and become more complex, which can have consequences over the quality of its design. Maintenance tasks — bug-fixing or adapting to new or changing requirements— will gradually degrade the original design, unless extra development effort is invested in refactoring the system [3].

In object-oriented programming, the degradation of the design can manifest in the form of bad smells [1]. Bad smells are a surface indication that usually corresponds to deeper problems in the system. A great number of these antipatterns are characterized and described in some catalogues. This means that they can be detected attending to certain rules and metrics [2].

Refactoring, restructuring an existing body of code, altering its internal structure without changing its external behaviour, can be applied as a solution to bad smells. removing the problems caused by a bad software design. As well as there are several identified ande documented bad smells, there exist refactoring books and guidelines that collect the solutions proposed by certain experienced developers on how to remove each one of them [5].

The purpose of this work is to analyse the refactoring strategies followed in actual software projects by checking the solutions that were applied in the past. This is a first exploratory study for mining refactoring recommendations. We have explored the feasibility of gathering empirical evidence on how bad smells are removed in practice, obtaining, if possible, rules that could help us to decide which refactoring strategies could be applied applied to solve this problems when they appear in other software systems. The approach is similar to the one draft idea presented in [4].

The procedure to conduct this study involved mining some mature open source repositories. A revision interval belonging to each one of these repositories was analysed and the bad smells present in each one of those revisions collected. This made it possible to know in which point in time they disappeared or were intentionally removed. Then we extrtacted the refactorings that were applied to each one of those cases and studied them.

A greater study would be needed in order to improve the results obtained and to understand the reasons behind some of our findings. Tracking the evolution of bad smells over time presents some difficulties. for example, some bad smells may disappear without being involved in an perfective maintenance processes. Some other bad smells might not show up after a certain point in time due to

[*] Universidad Politécnica de Madrid; Erasmus student at the Ansymo group

[†] Ansymo group, Universiteit Antwerpen

renamings applied over the affected entities. Nevertheless, we have observed that it is possible to mine repositories in order to get refactoring recommendations. The tool developed to perform the reported analysis, as well as the data, can be used in a future with this purpose [1].

Acknowledgments

We would like to thank Radu Marinescu and the rest of the Intooitus tem for providing us with an inFusion licence to run the experiments. This work has been sponsored by the Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT-Vlaanderen) under project number 120028 entitled "Change-centric Quality Assurance (CHAQ)". Luis Mayorga was funded by the EU Erasmus program.

References

[1] Kent Beck and Martin Fowler. *Bad Smells in Code*, chapter 3. Refactoring: Improving the Design of Existing Code. 1999.

[2] Michele Lanza and Radu Marinescu. *Object-Oriented Metrics in Practice - Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems.* Springer, 2006.

[3] Meir M. Lehman. Laws of software evolution revisited. In *EWSPT '96: Proceedings of the 5th European Workshop on Software Process Technology*, pages 108–124, London, UK, 1996. Springer-Verlag.

[4] Javier Pérez, Alessandro Murgia, and Serge Demeyer. A proposal for fixing design smells using software refactoring history. RefTest 2013: International Workshop on Refactoring & Testing, June 2013.

[5] William C. Wake. *Refactoring Workbook.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[1] `https://github.com/rapsioux/nose`

Explaining why methods change together

Angela Lozano, Carlos Noguera, Viviane Jonckers
Software Languages Lab.
Vrije Universiteit Brussel
Email: {alozano,cnoguera,vejoncke}@vub.ac.be

I. Introduction

Co-change has been identified as an appropriate method to detect hidden dependencies, and as a good predictor of the impact of a change. However, co-change analysis only considers the frequency with which source code entities (co-)change, but so far, no insight has been gained on *why they co-change*. Being able to derive the rationale behind co-changes *could allow* to document design knowledge, enforce design restrictions, and to make predictions for new methods.

This abstract shows an automatic approach to derive the *reason* behind a co-change, which aim at indicating the cause of logical couplings. We define the *reason* of a (set of) co-change(s) as a set of properties common to the elements that belong to the co-change, at that point in time. The granularity level for our approach is methods because they are functionality units with unique purposes, and therefore their relations may convey some underlying rationale.

The approach considers two kinds of properties that might provide reasonable explanations for a co-change. First, *structural properties* explain co-changes by evidencing explicit dependencies common to the elements that change. Structural properties represent relations along which changes may propagate. Examples of structural reasons include type references, exceptions thrown or cached, and the type that defines the method. Second, *semantic properties* explain co-changes by evidencing implicit dependencies common to the co-changing methods. Semantic properties identify the concepts that a method handles. Semantic properties include names of methods, parameters or local variables defined within the method, as well as recurring terms in the documentation (JavaDoc) attached to the co-changing methods.

Consider the commit transaction with message *'priority of task is save and open on xml files'* and time-stamp *'2003-05-22 22:56'* of GanttProject which modified two methods[*]. Using the properties defined above, we describe the methods on this commit transaction using 93 properties (50 for the first one, and 43 for the second one). Since the methods shared 9 characteristics, the *reason for the commit transaction* becomes: CALLS_METHOD_NAME: add, equals, get, getLength, size, toString; LOCAL_VARIABLE_DECLARATION_NAME: i, task; and LOCAL_VARIABLE_DECLARATION_TYPE: GanttTask.

We perform our analysis over the history of two Java open-source systems (Freecol and Ganttproject), analyzing nearly 19.000 methods and over 3700 (trunk) commits.

II. Evaluation of the reasons

We analyze to what extent our reasons: (1) cover a good percentage of co-changes, (2) are plausible explanations of the co-changes among those methods, (3) describe only the methods that should co-change, and (4) are different from each other –so that different co-change relations have different rationales. We propose a metric to evaluate each one of the previous characteristics, and we use them to evaluate the quality of different reasons.

Also, we compare the reasons found for two types of co-change relations. First, for sets of methods that co-changed in the *same commit transaction* and for sets of methods that *recurrently co-changed*. Given that that commits tend to be either too fine grained or too coarse grained we expect to find that the reasons for methods co-changed in a commit transaction perform worse than the reasons for recurrently co-changing methods. This is because the latter ones are more likely to reveal logical dependencies like the ones behind methods implementing the same feature or concept.

Do these reasons describe most of the co-changes? We analyze this question by comparing the percentage of commits (or methods) in the system that have a reason[†].

In general, our approach finds reasons for a third of the single co-changes (i.e., commits) but for less than 1% of the clusters of co-changes. Considering both type of properties lead to more reasons than when only a single type of property was analyzed. However the improvement was higher against structural properties than against semantic properties.

Do these reasons describe only co-changing methods? We analyze this question by comparing the percentage of methods that a reason described correctly (i.e., not by chance). A method "described by chance" by a reason is a method that has all property-values of the reason but was not modified in the co-change relation that the reason describes. Having many of these methods would indicate that the reason is not good because it does not describe only co-changing entities. For the example commit this is: 1-((5[‡] - 2[§])/(14895[¶])) = 1- 0.0002 = 0.9.

[*] GanttXMLSaver.writeTask(OutputStreamWriter,int,String) and GanttXMLOpen.DefaultTagHandler.startElement(String,String,String,Attributes

[†] Note that there could be co-change relations for which our approach cannot find a reason. That is, when there is no property-value shared by all co-changing entities

[‡] Methods whose description includes all the properties of the example commit: GanttXMLSaver.writeTask(OutputStreamWriter,int,String)
GanttXMLOpen.DefaultTagHandler.startElement(String,String,String,Attributes)
GanttXMLOpen.GanttXMLParser.startElement(String,String,String,Attributes)
GanttXMLSaver.writeTask(OutputStreamWriter,DefaultMutableTreeNode,String)
GanttXMLSaver.writeTask(Writer,DefaultMutableTreeNode,String)

[§] Number of methods modified in the example commit.

[¶] Methods belonging to GanttProject during the period analyzed

In general, all reasons extracted pointing out *only* at the methods that are likely to change, which indicate their usefulness of a reason for predicting co-changes . Results tend to be better for the single co-changes (i.e., commits) than for clusters of co-changes. Regarding the type of properties results are similar to those found for the first question. This is, both type of properties lead to higher discrimination values than when only a single type of property was analyzed, and also combining results was more beneficial to improve the results of structural properties than those of semantic properties.

Are these reasons unique? Given that reasons should serve as explanations *only* for the changes they describe, they should be sufficiently different between each other. We analyze this question by comparing the similarity of a reason to all other reasons extracted using their Jaccard index (i.e., the intersection over the union of their properties.).

For the example commit this is[‖] 0.985333.

In general, all reasons extracted tend to be unique. The uniqueness of reasons for the single co-changes (i.e., commits) is higher for Ganttproject, while the uniqueness of reasons for clusters of co-changes is higher for Freecol. When analyzing single co-changes (i.e., commits) using both type of properties lead to higher uniqueness values than when only a single type of property was analyzed. However, the improvement was not much different for structural or semantic properties. When analyzing clusters of co-changes the highest uniqueness values were achieved when using only semantic properties, followed by a combination of properties, and the worst uniqueness values were obtained when only structural properties were used.

Do these reasons make sense? We analyze this question by manually comparing the reasons for a (set) of commits produced by our analysis with the message that corresponds to the (set of) commits. We assert each reason as either plausible or not. We consider a reason a *plausible* explanation for a commit, if the words or terms mentioned in the properties of the reason appear in the commit message that accompanies the co-change. For groups of co-changes, for those that span more than 6 co-changes, we extract a word-cloud from the commit messages to provide an overview of the general terms therein. We apply a small degree of liberty when aligning the terms of the reason with those present in the commit message(s); for example if the commit message mentions UI, or mouse events, we will consider plausible reasons those that include dependencies to JPanel or ActionListener types (both types present in Java's SWING UI framework). Finally, note that the plausibility depends on the quality of the commit message, if the commit message provides no information (i.e., "bugfix" or "no message") we will consider the reason as an *implausible* explanation for the commit regardless of the properties found in the reason itself.

The example commit is considered plausible because the commit message and the reason refer to an I/O task[**].

Results indicate that reasons extracted from clusters are much more likely to describe logical dependencies among the co-changing methods than those extracted from commit transactions. This result was expected as clusters represent sets of methods that recurrently co-change, and thus are more likely to have structural and semantic similarities. Regarding the type of properties used, it is not clear which properties are more likely to convey plausible reasons. For Freecol, merging semantic and structural properties increases the chance of finding plausible reasons regardless of the sets of methods analyzed, while for GanttProject semantic properties better describe commits and structural properties better describe clusters.

III. Conclusions and Future work

In this paper we proposed an approach to automatically extract the rationale for a set of co-changing methods. We believe that the rationale for co-changes is orthogonal to the prediction of (co-)changes but is more likely to be remembered by developers because it helps to uncover design knowledge. The proposed approach is based on common properties shared by co-changing methods. We have shown that the reasons extracted are discriminating, unique and the plausible. The analysis of single co-changes provides a higher coverage, while the analysis of repeated co-change relations results in higher plausibility. Our study also shows that reasons that take into account structural and semantic properties tend to outperform reasons that use only one type of property. This approach could be used not only to document the reasons behind co-change relations, but also, to predict the impact of adding a new code entity to the system (i.e., by comparing its properties against known co-change reasons). We plan to perform a more extensive experimentation on the thresholds used for identifying the clusters of recurrently co-changing methods, improving the quality of the semantic properties extracted, and assessing the usefulness of other types of properties (like transitive-calls, or belonging to the same slice).

Acknowledgment

Angela Lozano is financed by the CHaQ project of the Agentschap voor Innovatie door Wetenschap en Technologie. Carlos Noguera is funded by the AIRCO project of the Fonds Wetenschappeljik Onderzoek.

[‖] The example commit had a non-empty intersection with 128 commits (from the 626 commits in GanttProject with a non-empty reason). Moreover, most of the intersections had only one property. Therefore, its similarity with other commits is very low (Min: 0, 1st Qu.:0, Median:0, Mean:0.018, 3rd Qu.:0, Max:0.3).

[**] See the description of the example in section **??**.

Evolution in a Clone-and-Own Setting – The Marlin Case

Stefan Stanciulescu
ITU Copenhagen
Email: scas@itu.dk

Sandro Schulze
TU Braunschweig
Email: sanschul@tu-braunschweig.de

Abstract

Multi-variant systems are often realized by clone-and-own. Usually, they do not diverge entirely and thus, share common source code over their whole lifecycle (while diverging in other parts). Hence, changes to such common parts should be recognized and propagated to related variants. While current SCMs basically support managing and propagating changes, only few is known how this can be used in a clone-and-own setting. In this paper, we discuss our experiences with the multi-variant open source system Marlin, a 3D printer firmware.

1. Introduction

Clone-and-own constitutes a development paradigm that aims at reusing source code in a structured and predefined way. Basically, an existing system (or parts thereof) is copied to a separate location and subsequently changed or extended to meet new requirements. As such, it is a preferred technique for developing *multi-variant systems (MVS)* (a.k.a. software product lines), that is, a set of similar, yet well-distinguished program variants that share a common code base [1]. Especially with the increasing importance of source code management systems (SCM) such as Git, this method receives even more acceptance. Particularly, the concept of *forking* has been shown to improve coordination between variants [2]. Note that we distinguish in the remainder between forking and cloning as follows: *Forking* is a copy using the respective built-in mechanism of SCM systems such as Github and thus, establishes a traceability link. In contrast, *cloning* is just copy&paste a whole repository without any traceability.

While the creation of new variants is easy and straightforward with clone-and-own, the evolution of an resulting MVS is not. Particularly, information about changes to the common code base (i.e., code that remains identical after cloning) has to be available in order to propagate changes. However, only few is known about whether and how developers keep track of this information in order to efficiently propagate changes between these systems.

In order to understand how current SCMs are used to support the evolution of a multi-variant system, we study the *Marlin* OSS, a firmware for 3D printers that constitutes the above mentioned characteristics. Particularly, we are interested in whether and how forking is used for both, creating variants as well as managing and propagating changes between variants. We report on preliminary results in this paper. Even more important, we would like to discuss these results as well as participants' experiences on evolving clone-and-own systems during the corresponding presentation.

Table 1. Statistics of Marlin MVS for the most active forks (all systems available at `www.github.com`)

system	start date	#commits*	formal/informal clone	#forks
Marlin	Aug 2011	1 498	informal clone of Sprinter/GRBL	1 478
Ultimaker/ Marlin	Dec 2012	1 428	formal clone of Marlin	9
Ultimaker2 Marlin	Mar 2013	1 172	clone of Marlin, but not via forking	40
Sailfish-MightyBoard	Oct 2012	1 488	informal clone of Marlin	16
Sprinter	Apr 2011	581	informal cone of Tonokip	279
Grbl	Jan 2009	537	N/A	426

* we only considered commits from the default branch

2. The Marlin Multi-Variant System

The development of Marlin, our subject system, started in 2011 by reusing parts of two other firmware systems, namely *Sprinter* and *Grbl*. Although both systems are projects on Github, reusing took place offline, followed by considerable modifications/extensions, leading to the Marlin firmware. Moreover, Sprinter has been evolved from other, meanwhile deprecated, firmware systems, all of which are settled around the *RepRap* 3D printer[1].

Given this starting point, numerous people have contributed to Marlin as well as created specialized variants. In Table 1, we give an overview of the resulting multi-variant system that is subject of our study. We distinguish between formal clones (as result of forking) and informal clone (as result of copy&paste). For instance, the Ultimaker firmware is a formal clone that has been forked by the

[1] http://reprap.org/wiki/Main_Page

correspondent company for using Marlin with a certain kind of 3D printer in a commercial setting. Hence, we assume they feel more secure when developing the firmware in a separate fork. Moreover, the Sailfish firmware actually comes in two different versions, both of which originate from Marlin, but are used for different platforms. Amongst others, these firmwares improve the speed of the acceleration planner.

We even discovered more informal clones of Marlin, but omit them here for sake of brevity. To summarize, we observed a variety of variants for different reasons (commercial use, feature enhancement, multi-platform), mostly by creating informal clones of Marlin, that make up a family of highly related systems. Since all of them are based on Marlin, we would expect that a subset of changes of these systems are also related to Marlin's changes and thus, change propagation would be beneficial.

3. Evolving a Clone-and-Own Multi-Variant System

Next, we present first insights of our analysis of the Marlin MVS. To this end, we formulate two questions we would like to answer by investigating the systems.

How is forking used to manage variants? Regarding common assumptions from the literature, forking is the main mechanism for creating variants. Compared to ad-hoc cloning by copy-and-paste, forking is a built-in mechanism of SCMs, such as Github, that establishes traceability links. In Table 2, we provide statistics about forking in the Marlin multi-variant system.

Our data reveal that forking is used quite frequently to clone the Marlin repository, but the majority of forks ($\sim 73\%$) are *not active*, i.e., no commits are authored and committed by the fork owner and thus, are probably used for tracking purposes only. Considering the remaining forks, we observed that most of them have only few commits. By manually reviewing a sample set of these forks, we observed that these changes are mainly for configuration issues, that is, to configure the firmware for a specific 3D printer. Hence, only the remaining 60 forks with more than 10 commits represent those systems that are subject to more fundamental changes such as enhancing or changing functionality.

Beyond these systems, we also discovered systems that have been cloned manually and thus, can not be detected by analyzing Github forking data only. Currently, we are not aware of any specific reason why ad-hoc cloning has been preferred over the built-in forking mechanism. However, regarding both, the huge number of meaningless forks as well as observing ad-hoc cloning, we conclude that forking is less often used for software variation than expected. This is not only opposed to common assumptions but may also have implications on methods for tracking and propagating changes amongst such related systems.

Table 2. Fork statistics of Marlin (extracted via GHTorrent by Oct 3rd 2014).

# forks	1 478
# forks directly from Marlin	1 131
# forks NOT directly from Marlin	347
# active forks (i.e., ever used)	394
# inactive forks (e.g., read-only, tracking)	1 084
# commits of fork owners	2 370
# forks with > 10 commits by fork owner	60
# forks with > 50 commits by fork owner	3
# forks with exactly one commit by owner	111
# forks with $1 < n < 10$ changes	334

How are mechanisms such as forking used for tracking and propagating (co)-changes between variants? Given the Marlin MVS with both, formal and informal clones, we are curious about how changes are propagated amongst the related systems. Clearly, this would be beneficial, especially in case of bug or performance fixes.

Generally, the formal clones, i.e., fork-related systems, are frequently synchronized with the fork parent. Additionally, Marlin has accepted more than 290 pull-requests, with patches for features, bug-fixes and support for new hardware. On the other hand, there are less changes between Marlin and its informal clones. One reason may be the lower amount of clones (6-8) compared to the high number of forks of Marlin (1500). Beyond that, propagation of changes has to be done manually. By comparing patches and analyzing commit messages, we identified six patches that have been propagated from Marlin to Sprinter. For all of them, the propagation is clearly indicated, e.g., by commit messages such as *"…from Marlin V1- big thanks"*.

However, while we are at the beginning and plan to investigate all of the firmwares that are part of this multi-variant system, it becomes already clear that the informal (and ad-hoc) cloning bears some problems for change propagation. First, pull-requests can not be used to communicate and accept changes. Second, due to missing traceability, developer are simply not aware of changes that might be of interest for their specific clone. Hence, we argue lots of reuse potential is not exploited, because mechanisms are needed to support this assumably widely used informal way of cloning repositories. For instance, a recently proposed methodology is *Virtual Platform*, which explicitly addresses the aforementioned shortcomings for clone-and-own systems [3].

References

[1] P. Clements and L. Northrop, *Software Product Lines – Practices and Patterns*. Addison-Wesley, 2001.

[2] A. N. Duc, A. Mockus, R. Hackbarth, and J. Palframan, "Forking and coordination in multi-platform development: a case study," in *ESEM*. ACM, 2014, p. to appear.

[3] M. Antkiewicz *et al.*, "Flexible product line engineering with a virtual platform." in *ICSE NIER*, 2014, pp. 532–535.

Towards An Empirical Analysis of Debian Package Conflicts

— BENEVOL 2014 extended abstract —

Maëlick Claes, Sébastien Drobisz, Tom Mens, Roberto Di Cosmo

November 3, 2014

1 Problem Statement

Package-based software distributions (such as the Debian Linux distribution) and other large component-based software repositories have been shown to suffer from maintainability and scalability problems due to the so-called "co-installability problem" [1]. Desired software packages or components may not be installable together due to conflicting dependencies, and detecting such conflicts is algorithmically hard.

While efficient algorithms and tools have been proposed for detecting co-installation conflicts and supporting their resolution [2, 3, 4, 5], little is known about how this problem evolves throughout the history of the considered software repository.

Therefore, we empirically analyse the evolution history of package-based software repositories, in order to assess the extent of the "co-installability problem" and how this problem evolves over a longer time period, involving many different versions of the software package repository. This is different in scope from the research presented in [6], where the evolution of the co-installability problem is studied between pairs of successive versions of a package repository, in order to identify the so-called "broken sets".

To avoid any confusion, we emphasise that the *co-installation conflicts* we are interested in are different from the *declared conflicts* that are explicitly specified in the control file of each package.

2 About the case study

Debian GNU/Linux is a free distribution of the Linux operating system, initially released in 1993. To facilitate maintenance and collaborative work, Debian is composed of tens of thousands of different packages, developed by thousands of developers. In our study we focus on the i386 architecture as it can be considered historically as the first one for which Debian was available.

There are essentially three types of Debian distributions. The *stable* distribution is the latest official release, and only contains stable, well-tested packages. The *testing* distribution contains package versions that should be considered for inclusion of the next stable Debian release. The *unstable* distribution contains packages that are not thoroughly tested and that may still suffer from stability and security problems.

Because we are interested in studying the evolution of Debian *development* activity, we will analyze these three types of Debian distributions in order to highlight differences between them. We have extracted all available daily snapshots from `http://snapshot.debian.org/archive/debian` for the three versions for the i386 architecture.

A major problem when analysing co-installability of packages is the sheer size of the package dependency graph: there are typically thousands of different packages with implicit or explicit dependencies to many other packages. Vouillon [5] addressed this problem by proposing an algorithm and theoretical framework to compress such a dependency graph to a much smaller one with a simpler structure, but with equivalent co-installability properties. The idea is that sets of packages are bundled together into an equivalence class if these packages are co-installable together, while they are not co-installable with the same other packages.

The coinst tool (http://coinst.irill.orgcoinst.irill.org) was developed specifically for extracting and visualizing coinstallability kernels for GNU/Linux distributions. We used the output of this tool as the basis of our analysis. We excluded packages in the other archive areas (*contrib* and *non-free*) as they are not considered to be part of the Debian distribution.

To retrieve the information about the co-installation conflicts of these packages we used the output generated by the coinst tool with the command

```
coinst -conflicts conflicts.json -stats -o graph.dot Packages.bz2 >& log
```

3 Research Questions

In this talk, we present some of the findings that we obtained while studying the evolution of this distribution. In particular, we focus on the following research questions:

- How long does it take before a co-installation conflict is introduced in (resp. removed from) a package?
- Is there any correlation between co-installation conflicts and other characteristics of packages (e.g., their longevity)?
- How do package dependencies influence co-installation conflicts?

References

[1] R. Di Cosmo and J. Vouillon, “On software component co-installability,” in *SIGSOFT FSE.* ACM , 2011, pp. 256–266. [Online]. Available: http://dx.doi.org/10.1145/2025113.2025149

[2] C. Artho, K. Suzaki, R. Di Cosmo, R. Treinen, and S. Zacchiroli, “Why do software packages conflict?” in *Int’l Conf. Mining Software Repositories*, 2012, pp. 141–150.

[3] R. Di Cosmo, D. Di Ruscio, P. Pelliccione, A. Pierantonio, and S. Zacchiroli, “Supporting software evolution in component-based FOSS systems,” *Sci. Comput. Program.*, vol. 76, no. 12, pp. 1144–1160, 2011.

[4] R. Di Cosmo and J. Boender, “Using strong conflicts to detect quality issues in component-based complex systems,” in *Indian Software Engineering Conf.*, 2010, pp. 163–172.

[5] J. Vouillon and R. Di Cosmo, “On software component co-installability,” *ACM Trans. Softw. Eng. Methodol.*, vol. 22, no. 4, p. 34, 2013.

[6] J. Vouillon and R. Di Cosmo, “Broken sets in software repository evolution,” in *Int’l Conf. Software Engineering*, 2013, pp. 412–421.

Analyzing the Linux Kernel Feature Model Changes Using FMDiff

Nicolas Dintzner
Software Engineering Research Group
Delft University of Technology
Delft, Netherlands

Arie van Deursen
Software Engineering Research Group
Delft University of Technology
Delft, Netherlands

Martin Pinzger
Software Engineering Research Group
University of Klagenfurt
Klagenfurt, Austria

Keywords: *Software Product Line, Feature Model, Evolution.*

The Linux kernel feature model has been studied as an example of large scale evolving feature models and yet details of its evolution are not known. We present here a classification of feature changes occurring on the Linux kernel feature model, as well as a tool, FMDiff, designed to automatically extract those changes. With FMDiff, we obtained the history of more than twenty architecture-specific feature models, over sixteen releases, and we compared the recovered information with Kconfig file changes. We establish that FMDiff provides a comprehensive view of feature changes and show that the collected data contains valuable information regarding the Linux feature model evolution.

Using this information, we performed an exploratory study of changes occurring in the Linux kernel feature model. We show that modifications of existing attributes and constraints of features play a major role in the evolution of the Linux kernel feature model, and yet such changes are often overlooked by current research. Finally, by comparing the evolution of the different architecture specific feature models, we show that 10 to 50 % of feature changes performed in a given release affect the capabilities of all of them, thus making generalization of observations on feature evolution from one architecture specific feature model to others difficult.

On the use and purpose of Java annotations

Carlos Noguera, Angela Lozano, Viviane Jonckers
Software Languages Lab.
Vrije Universiteit Brussel
Email: {cnoguera,alozano,vejoncke}@vub.ac.be

I. Introduction

Java 1.5 (released by Sun in 2004) introduced, amongst other programming constructs, an annotation facility. Annotations are a mechanism to attach meta-data to programming language constructs such as classes, fields or methods. The meta-data is modelled with annotation-types.

This paper presents an exploratory study into the use and definition of annotations in 97 active open-source projects. The study has two goals: to identify which annotations are used and to assess what they are used for.

II. Study set-up

We constructed a corpus made out of 97[1] open source Java repositories hosted in GitHub. The repositories were randomly selected through the use of GitHub's search. We selected projects implemented in Java, which were modified in the last 7 days (counted from December 4, 2013) and that had a repository size greater than 3 MB. The last snapshot of each repository was downloaded, and each Java file for each repository was processed using a JavaCC-based Java parser[2]. In total we gathered 101.611 Java files (the mean repository having 1024 files). The repositories in total contained 373.795 annotations, with all repositories containing at least one annotation.

III. Analysis

Before considering our research questions, we first consider whether annotations are actually used in the repositories in the sample. Figure 1 plots the relation between number of files (on the x axis) and number of annotations (on the y axis) for each of the repositories in the sample. The figure shows that the number of annotations in a repository is roughly proportional to the size of the repository. When considering the ratio of files that contain annotations, we find that repositories have a mean of 62% of annotated files with 54% as the first quartile, 65% as the median, and 74% as 3rd quartile. This suggests that repositories are homogeneously annotated.

A. Assessing which annotations are used and where

RQ1.1 Which source code elements are most frequently annotated?: Table I shows two metrics gathered in order to identify the location of annotations. First, we consider the possible targets of annotations. The *likelihood* indicates the percentage of annotations placed on a type of source code entity. We find that 85% of the annotations are placed on methods, followed by classes or interfaces (6%) and fields (3%). Second, we consider *chance* of a source code entity being annotated, which follows a different pattern: only 7% of the methods, 5% of the types and 0.5% of the fields are annotated. The most *annotated* source code elements are annotation type declarations, of which almost 90% carry an annotation.

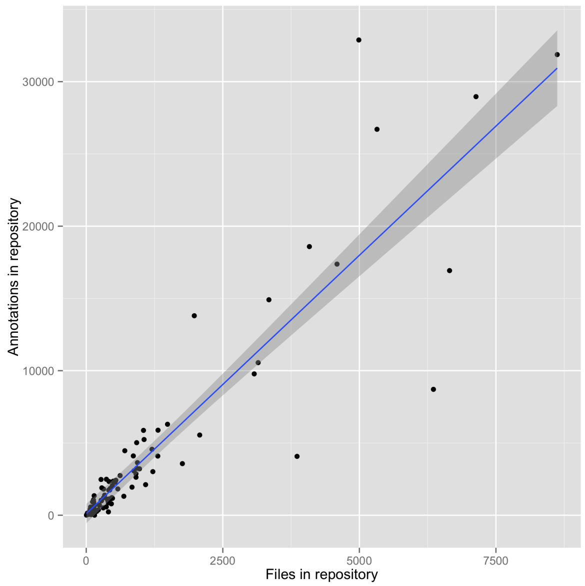


Fig. 1. Number of annotations in repository vs number of files in repository

Code Entity	Likelihood	Chance
Method	85.3	7.5
Type	5.8	4.7
Field	3.71	0.5
Parameter	3.7	0
AnnType	0.72	89.1
Constructor	0.64	0.2
Enum	0.05	0
EnumConst	0.02	0
AnnMember	0.01	0

TABLE I
Likelihood and chance of annotated elements. Likelihood represents the probability of an annotation being on an element and chance the probability of an element being annotated.

[1] Originally we selected 100 projects, but we had to discard the 3 largest ones because of their size (combined 14.5GB)

[2] https://code.google.com/p/javaparser/

Purposes			
Aspect	Bug Prevention	Compiler	Framework
Generation	Persistence	Security	Testing
Android	Annotation	Concurrency	Configuration
Debugging	Documentation	Evolution	Test-Input
Logging	Mapping	Mock	Monitoring
Performance	Resources	Runtime	Simbiosis
Special	UI	VM	Web

TABLE II
PURPOSES OF THE ANNOTATIONS IN THE REPOSITORY.

RQ1.2 Which annotations are the most frequently used across projects?: To address this question, we count first in absolute numbers the annotations present in the sample, and then we count the number of repositories in which each annotation is present. These two metrics allow us to assess both which annotations are most frequently used, and which annotations are most popular across projects.

We find that `java.lang.Overrides` is both the most used annotation (with 56% of the annotations found being `@Overrides`) and the most popular across repositories (with all of the repositories having at least an `@Overrides` annotation). The popularity of `@Overrides` can be attributed to IDEs (such as Eclipse) automatically adding the annotation to overridden methods, and sometimes signalling a warning whenever the annotation is missing.

B. Assessing the purpose of annotations

RQ2.1 Which annotation-frameworks are most commonly used?: We analysed the qualified names of the annotations used by the repositories of the sample, assigning each one to a framework. Doing this, we identified 116 distinct annotation frameworks. We find that the Java SDK is the most used annotation framework, both in number of annotations and number of repositories that use it. This is without a doubt due to the prevalent use of the `@Overrides` annotation, as well as `@SupressWarnings` and `@Deprecated`. We find that there are no frameworks widely used by the majority of repositories other than the Java SDK and JUnit. This implies that, as with annotations (Section III-A), annotation-frameworks used tend respond to repository or domain specific concerns.

RQ2.2 What are the most common purposes for annotation-frameworks?: Finally we consider the purpose of the annotations used in the sample. To assess this, we assign each package within a framework to one (or more) purposes. This was done to reflect the fact that annotations defined by a framework can fulfil different purposes. For example, the Spring framework defines annotations for persistence (`@Query`), testing (`@ExpectedException`), or runtime bug prevention (`@Validated`) In total we identified 28 possible purposes for annotations. Each package within a framework could be assigned one or more purposes (for example, Hibernate's `@WithMappingFiles` annotation in the `org.hibernate.jpamodelgen.test.util` package serves the purpose of both persistence and testing).

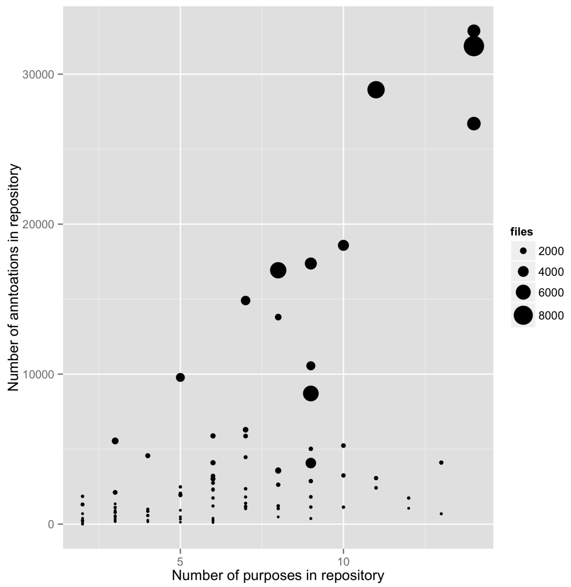


Fig. 2. Number of purposes versus number of annotations on a repository

Counting the number of annotations devoted to each purpose, we find that the most annotations are used for Bug prevention (63%), configuring the Compiler (62%), Testing (18%), Framework-specific purposes (8%), Configuration (4%), Persistance (3%), and Mapping (3%). The rest of purposes representing less than 1% (each) of the total amount of annotations in the sample. Note that `@Overrides`, is classified as both a Bug prevention annotation and a Compiler configuration annotation since it's purpose is to have the compiler flag a warning if a method is annotated as `@Overrides` but in effect does not overrides a method in a super class. This classification explains why Bug prevention and Compiler are the two most common purposes for annotations.

When considering whether annotations are used for different purposes on each repository, we find that repositories contain annotations for a median of 6 different purposes, with a first quartile, of 3 purposes, and a third quartile of 8. With this, we observe that annotations are used for multiples purposes across repositories. Furthermore, as Figure 2 shows, we observe two profiles for annotation use: either the number of purposes annotations are used for is proportional to the size and number of annotations (diagonal), or the repository is small and then it can use annotations for few or many purposes (lower area of the figure). No repository in the sample uses a large number of annotations for a single (or few) purposes. This seems to imply that when projects make the decision of using annotations, they use them for several purposes. This, in turn, seems to indicate that annotation use is a conscious decision, and not just added automatically by the IDE (as is the case with `@Overrides`).

Adventures in Analyzing Full Javascript Programs

Quentin Stievenart, Jens Nicolay, Coen De Roover, Wolfgang De Meuter

October 20, 2014

Nowadays, Javascript is used everywhere on the web, both client-side and server-side. However, Javascript's quirky semantics introduce many problems. Debugging Javascript applications can be really cumbersome, and Javascript applications are easily subject to security issues. It is therefore necessary to be able to precisely analyze Javascript programs in order to reason about their behaviour and their potential defects.

Having a tool able to reason precisely about the behaviour of Javascript programs would prove useful in many situations. In the case of software evolution, being able to compute the data flow of a program could for example be used to analyze whether changes between two different versions of a program consist of refactoring, feature addition, or modification of the behaviour of the program.

However, Javascript's suprising semantics complicate the job of designing static analysis tools. In order to detect complex defects or to compute the data flow in a program, one should go beyond the syntactic aspect of the language, and would need to simulate Javascript semantics, which is not an easy task. Existing tools either don't support the full language, or tools such as TAJS[5] and JSAI[3] rely on older versions of Javascript, and are therefore not adapted to analyzing programs that make use of features introduced in ECMAScript 5.

There has been a recent effort in formalizing Javascript semantics[1, 4], and in reducing the complexity of those semantics by introducing simpler languages that encode those semantics. Javascript programs can be automatically translated into those simpler language via a desugaring process. This is the approach taken by λ_{JS}[2] and its successor λ_{S5}, which can desugar ECMAScript 5 code to a simpler scheme-like language called *S5*.

We investigate the combination of this desugaring process with abstract interpretation, a static analysis method that consists of approximately reproducing the semantics of the language in order to have a decidable approximation of every path a program can take. The formalism we use is based on Might and Van Horn's CESK machine[6] and has the advantage of having tunable and arbitrarily high precision (at the cost of speed), depending on the precision needed for the analysis.

We developped a CESK machine for the S5 language and tried it out on desugared Javascript programs. Many challenges arose from this situation. To keep the S5 language simple, there exists a big *environment* definition in S5 itself, that encodes the behaviour of Javascript standard library. This allows us to keep the CESK machine simple, but introduces an interesting trade-off: when analyzing a function inside this environment, we would like to keep a high precision; however, outside this environment, precision can be lower. Another problem coming from this environment is its size. CESK machines are typically tested on small or medium programs, whereas the S5 environment file comprises around 8000 LOC, and the desugaring process introduces a large increase between the size of the program, due to the semantic gap between Javascript and S5.

In this presentation, we will explain the advantages of using S5 as an intermediate step to analyze Javascript programs, therefore allowing analysis of the full ECMAScript 5 language. We will illustrate the problems this approach introduces, explain how we solved some of those problems and what are the remaining challenges.

References

[1] Martin Bodin, Arthur Charguéraud, Daniele Filaretti, Philippa Gardner, Sergio Maffeis, Daiva Naudziuniene, Alan Schmitt, and Gareth Smith. A trusted mechanised javasript specification. In *Proceedings of the 41st annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 87–100. ACM, 2014.

[2] Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi. The essence of javascript. In *ECOOP 2010–Object-Oriented Programming*, pages 126–150. Springer, 2010.

[3] Vineeth Kashyap, Kyle Dewey, Ethan A Kuefner, John Wagner, Kevin Gibbons, John Sarracino, Ben Wiedermann, and Ben Hardekopf. Jsai: A static analysis platform for javascript. 2014.

[4] Sergio Maffeis, John C Mitchell, and Ankur Taly. An operational semantics for javascript. In *Programming languages and systems*, pages 307–325. Springer, 2008.

[5] Gregor Richards, Sylvain Lebresne, Brian Burg, and Jan Vitek. An analysis of the dynamic behavior of javascript programs. In *ACM Sigplan Notices*, volume 45, pages 1–12. ACM, 2010.

[6] David Van Horn and Matthew Might. Abstracting abstract machines. In *ACM Sigplan Notices*, volume 45, pages 51–62. ACM, 2010.

Mining Refactoring Guidelines from Stack Overflow

Alessandro Murgia*, Daan Janssens*,
Javier Pérez*, Serge Demeyer*

The first reference to refactoring belongs to Opdyke and is dated to 1992 [3]. From that time, software development has continuously evolved: modern object oriented and scripting languages such as Java and Python became more and more popular and Agile methods become the norm [1].

On one hand, refactoring guidelines —by means of books and tutorials— must catch up with recent trends in software development to satisfy the emerging needs of the broad and heterogeneous community of developers. On the other hand, the existing guidelines are written by a limited number of people based on their previous experience. In this research we explore the potential of crowdsourcing for obtaining up-to-date refactoring guidelines. We demonstrate that it is possible to mine Stack Overflow posts to obtain detailed refactoring guidelines including relevant code-snippets for areas which have received little attention in the current refactoring literature so far (web scripting, database queries).

We focus on Stack Overflow[1] one of the most popular websites among software developers and researchers who study the developer community [2]. Stack Overflow, with more than 3 million of users and more than 5 million of posts, keeps track of problems belonging to developers having different cultural and educational backgrounds, different skill sets and with experience matured in different environments. Exploiting the information reported in this knowledge-base repository is crucial if we want to understand how software evolution is reflected in thoughts and needs of developers approaching refactoring.

*Ansymo group, Universiteit Antwerpen

[1] http://stackoverflow.com/

We have analysed Stack Overflow to identify which are the main refactoring topics of developer discussions. Then, we have pursued the following research questions:

- **RQ1:** *What is the relevance of code on refactoring discussions?*
- **RQ2:** *Which are the most popular refactoring names developers use?*

Among other results, we have found that only a few of the refactoring related questions and answers on Stack Overflow mention a conventional refactoring name. We have also found that refactoring posts are more code-driven that other Stack Overflow posts.

The findings of our research have a direct impact on how refactoring guidelines should be written. We show how they should blend text descriptions and code snippets according to the typical layout of accepted answers (RQ1). Finally to maximize the audience of readers, we show which are the most well known refactoring names (RQ1) and the most popular programming languages (RQ2) suitable for talking about refactoring and writing code snippets as refactoring examples.

Acknowledgments

This work has been sponsored by the Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT-Vlaanderen) under project number 120028 entitled “Change-centric Quality Assurance (CHAQ)”.

References

[1] Scott Ambler. Survey says: Agile works in practice. Dr. Dobss, August 2006.

[2] Anton Barua, Stephen W. Thomas, and Ahmed E. Hassan. What are developers talking about? an analysis of topics and trends in stack overflow. Empirical Software Engineering, pages 1–36, 2012.

[3] William F. Opdyke. Refactoring Object-oriented Frameworks. PhD thesis, Champaign, IL, USA, 1992. UMI Order No. GAX93-05645.

The relationship between CC and SLOC: a preliminary analysis on its evolution

Davy Landman*, Alexander Serebrenik*†, Jurgen Vinju*†‡
* Centrum Wiskunde & Informatica, Amsterdam, The Netherlands
{Davy.Landman, Jurgen.Vinju}@cwi.nl
† Eindhoven University of Technology, Eindhoven, The Netherlands
a.serebrenik@tue.nl
‡ INRIA Lille Nord Europe, Lille, France

I. Introduction

Is it useful to measure both Cyclomatic Complexity (CC) and Source Lines of Code (SLOC)? In previous work [1] we have analyzed the reported linear relationship between CC and SLOC. In our large corpus of Java projects, we could not find such a linear relationship. Raising questions for future work.

Object Oriented Programming (OOP) could cause the lack of a linear relationship between CC and SLOC. In an OO language, dynamic dispatch and polymorphism are used as an alternative to control flow statements. However, related work [2], [3] reported linear relationships for both C++ and Java.

As identified in our earlier work, there is an open question of the evolution of this relationship. Therefore we explorer a possible evolutionary argument: are the Java programs of today using more OOP? And does this cause the decreased power of SLOC to predict CC?

II. Research method

As a preliminary study of the evolution of this relationship, we select one software systems and calculate CC and SLOC for each method over the last 10 years. To summarize over this period, we sample only the methods in the system at the end of every full year (e.g. 2003–2013 range).

A. Hypothesis

Related work measured the linear relationship with Pearson's correlation (R^2). Similarly to our previous work [1], we will calculate both the Pearson and Spearman correlation. The Pearson correlation will be calculated before and after a power transform. Moreover, we will also perform the Breusch-Pagan test [4] to confirm non constant variance (heteroscedasticity).

We have formulated the following two hypothesis.

Hypothesis 1. *Older revisions of a software system have a stronger linear (Pearson) correlation between the CC and SLOC metrics for Java methods then newer revisions.*

Hypothesis 2. *Older revisions of a software system do have constant variance between the CC and SLOC metrics for Java methods.*

B. System

We have selected a single system out of the Qualitas Corpus, DrJava. It was selected due to its domain and age. It is a Java Integrated Development Environment (IDE) with over 3000 revisions since 2000. The system grew from 30 K SLOC in 2003 to 200 K SLOC in 2013. We chose an IDE since they contain elements of multiple domains.

C. Measuring SLOC and CC

We use the same tools as in our previous study [1]. Eclipse JDT is used to parse Java methods into Abstract Syntax Tree (AST) form. This AST is visited and for each node that would generate a fork in the Java control flow graph, 1 is added to the CC of that method. For SLOC we use Rascal to tokenize Jave into newlines, whitespace, comments and other words. These tokens are then used to calculate the SLOC of a method.

III. Results

A. Correlation

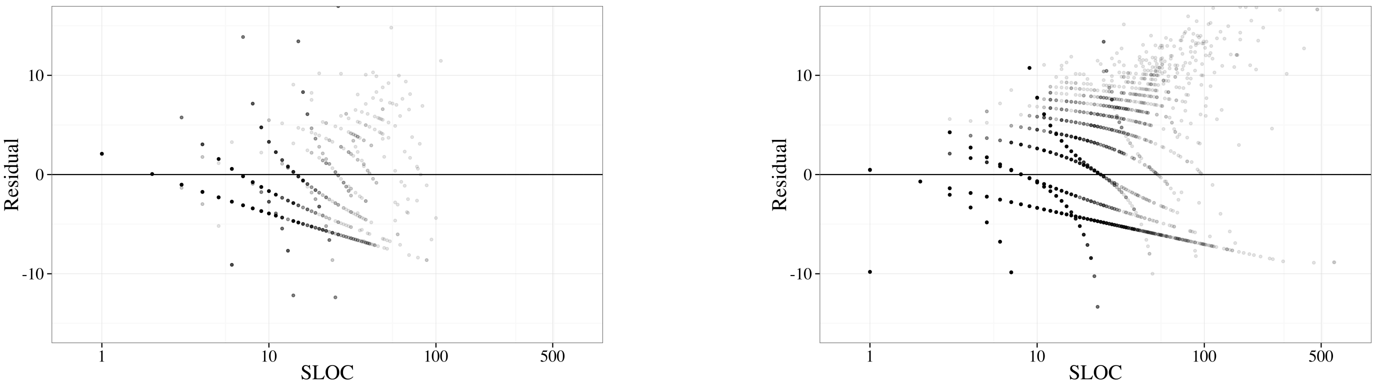
Table I contains the Pearson correlations before and after power transform, Spearman's correlation, and if the linear model was heteroskedastic.

TABLE I
Correlations between CC and SLOC for a period of 10 years and the total number of methods in that revision. All correlations have a high significance level ($p \leq 1 \times 10^{-16}$). Hetroskedasticy is checked by the Breusch-Pagan test (in all cases $p \leq 1 \times 10^{-16}$).

Year	Methods	R^2	$\log R^2$	ρ	Heteroscedastic
2003	3090	0.45	0.45	0.65	Yes
2004	4812	0.45	0.47	0.66	Yes
2005	9859	0.59	0.52	0.70	Yes
2006	10 262	0.56	0.47	0.67	Yes
2007	13 784	0.34	0.38	0.62	Yes
2008	14 998	0.35	0.39	0.62	Yes
2009	17 466	0.43	0.39	0.61	Yes
2010	19 765	0.44	0.40	0.61	Yes
2011	20 421	0.43	0.41	0.62	Yes
2012	20 470	0.42	0.42	0.63	Yes
2013	20 476	0.42	0.42	0.63	Yes

B. Scatter plots

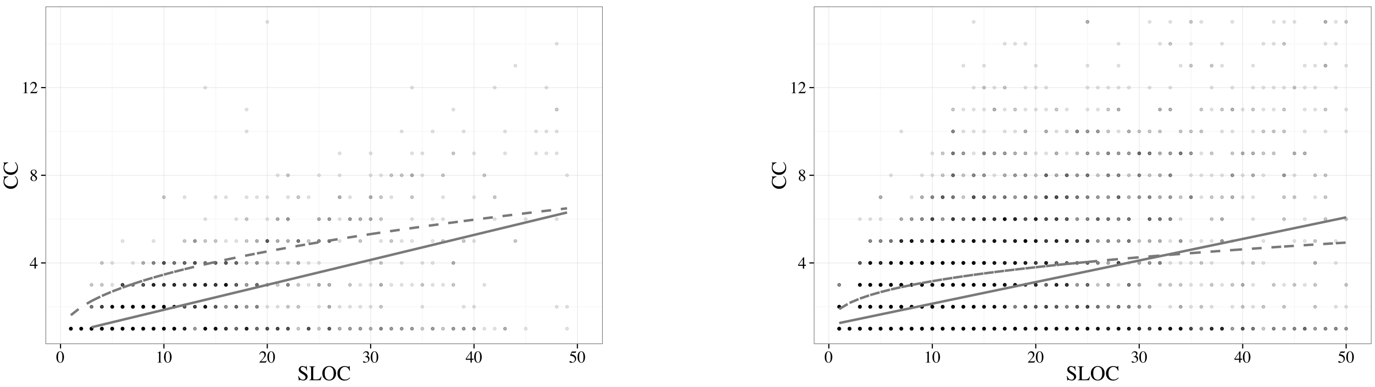
Figure 2 shows a zoomed-in (CC $\leq$ 20 and SLOC $\leq$ 50) scatter-plot of the methods of DrJava in 2003 and 2013. Due to



(a) 2003

(b) 2013

Fig. 1. Residual plot of the linear regression after the power transform, both axis are on a log scale. The non-constant variance complicates the interpretation of the linear regressions.



(a) 2003

(b) 2013

Fig. 2. Scatter plots of SLOC vs CC. The solid and dashed lines are the linear regression before and after the power transform.

the skewed-data, this figure still shows 98% of all data points. The two gray lines in the figure shows the linear regressions before and after the power transform. The gray scale gradient of the points in the scatter-plot visualizes how many methods have that combination of CC and SLOC: the darker, the more data points.

IV. Analysis

A. Hypothesis 1: correlation in older revisions

In Table I we see that although R^2 fluctuates over the years of DrJava's development, it remains near the 0.40, with the exception of 2005 and 2006. The increase in correlation could perhaps be explained by the big growth in 2005. However, we cannot confirm Hyptothesis 1, older versions of the software do not have a higher correlation.

B. Hypothesis 2: constant variance in older revisions

Table I shows that for all years the relation between CC and SLOC has non constant variance. The scatter plots in Figure 2 also visualize this growing variance, further shown in the residual plots in Figure 1. Therefore, we cannot confirm Hypothesis 2.

V. Discussion

We have presented a preliminary study on the evolution of the relationship between CC and SLOC. In the software system we analyzed, we did not observe a obvious change the linearity of the relationship. We also found that the heteroscedasticity reported in our previous work was present all versions of DrJava. Hetroscedasticity further complicates the inpretation of linear models.

In this abstract we have presented the evolution of one system. For future work, we would like to analyse the evolution of a whole corpus over the span of at least 10 years. Moreover, we are interested what other variables we might be measuring by comparing older version of the system with newer versions.

References

[1] D. Landman, A. Serebrenik, and J. Vinju, "Empirical analysis of the relationship between CC and SLOC in a large corpus of Java methods," in *30th IEEE International Conference on Software Maintenance and Evolution, ICSME 2014*, 2014.

[2] K. E. Emam, S. Benlarbi, N. Goel, and S. N. Rai, "The confounding effect of class size on the validity of object-oriented metrics," *IEEE Transactions on Software Engineering*, vol. 27, no. 7, pp. 630–650, 2001.

[3] G. Jay, J. E. Hale, R. K. Smith, D. P. Hale, N. A. Kraft, and C. Ward, "Cyclomatic Complexity and Lines of Code: Empirical Evidence of a Stable Linear Relationship," *Journal of Software Engineering and Applications*, vol. 2, no. 3, pp. 137–143, 2009.

[4] T. Breusch and A. Pagan, "A simple test for heteroscedasticity and random coefficient variation," *Econometrica*, vol. 47, no. 5, pp. 1287–1294, Sep. 1979.

An Exploratory Analysis of Identical Function Clones in CRAN

— BENEVOL 2014 extended abstract —

Narjisse Tabout, Maëlick Claes, Tom Mens, Philippe Grosjean
COMPLEXYS Research Institute, University of Mons, Belgium

October 17, 2014

1 Research Goal

Analysing the impact (whether it be harmful or beneficial) of code duplication has been an active subject of research for many years. Developers are often confronted with the difficult choice between depending on existing functions developed in other packages or libraries, or copy-pasting or reimplementing similar functions in their own code. In the former case of function dependencies across packages, errors may be introduced inadvertently during package updates, and finding and fixing these errors can be cumbersome. In the latter case, duplicating functions across different packages may be detrimental to the maintainability of the software ecosystem (i.e., the collection of software packages) in the long run.

In our research, we aim to study these issues for a specific software ecosystem, called CRAN, which is the official source of packages for the statistical R project and its associated language. By gaining a better understanding in package dependencies and function duplication across R packages, and how this evolves over time, we hope to be able to provide more adequate support for R package maintainers.

We are currently carrying out an exploratory qualitative analysis of the presence of identical function clones across R packages. Surprisingly, this appears to be a common practice in the R community. We aim to shed insight in this phenomenon to understand why this is the case. To do so, we propose a series of metrics to compute the frequency, size and abundance of function clones in CRAN snapshots, and to relate this to package dependency information. We use these metrics to provide statistical visualisations that help us in assessing how and why cloning behaviour occurs, and how this behaviour evolves over time.

2 About CRAN

CRAN is an archive network of software packages maintained by the community surrounding the statistical project R. The size of the CRAN archive is very large, containing over 5000 R packages being actively maintained by over 2500 maintainers. The number of packages is growing very rapidly, which constitutes a problem in the management of package dependencies.

From the user point of view, installing a package from CRAN will always download the latest version of the package (and its dependent packages, insofar as they have not yet been installed on previous occasions). From the maintainer point of view, A daily automated CRAN check verifies the compilability and other quality characteristics of its packages. Maintainers of problematic packages will be informed, and these packages will be archived from CRAN if the problem persists. This puts a heavy burden on package maintainers, especially if the problem is due to an update of a dependent package over which the maintainer has no control. A "solution" to avoid this problem would be to reduce the number of package dependencies, by duplicating the code of the functions of the dependent packages into ones own package. Copying existing code and pasting it in somewhere else followed by minor or major edits is a common practice that developers adopt to increase productivity [1].

Our previous work on CRAN includes an empirical study [2] of the maintainability of CRAN packages by focusing on the package update problem in presence of package dependencies. We have also implemented a web-based dashboard, called *maintaineR*, that is intended to be used by CRAN package maintainers [3]. This tool enables, among others, to see which functions in which package versions are cloned in other package versions. Other researchers have studied the evolution of CRAN [4], but we are not aware of any study that focuses on the presence of code clones in R packages and the consequences thereof. Obviously, there is a plethora of research results and tools for software cloning in general, but the context of CRAN is particular, and makes it worthwhile to study this aspect.

3 Research Questions

In our current research, we are performing an in-depth study of the phenomenon of identical function clones, that we have found to be present in many packages. By analysing this phenomenon across all CRAN packages over time, we aim to answer the following research questions:

- What is the number and proportion of packages in a given CRAN *snapshot* (i.e. the set of all CRAN packages available at a particular point in time) containing function clones?
- What is the proportion of functions in a package or in a CRAN snapshot that are actually identical clones?
- Can we identify specific patterns in the clone graph (composed of function clones, the packages in which they are contained, and the dependencies between these packages)?
- Is there a relation between the presence of identical function clones and other characteristics (such as function size, package size, number of package dependencies, package maintainer)?
- How does the presence of function clones evolve over time from different points of view (the cloned function, the package containing it, and the CRAN snapshot as a whole)?
- Is it possible to remove identical function clones in an automated way?

The ultimate goal is to provide tools, based on the answers to the above questions, that will help R package maintainers to manage their packages more effectively, and to improve the quality of these packages.

References

[1] C. K. Roy, M. F. Zibran, and R. Koschke, "The vision of software clone management: Past, present, and future," in *IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*, 2014, pp. 18–33.

[2] M. Claes, T. Mens, and P. Grosjean, "On the maintainability of CRAN packages," in *IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*, 2014, pp. 308–312.

[3] ——, "maintaineR: A web-based dashboard for maintainers of CRAN packages," in *International Conference on Software Maintenance and Evolution (ICSME)*, 2014.

[4] D. M. Germán, B. Adams, and A. E. Hassan, "The evolution of the R software ecosystem," in *European Conf. Software Maintenance and Reengineering*, 2013, pp. 243–252.

Mining GitHub for Fun and Profit: The GHTorrent project

Georgios Gousios,
Delft University of Technology
The Netherlands
g.gousios@tudelft.nl

Abstract

We present an overview of our activities in mining and analyzing the GHTorrent dataset, a large (5.6TB) off-line mirror of the data offered through the GithHub API.

I. Introduction

GitHub is a collaborative code hosting site built on top of the `git` version control system. It includes a variety of features that encourage teamwork and continued discussion over the life of a project. GitHub uses a “fork & pull” model where developers create their own copies of a repository and submit requests when they want the project maintainer to incorporate their changes into the project’s main branch, thus providing an environment in which people can easily conduct code reviews. Every repository can optionally use GitHub’s issue tracking system to report and discuss bugs and other concerns. GitHub also contains integrated social features: users are able to subscribe to information by “watching” projects and “following” other users, resulting in a constant stream of updates about people and projects of interest. The system supports user profiles that provide a summary of a person’s recent activity within the site, such as their commits, the projects they forked or the issues they reported.

With over 10.6 million repositories, GitHub is currently the largest code hosting site in the world. Software engineering researchers have been drawn to GitHub due to this popularity, as well as its integrated social features and the metadata that can be accessed through its API.

II. Large Scale Distributed Mining

To make research with GitHub data approachable, we created the GHTorrent project [1], a scalable, off-line mirror of all data offered through the GitHub API. GHTorrent follows the GitHub event stream and systematically retrieves all data, their metadata and their dependencies from it. It then processes and stores all retrieved items in a relational database, while also storing the original data in a MongoDB database. Interestingly, the data collection process is distributed; any interested researcher can participate either by contributing API keys or data collection workers. GHTorrent offers downloads of the corresponding database dumps (currently, 5.8 TB of data) and online querying facilities to interested researchers.

A novel feature of GHTorrent is the so-called “lean” version, an online service that offers customisable data dumps on demand [2]. The GHTorrent data-on-demand service offers users the possibility to request via a web form up-to-date GHTorrent data dumps for any collection of GitHub repositories. We hope that by offering customisable GHTorrent data dumps we will not only lower the ”barrier for entry” even further for researchers interested in mining GitHub data (thus encourage researchers to intensify their mining efforts), but also enhance the replicability of GitHub studies (since a snapshot of the data on which the results were obtained can now easily accompany each study).

III. Promises and Perils

Doing research with data as vast and diverse as those offered by GHTorrent includes inherent risks. To help developers avoid them, we documented [3] the results of an empirical study aimed at understanding the characteristics of the repositories and users in GitHub; we see how users take advantage of GitHub’s main features and how their activity is tracked on GHTorrent and related datasets to point out misalignment between the real and mined data. Our results indicate that while GitHub is a rich source of data on software development, mining GitHub for research purposes should take various potential perils into consideration. For example, we show that the majority of the projects are personal and inactive, and that almost 40% of all pull requests do not appear as merged even though they were. Also, approximately half of GitHub's registered users do not have public activity, while the activity of GitHub users in repositories is not always easy to pinpoint. We also used the identified perils to see if they can pose validity threats; we reviewed selected papers from the MSR 2014 Mining Challenge and saw several of those perils manifest, thereby rendering the results vulnerable to falsification. Finally, we provided a set of recommendations for software engineering researchers on how to approach the data in GitHub.

IV. Pull Requests

The advent of distributed version control systems has led to the development of a new paradigm for distributed software development; instead of pushing changes to a central repository, developers pull them from other repositories and merge them locally. Various code hosting sites, notably Github, have tapped on the opportunity to facilitate pull-based development by offering workflow support tools, such as code reviewing systems and integrated issue trackers.

In [4], we studied how the pull-based software development works, first on the GHTorrent corpus and then on a carefully selected sub-sample of 291 projects. We found that the pull request model offers fast turnaround, increased opportunities for community engagement and decreased time to incorporate contributions. We showed that a relatively small number of factors affect both the decision to merge a pull request and the time to process it. We also examined the reasons for pull request rejection and find that technical ones are only a small minority.

In the pull-based development model, the integrator has the crucial role of managing and integrating contributions. In a work submitted to ICSE 2015, we focused on the role of the integrator and investigated working habits and challenges alike. We set up an exploratory qualitative study involving a large-scale survey involving 749 integrators, to which we add quantitative data from the integrators project using the GHTorrent platform. Our results provide insights into the factors they consider in their decision making process to accept or reject a contribution. Our key findings are that integrators struggle to maintain the quality of their projects and have difficulties with prioritizing contributions that are to be merged.

V. Conclusions

Due to its openess and size, GHTorrent is becoming the *de facto* dataset for large scale quantitative analysis for GitHub data. So far more, than 50 researchers have subscribed and used the online access points. GHTorrent has enabled research ranging from distributed collaboration to sentiment analysis and pull request prioritization. GHTorrent was also the target of the 2014 mining challenge at the Mining Software Repositories conference and the visualization challenge of the 2014 VISSOFT conference. GitHub itself suggested GHTorrent as a potential datasource in their latest data analysis challenge.

The future of the GHTorrent project lies in the hands of the community.

References

[1] G. Gousios, "The GHTorrent dataset and tool suite," in *Proceedings of the 10th Working Conference on Mining Software Repositories*, ser. MSR '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 233–236.

[2] G. Gousios, B. Vasilescu, A. Serebrenik, and A. Zaidman, "Lean ghtorrent: Github data on demand," in *Proceedings of the 11th Working Conference on Mining Software Repositories*, ser. MSR 2014. New York, NY, USA: ACM, 2014, pp. 384–387.

[3] E. Kalliamvakou, G. Gousios, K. Blincoe, L. Singer, D. M. German, and D. Damian, "The promises and perils of mining github," in *Proceedings of the 11th Working Conference on Mining Software Repositories*, ser. MSR 2014. New York, NY, USA: ACM, 2014, pp. 92–101.

[4] G. Gousios, M. Pinzger, and A. v. Deursen, "An exploratory study of the pull-based software development model," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014. New York, NY, USA: ACM, 2014, pp. 345–355.

Decision Modules in Models and Implementations

Extended Abstract

Ella Roubtsova[1] and Serguei Roubtsov[2]

[1] Open University of the Netherlands
[2] Technical University Eindhoven, the Netherlands

Modularization plays an important role in system evolution. The chosen granularity and the content of the modules are able either improve or complicate system modification, requirements traceability and code analysis. Commonly used Object Life Cycle Modules (OLCM) hide the process control points, i.e. the decisions on which path to follow, inside objects. To modify a program or generate a test, the control points inside OLCM have to be analysed. In this paper we propose *decision modules* (DM), show their advantages for system evolution and investigate the possibility of their implementation in Java programs.

The necessity to modularise the control points had been recognised by the Business Rules community. Modules called enablers were suggested [1] there. An enabler has varying interpretations depending upon the nature of the correspondent object: it may permit (i.e., enable) the creation of a new instance; permit another action assertion; permit an action execution [1]. An enabler is often called an integrity constraint, a condition or a test.

We extend the idea of enablers and *define a decision module as an abstract description of the system actions and the states before and after these actions allowing or forbidding these actions. A decision module contains a description allowing to make a choice of an action from a predefined set or to make no choice.* Our decision module may be seen as a module because it can be associated with different objects as a separate entity. We name this module a decision module because it forms the condition for the acceptance or refusion of an action. The condition is derived from the pre- and post-states of the action in the life cycle modules of objects.

In a rather advanced form, such an approach to modularization can be seen in protocol models [3]. Protocol modeling [4] uses the CSP parallel composition [2] defined at the level of event accepting and refusing and extended for modules with internal data. The decision modules, localised in protocol models, possess unidirectional dependency. Unidirectional dependency (also known as obliviousness) means that the decision modules can read the information about the state of OLCMs but the OLCMs ”do not know” about existence of decision modules. The composition of decision modules with OLCMs does not change the execution sequences specified by OLCMs. This property is called observational consistency. Unidirectional dependency and observational consistency of decision modules make the system evolution less laborious: adding, modifying and deleting decision modules do not require changes in the related OLCMs.

It is desirable to implement such properties in executable programs. We have carried experiments with different implementation techniques (Table 1). Our experiments have shown that within the common Java paradigm with object composition, the desired unidirectional dependency cannot be implemented as the OLCMs have to explicitly invoke decision modules. The event-driven mechanism is also absent.

Technique	Modularity	Unidirect dependency	Mechanism:) Event-Dr.(CSP‖
Object Composition	yes	no	no
Publ.-Subscr.& Java Reflection	yes	state reading:yes; obliviousness:no	yes
EJB 3 with Interceptors	yes	yes	yes
EJB 3 with Delegation	yes	yes,for a given interface	yes

Table 1. Properties of decision modules in different Java implementations

Using Publisher-Subscriber design pattern, the event-driven mechanism can be implemented. In this case, OLCM becomes a listener of an event. Java Reflection allows the decision module to read the state of OLCMs. However, the OLCMs still need to invoke decision modules and, thus, "know about" them.

EJB 3.0 specification completely supports implementation of decision modules with the interceptor mechanism. In order to be composed with decision module, the OLCM should contain an *@Interceptors* annotation of the business method corresponding to an event. This annotation informs the application server that before invocation of this business method the corresponding decision module has to be invoked. The disadvantage of implementation of the decision modules using EJB3 is obvious: it's too heavy as it needs an application server to implement the interceptor mechanizm. However, if the system is already implemented as an enterprise application, this may be a viable solution. For the classes that have the same external behaviour (implement a certain interface) the decision modules can be implemented with the Decorator design pattern.

These results form a promising start for implementation of decision modules for real Java projects.

References

1. Business Rules Group. Defining Business Rules. What Are They Really? *http://www.businessrulesgroup.org/firstpaper/BRG-whatisBR_3ed.pdf*, 2000.
2. C. Hoare. *Communicating Sequential Processes.* Prentice-Hall International, 1985.
3. A. McNeile and E. Roubtsova. Aspect-Oriented Development Using Protocol Modeling. *LNCS 6210*, pages 115–150, 2010.
4. A. McNeile and N. Simons. Protocol Modelling. A Modelling Approach that Supports Reusable Behavioural Abstractions. *Software and System Modeling*, 5(1):91–107, 2006.

Managing the Evolution of Information Systems with Intensional Views

David Colpaert, Kim Mens & Bernard Lambeau
ICTEAM Institute, Computer Science Engineering pole
Université catholique de Louvain, Louvain-la-Neuve, Belgium
Email: kim.mens@uclouvain.be, bernard.lambeau@uclouvain.be

***Abstract*—Like any software system, information systems suffer from structural inconsistencies that may arise during system evolution. Appropriate tools are needed to encode the structural regularities the system should adhere to, and to check conformance of the system against those regularities upon evolution. Taking inspiration from the intensional views approach to document and verify structural regularities in source code, we developed a similar tool to document and verify structural regularities in large databases. Regularities are expressed by the user at a high level in a graphical user interface, and then translated into relational algebra in order to check the regularities over the data. Discovered inconsistencies are presented back to the user in appropriate high-level data views. As a case study, the developed tool was successfully applied to a safety critical information system deployed at a large Belgian university. It is used by the rescue services to accurately locate users based on the location of their IP phones from which an emergency call was made.**

***Keywords*—Intensional views, tools, conformance checking, structural regularities, information systems, relational algebra.**

I. Introduction

In previous work [1], [2], [3] we proposed the *Intensional Views* approach to document high-level *structural regularities* in the source code of a software system, and to *check conformance* of the source code against those regularities, in order to facilitate various *software maintenance and evolution* tasks. By documenting explicitly some of the coding conventions and idioms that are typically adhered to by software developers, and by providing an automated mechanism for verifying where and which source code entities do or do not respect these regularities, structural quality of the code can be improved and certain bugs can be discovered or avoided.

Essentially, an *intensional view* is nothing but a set of source-code entities (e.g., classes or methods in an object-oriented program) which are structurally similar (e.g., having a similar name, containing the same entities, or being related to other entities in a similar way). Instead of enumerating all entities that make up a view, they are defined by means of an *intension*, that is, an executable description which yields the set of entities belonging to the view. A logic language embedded in a reflective object-oriented programming language proved to be an ideal choice in which to define intensional views over source code entities in the object-oriented language. On top of this embedded logic language we developed a set of tools to facilitate the definition, conformance checking and visualisation of the intensional views.

The original approach described above focused on source code regularities only. However, it occurred to us that the underlying idea of the approach was generic enough to be applicable to any system containing structured information obeying certain regularities, where system evolution may lead to a decay in the conformance of the system to these regularities. In particular, in addition to software systems (and source code in particular), we believe that information systems (and databases in particular), suffer from similar problems. This article relates on a recent experience where we transposed the idea and tools of intensional views to this new application area.

More specifically, we implemented an information system intended to localise IP telephones at a large Belgian university. In this case study, we observed that the data sources used by the system suffered from problems very similar to those encountered by our earlier research on intensional views. I.e., the data sources contain a lot of structured information that obeys many regularities, but which are often not documented explicitly and for which evolution causes inconsistencies to arise in the data with respect to those regularities.

To address this problem we implemented a new tool, strongly inspired by the original *Intensional View Environment* [3], but now adapted and dedicated to databases. Like the original tool suite, this new tool consists of an *intensional view editor* in which the user can declaratively codify structural regularities to be respected by the databases of the information system. These regularities are expressed in terms of high-level intensionally declared views on the databases, and relations between those views. To check conformance of these regularities, the tool translates the high-level views and relations into more low-level relational algebra expressions that can be verified directly on the databases. After performing this verification, the results are reported back to the user in sufficiently high-level views, allowing him or her to easily discover what entities did and did not respect the regularities.

Section II describes our case study in detail. Sections III and IV then describe how we adapted the original idea of intensional views to apply it to codify and verify structural regularities on the data sources of our case study. Section V takes a step back to discuss some of the advantages, limitations and future improvements of this approach and provides some comparisons to the original intensional view approach. We conclude that the similarities between the intensional views approach applied to databases and to source code are striking, and that the approach can probably be applied to many other kinds of systems suffering from a decay of structured information throughout system evolution.

II. Case Study

The information system of our case study deals with localizing IP phones at a Belgian university. In case of emergency calls, the system needs to be able to inform rescue services about the precise location of an accident. To do this, it relies on three sources of data. First, each time an IP phone is deployed, a technician fills out an Excel document to specify the location of this phone. However, the problem is not only that these phones can be moved, but also that users can disconnect from a phone and then connect to another phone somewhere else, while keeping the same phone number. Phone number locations can thus evolve over time.

A more dynamic source of information about phone locations is thus needed. The system uses two additional data sources. The first is the network locations. Thanks to a mapping of port locations for each switch, we are able to localize phones connected to the network. Another source of locations is through the phone central (called MX1) and SAP. While the phone central specifies which phone number is connected to what phone MAC address, the SAP system contains the phone number and office location for each employee.

With these three data sources (deployment, network and MX1-SAP), we would like to localise phones accurately. Unfortunately, many inconsistencies remain between these sources. The main problem is that phone locations are not always the same according to the different sources. It occurs frequently that, according to some source, a phone is said to be in one building, while according to another source it is located in another building several miles away. And this is not the only problem; a lot of other constrains between the data sources are not satisfied either. For example, some phones existing in one source are simply missing from other sources. We faced thousand and thousand of errors of different types. All these errors were carefully stored in logs, but it soon appeared to be impossible to deal with all these inconsistencies manually.

Yet, it remains crucial to have consistent information about the locations, given the dire consequences that a location error could have. Indeed, sending the emergency services to a wrong place could cause them to lose precious seconds. We cannot afford inaccurate location information such as “Well, the victim may be here, or there, or perhaps there”.

We thus need a tool to help us detect inconsistencies in the data. The tool should allow us to define and verify a variety of structural constraints on our data sources. For example, we would like to express the constraint that a location (building + office) must be the same in all data sources for a same phone. One solution could be to express such constraints directly in the SQL query language. However, we wanted our tool to be high-level enough to be used by users which are not necessarily SQL experts. Especially since expressing such constraints often requires rather complex or verbose SQL expressions. Also, the results returned by such SQL expressions may not be easily exploitable. Our goal was to develop a tool where a user can express his constraints in an intuitive and high-level way, using a simple GUI, and which would return its results (i.e., discovered inconsistencies) in a way that can straightforwardly be exploited by end users.

Actually, some tools that satisfy some of the above requirements already exist. Query By Example (QBE) [4] could be a convenient way to express constraints such as the above. In fact, it could express most of the constraints we need to verify. However, it would require users to learn the specific QBE syntax. We would prefer a tool where the user does not have to learn a new language or syntax in order to be able to express, understand or detect violations of constraints.

Alternatively, we might directly use the constraint checking provided by SQL. Expressing constraints in terms of *uniqueness*, *primary key*, *not null* or other SQL constraints could already prevent a lot of inconsistencies. A problem we have, however, is that we need to be able to keep all original data, even though some of it is currently inconsistent. With upstream SQL constraint checking, the DBMS would simply deny such data, causing a loss of essential information that could have allowed us to discover the root causes of the detected problem more easily.

From the above analysis, we concluded that no existing tool seemed to satisfy all of our requirements, but that a tool akin to the original intensional views tool was probably what we needed to solve our problem. Indeed, intensional views for source code allow end users to define, in a high-level way, using an intuitive GIU, structural regularities on source code. These regularities are then translated to logic and verified over the code seen as a logic repository.

Database constraints could also be expressed and verified using first-order logic [5]. Theoretically, constraints could be specified by the user in an appropriate GUI and then automatically translated into propositional formulae, and then further into SQL. In practice, however, relational algebra may provide a more flexible intermediate language than first-order logic expressions. Indeed, relational algebra is closed over relations (every operator takes relations as input and produces a relation as output) which yields a naturally composable way for building complex constraints from user input.

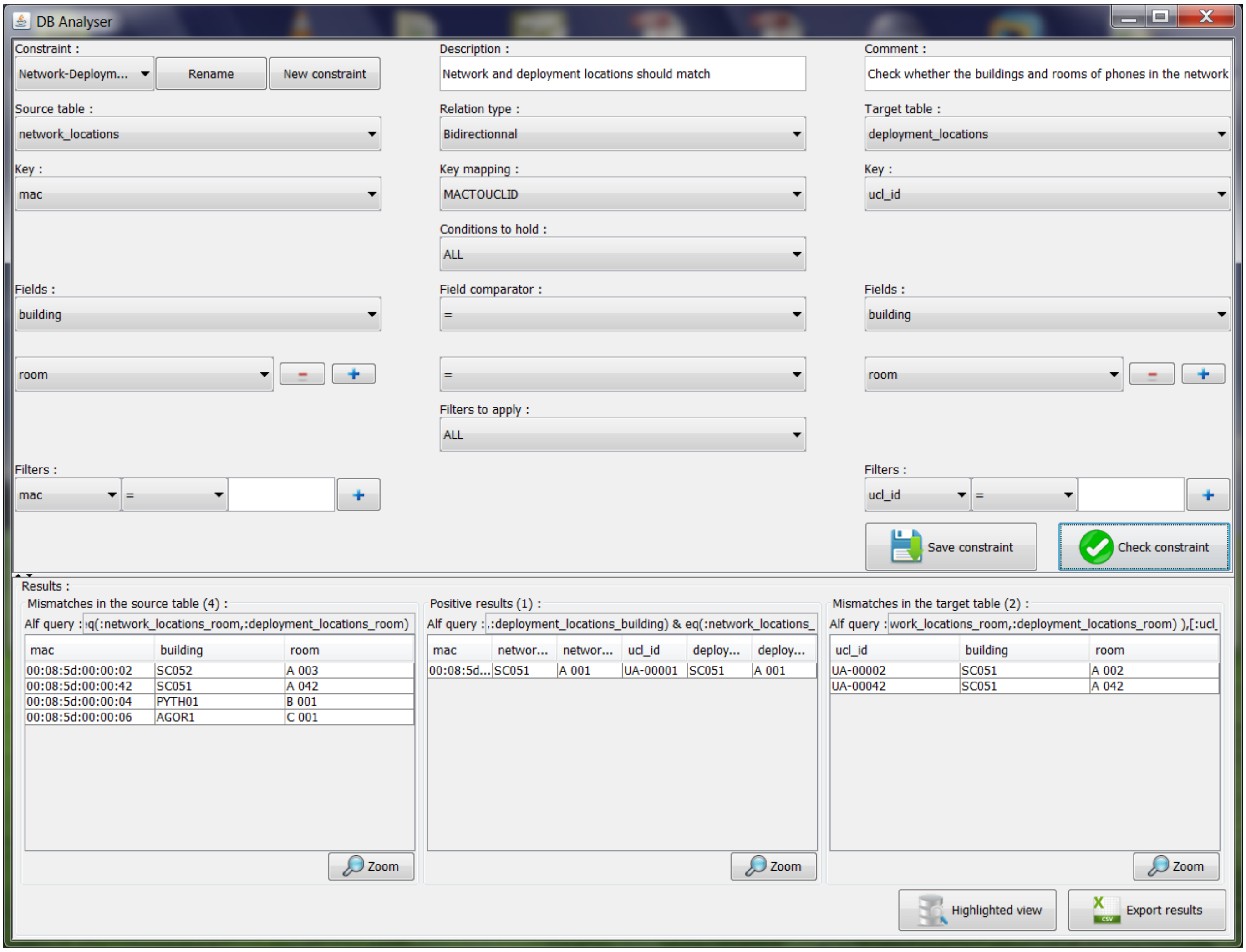
When porting the intensional views approach to the domain of databases, we therefore decided to translate the high-level structural regularities on the databases into relational algebra instead of logic. We use the *Tutorial D* language [6] as relational algebra, and the *Alf* tool [7] as concrete implementation. Alf provides support for compiling SQL code from arbitrary relational expressions.

Using examples from our case study, in the next section we will now describe our instantiation of intensional views for information systems, which relies upon Alf for verifying constraints over the data.

III. Proposed solution

Whereas initially intensional views were intended to check structural source code regularities [1], [2], here we want to apply this concept to check constraints on database tables and their tuples.

Figure 1 shows the *intensional view editor*, i.e. the GUI wherein the user would define his constraints on the data. It illustrates how a user can easily define a desired regularity to be respected by the data of two different tables, and how discovered inconsistencies with respect to that regularity would be reported back to the user. In this (simplified) example, we want to express the regularity that the locations of phones



Constraint : Network-Deploym...

Description : Network and deployment locations should match

Comment : Check whether the buildings and rooms of phones in the network

Source table : network_locations

Relation type : Bidirectionnal

Target table : deployment_locations

Key : mac

Key mapping : MACTOUCLID

Key : ucl_id

Conditions to hold : ALL

Fields : building

Field comparator : =

Fields : building

room | = | room

Filters to apply : ALL

Filters : mac =

Filters : ucl_id =

Results :

Mismatches in the source table (4) :

Alf query : :q(:network_locations_room,:deployment_locations_room)

mac	building	room
00:08:5d:00:00:02	SC052	A 003
00:08:5d:00:00:42	SC051	A 042
00:08:5d:00:00:04	PYTH01	B 001
00:08:5d:00:00:06	AGOR1	C 001

Positive results (1) :

Alf query : .:deployment_locations_building) & eq(:network_locations_

mac	networ...	networ...	ucl_id	deploy...	deploy...
00:08:5d...	SC051	A 001	UA-00001	SC051	A 001

Mismatches in the target table (2) :

Alf query : work_locations_room,:deployment_locations_room)),[:ucl_

ucl_id	building	room
UA-00002	SC051	A 002
UA-00042	SC051	A 042

Fig. 1. The *intensional view editor* for defining and checking structural constraints between database tables.

coming from the network data source and from the deployment data source are consistent.

To encode this particular constraint, the user must first select the tables concerned by the constraint. Here, we select the table containing locations from the network and deployment databases. The relation type must also be selected. *Bidirectional* means that all tuples from the source table must have a corresponding tuple in the target table and vice versa. *Left_to_right* means that only each left tuple must have a corresponding right tuple, whereas *Right_to_left* means the opposite.

Next, the way in which the different database tuples should be compared must be defined. The user must select the field needed to identify a corresponding tuple in the source and target table. Typically, this is done by indicating what field in the source and target table represents the identifier or key of that tuple. Sometimes, however, tuples from both tables cannot be directly matched. They can only be matched by using an intermediate table containing a correspondence between identifiers of the source and target table. In our example, because phones in the network table are identified using their MAC addresses, and in the deployment table using their UCL-ID [1], we need to use an intermediate table containing the mapping between MAC addresses and UCL-IDs. This *custom mapping* is encoded in a user-defined predicate MACTOUCLID. Such custom mappings can be defined straightforwardly by a user in a simple XML configuration file. Due to space limitations we refer to [8] for more details on how this is done.

After having selected the concerned tables, their respective keys, and optionally a key mapping, the user can now specify some conditions or constraints on the corresponding tuples between these tables. These conditions can be combined using logical conjunction or disjunction. In order to keep the user interface simple, for now the tool only allows to combine *all* individual conditions with either logical conjunction (ALL) or disjunction (ANY), but does not support a more fine-grained combination or nesting of logical operators. In our example, we define two conditions to encode the constraint that the building AND the room must be equivalent in the two tables. To do this, we require the corresponding fields in the source and target table, i.e. the fields 'building' and 'room', to be equal.

[1] The UCL-ID is a unique identifier given by UCL university to each phone in use at the university.

It is also possible to define some additional filters in order to consider only a subset of the data, for instance, only considering one particular building. This can be useful, for example, when analyzing very large databases with lots of inconsistencies, and the user wants to inspect the inconsistencies for a particular subset of the data only. In our particular example, we didn't apply any such filters.

Finally, when the user clicks on the 'Check constraint' button, three different *Alf* queries are generated. The first one is a query to find the positive results, i.e. all tuples that satisfy the declared constraint. A second query will calculate the mismatches in the source table, i.e. all tuples in the source table that do not satisfy the declared constraint. A third query calculates the mismatches in the target table.

The generated *Alf* query for the positive results looks somewhat like this:

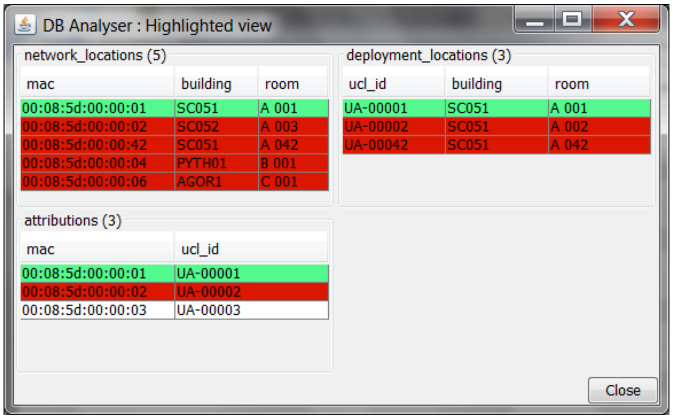
```
restrict(
 join_on(source_table, target_table,
         common_key),
 eq(:source_table_building,
    :target_table_building) &
 eq(:source_table_room,
    :target_table_room))
```

From this query it can be observed that the two concerned tables are first joined based on their common key, and then the results are restricted to the tuples satisfying all conditions, i.e. that the buildings and rooms must be equal. In reality, the actual generated query [2] is a bit more complex than this, to take into account custom mappings (in our example, for instance, there is no common key but the correspondence between MAC addresses and UCL ID's needs to be looked up in an intermediate table), renaming (for instance, when two corresponding fields have a different name in the different tables), and filters (an extra restriction based on the specified filters should be applied).

Each of these generated queries are then executed through *Alf*. As exemplified by Figure 1, positive results are displayed in the table at the bottom center of the GUI, whereas negative results are shown on the bottom left and right, respectively. (For non-bidirectional relations there will no table either on the left or on the right.)

In our example, we see that only one phone (the one with MAC address 00:08:5d:00:00:01 and UCL-ID UA00001) satisfies the constraint of having the same location in both sources. For all other phones, we find inconsistencies and they thus end up in the negative results. A negative result means that either the building or room was different in the other table, or that no correspondence whatsoever was found for this phone in the other table.

Whereas the presented positive and negative results already provide a lot of useful information about detected (in)consistencies in the data, they are not always easy to interpret by the end-user because they are not shown in the context of the original tables. For this purpose, our tool provides an alternative *highlighted view* which simply highlights the detected (in)consistencies in the original tables. To open this view it suffices to click on the button 'Highlighted view' at the bottom of the intensional view editor.



network_locations (5)

mac	building	room
00:08:5d:00:00:01	SC051	A 001
00:08:5d:00:00:02	SC052	A 003
00:08:5d:00:00:42	SC051	A 042
00:08:5d:00:00:04	PYTH01	B 001
00:08:5d:00:00:06	AGOR1	C 001

deployment_locations (3)

ucl_id	building	room
UA-00001	SC051	A 001
UA-00002	SC051	A 002
UA-00042	SC051	A 042

attributions (3)

mac	ucl_id
00:08:5d:00:00:01	UA-00001
00:08:5d:00:00:02	UA-00002
00:08:5d:00:00:03	UA-00003

Fig. 2. Inspecting data (in)consistencies with the *highlighted view*.

Figure 2 illustrates what this highlighted view would look like for our previous example. It displays each of the concerned tables, that is, the source and target tables but also the intermediate table used for defining the key mapping. For each of these tables the tuples are coloured either in red if they correspond to an inconsistency, in green if they correspond to a positive result, or just appear in white if the tuple is not concerned by this particular constraint.

In our example, we see that three tables are concerned. The locations from the network and from deployments, but also the intermediate attribution table which maps MAC addresses to phone IDs. The only positive case appears in green, all others in red. One element in the attributions table appears in white because no element in either the network or deployments table had such MAC address or UCL-ID.

Using the highlighted view we can observe, for instance, that the information for the phone with MAC address 00:08:5d:00:00:02 and UCL-ID UA00002 is inconsistent, since it appears with location SC052–A003 in the network table, whereas it has location SC051–A 002 in the deployments table.

IV. Validation

As explained above, intensional views allow the end-user to declare high-level constraints between data sources with relative ease, and reported inconsistencies can then be inspected in two different views to help him identify the causes of the inconsistencies.

In our actual case study, containing the data for about 6500 phones, many inconsistencies were found, such as missing phones, missing information for a given phone, missing mappings between phone IDs and their MAC address, and unknown buildings. All these inconsistencies can be found with our tool. The amount of phones dealt with and the amount of inconsistencies discovered were simply too large to be handled manually, which was the prime the reason for creating this intensional view tool for analyzing data inconsistencies.

For the constraint declared in the previous section, for example, when applied to the 6500 IP phones in use at the university, comparing locations in network and deployments

[2] More details on the query generation process can be found in [8].

returned 68% of inconsistent locations (either building or room). Only 13% of the 6500 phones had exactly the same location (building and room) in all sources. Whereas the underlying reasons for all this inconsistencies varied, an automated tool like ours to identify and inspect these errors was a crucial tool to start solving the inconsistencies.

Before putting our tool to use, the approach used was to merge all different data sources into one huge table. But this approach was infeasible due to the many inconsistencies between the data sources. Producing the merged table (which had to be done regularly because of the dynamic nature of some data sources) also took about 5 minutes, whereas keeping the data in their original sources but checking all regularities between them only took a few seconds. Another advantage of defining many different regularities against which to check conformance, was that it makes it easier to isolate and detect certain types of inconsistencies, as opposed to when having to discover inconsistencies in a huge merged database table.

V. Conclusion and future work

Our main contribution is the idea of combining intensional views with relational algebra, to address the problem of managing the structural consistency of an evolving information system. The intuitiveness and simplicity of intensional views match well with the expressive power of relational algebra.

The idea was implemented in an actual tool and put in practice on a non-trivial case study at a large Belgian university, where it is still in use today. The current implementation is only a first prototype, however, and many improvements can still be made. One of its most important limitations is probably that it can only express constraints between two tables, optionally connected through an intermediate table for mapping keys. While this proved to be sufficient for our case study, the tool could be extended so that constraints on multiple tables can be expressed, at the risk of making the GUI less intuitive to use. This is also one of the reasons why the original Intensional View Environment supports intensional relations over two intensional views only.

A similar remark holds for the combination of conditions. The tool currently allows to combine all conditions only with either a conjunction or a disjunction, but doesn't support a finer-grained combination or nesting of logical operators. This too was a deliberate choice because it sufficed for the constraints we needed to express on our case, and because it keeps the user interface simple.

The original Intensional View Environment also offers a notion of *alternative views*, which are useful to define interesting constraints on a single view. We could explore how to use this idea to define unary constraints on a given data source, for example to express that all phones in a given building should have a MAC address with a similar prefix.

We could also improve how custom mappings between tables are expressed. Currently, they are expressed in XML files and only allow mapping the field of one table to the field of another table. Allowing a user to define his own predicates directly in *Alf* would provide more expressiveness. However, this would create a strong dependency of our solution upon *Alf* and contradicts our goal of not requiring the user to know a particular query language. An alternative is to offer this possibility only to expert users, while providing to the average user a dedicated interface for creating custom mappings, which generates the necessary *Alf* query.

An open question remains upon what underlying language our tool should rely. We already motivated our choice for using relational algebra, and *Alf* in particular, but using logic instead (like the original intensional views approach), is possible too, as well as to directly generate SQL queries. Nevertheless, using *Alf* was a good choice from the point of view of ease of implementation and efficiency. It could be worthwhile exploring whether the original intensional view approach couldn't rely on relational algebra as well, instead of upon logic.

Our current approach does not allow to express constraints requiring aggregation, such as "An office cannot contain more than 10 phones". For this, we would need to use aggregation functions such as sum, count or avg. Such functions already exist in *Alf*, and could be added to the tool using some kind of quantifiers. The original intensional view approach did allow for quantification over views, but only universal and existential quantification, with its obvious interpretation in logic.

A further improvement requested by our end-users is to make the GUI more ergonomic. E.g., it could come with a wizard to help novice users define their constraints step by step. We should also add support to make it easy to find out, for a given data element, what constraints it does not satisfy. The original intensional view approach offered such support by integrating the tool in an existing development environment. By analogy we should provide a seamless integration of our current tool in an database management system.

The current paper is a nice case of cross-fertilisation research, where proven ideas of one domain (source code maintenance) are applied to another domain (database maintenance). A similar approach could even be used to any other domain dealing with structured information and suffering from problems due to implicit structural regularities not being respected upon evolution.

References

[1] K. Mens, B. Poll, and S. González, "Using intentional source-code views to aid software maintenance," in *International Conference on Software Maintenance (ICSM 2003)*. IEEE Computer Society, 2003, pp. 169–178.

[2] K. Mens and A. Kellens, "Towards a framework for testing structural source-code regularities," in *International Conference on Software Maintenance (ICSM 2005)*. IEEE Computer Society, 2005, pp. 679–682.

[3] K. Mens, A. Kellens, F. Pluquet, and R. Wuyts, "Co-evolving code and design with intensional views: A case study," *Computer Languages, Systems & Structures*, vol. 32, no. 2–3, pp. 140–156, 2006.

[4] M. M. Zloof, "Qbe/obe: a language for office and business automation," *Computer*, vol. 14, no. 5, pp. 13–22, 1981.

[5] R. Elmasri, *Fundamentals of database systems*. Pearson Education India, 2008.

[6] C. J. Date and H. Darwen, *Foundation for object/relational databases: the third manifesto: a detailed study of the impact of objects and type theory on the relational model of data including a comprehensive proposal for type inheritance*. Addison-wesley, 1998.

[7] B. Lambeau, "Alf, relational algebra at your fingertips," 2013. [Online]. Available: http://www.try-alf.org/ or http://github.com/alf-tool

[8] D. Colpaert, "Un outil de gestion d'incohérences de données basé sur les vues intensionnelles et l'algèbre relationnelle appliqué à un cas de localisation de téléphones IP," Master's thesis, Université catholique de Louvain, 2014.

Model Differencing for Textual DSLs

Riemer van Rozen
Amsterdam University of Applied Sciences (HvA)
Amsterdam, The Netherlands
Email: r.a.van.rozen@hva.nl

Tijs van der Storm
Centrum Wiskunde & Informatica (CWI)
Amsterdam, The Netherlands
Email: storm@cwi.nl

***Abstract*—The syntactic and semantic comparison of models is important for understanding and supporting their evolution. In this paper we present TMDIFF, a technique for semantically comparing models that are represented as text. TMDIFF incorporates the referential structure of a language, which is determined by symbolic names and language-specific scoping rules. Furthermore, it employs a novel technique for matching entities existing in source and target versions of a model, and finds entities that are added or removed. As a result, TMDIFF is fully language parametric, and brings the benefits of model differencing to textual languages.**

I. Introduction

Model differencing is a well-researched topic in the context of Model-Driven Engineering (MDE). For instance, the seminal paper by Alanen and Porres [1] introduced a generic algorithm to compute the difference and union between two models. In this paper we introduce Textual Model Diff (TMDIFF): an adaption of the Alanen and Porres algorithm for models represented as textual source code. Textual representation is common in the area of domain-specific languages (DSLs). We expect that TMDIFF will pose new opportunities to better understand and support the evolution of DSL programs in a similar way that the MDE process is supported by numerous tools for comparing, merging, and migrating models.

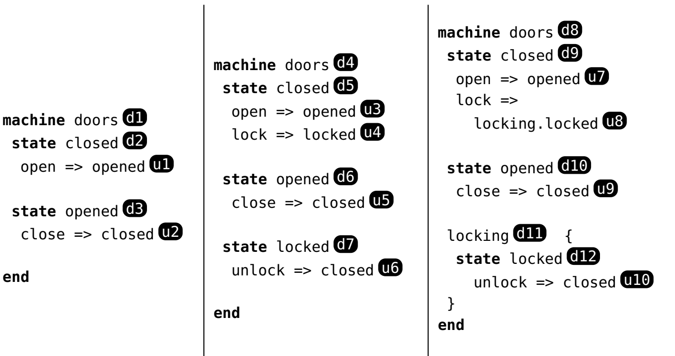
Applying model-based differencing techniques to textual models is non-trivial for two reasons. First, the referential structure of a textual model is encoded using symbolic names and language specific scoping rules. Second, textual languages are dependent on parsing for obtaining a structured representation. As a result, model elements do not have a stable identity across versions of a model.

TMDIFF addresses these problems as follows. First, TMDIFF is parameterized in the name binding semantics of the modeling language using a generic, relation-based representation of references. Second, the identities of entities across revisions of a model are recovered by aligning their defining name occurrences using stock `diff` algorithms (e.g.,[4]).

Below we present a motivating example based on textual state machine models. Then we present an overview of TMDIFF. We conclude with a discussion on limitations and future work.

II. Motivating Example

Figure 1 shows three versions of a textual model in a simple language for state machines. A state machine has a name and contains a number of state declarations. Each state declaration contains zero or more transitions. A transition fires on an event, and then transfers control to a new state. Figure 1a displays a state machine for controlling doors (*Doors*$_1$). The state machine is extended with a `locked` state in *Doors*$_2$ (Fig. 1b). The third version, *Doors*$_3$ (Fig. 1c), shows a grouping feature of the language: the `locked` state is part of the `locking` group. The grouping construct acts as a scope: it allows different states with the same name to coexist in the same state machine model.



```
machine doors d1
 state closed d2
  open => opened u1

 state opened d3
  close => closed u2

end
```

(a) *Doors*$_1$

```
machine doors d4
 state closed d5
  open => opened u3
  lock => locked u4

 state opened d6
  close => closed u5

 state locked d7
  unlock => closed u6

end
```

(b) *Doors*$_2$

```
machine doors d8
 state closed d9
  open => opened u7
  lock =>
    locking.locked u8

 state opened d10
  close => closed u9

 locking d11 {
  state locked d12
    unlock => closed u10
 }
end
```

(c) *Doors*$_3$

Fig. 1. Three versions of a simple state machine model. Definitions and uses of states are labeled with d_i and u_j respectively.

In each of the state machine models, the constructs that *define* entities are annotated with unique labels `dn`. For instance, in *Doors*$_1$, the machine itself is labeled `d1`, and both states `closed` and `opened` are labeled `d2` and `d3` respectively. Similarly, *uses* of states in transitions are labeled with labels `un`. For instance, the target state `opened` of the transition in `closed` is labeled `u1`.

To the human reader it is intuitively clear that states `closed` and `opened` are stable across revisions: only the relations between these states and other states are changed through the addition or change of transitions and addition of new states (`locked`). Textual or structural difference algorithms, however, are oblivious to the semantic identity of states and will generate spurious differences as a result.

TMDIFF does take into account constructs that represent semantic entities. It reports differences as imperative edit scripts in terms of a metamodel that is implicitly derived from the grammar of the language and its name binding semantics. For instance the difference between *Doors*$_1$ and *Doors*$_2$ is reported as:

```
create State d7                           //create State def
d7 = State("locked",[Trans("unlock",d2)]) //init new State
d2.out[1] = Trans("lock", d7)             //store 2nd Trans
d1.states[2] = d7                         //store new State
```

A new state d7 (locked) is created and initialized to contain a single transition to the (existing) state d2. Then the closed state gets a new transition to the state that was just created (d7). Finally, state d7 is added to the list of states of the state machine d1.

To illustrate the fact that TMDIFF can deal with scoping constructs, consider the difference between $Doors_2$ and $Doors_3$. Informally, the only thing that is changed is that the locked state is placed in a scope called locking. As a result, the reference to the locked state u3 in $Doors_2$ needs to be updated to use the qualified name locking.locked. However, semantically the transition structure between states does not change. The edit script produced by TMDIFF accurately reflects this description:

```
create Group d11                      //create Group def
d11 = Group("locking",[d7])           //initialize new Group
remove d4.states[2]                   //remove 3rd State
d4.states[2] = d11                    //store new Group
```

The script first creates the Group construct d11 and then initializes its name to locking and its owned states to contain a pointer to locked. Next, that state is removed from the list of states of the machine. Finally, the newly created group becomes the third element in this list. Everything else stays the same.

III. Overview of TMDIFF

TMDIFF is based on two relations: the reference relation between entities in a single model, and a matching relation between entities in different versions of the same model. We briefly describe each in turn.

a) Name Analysis: The user-specified name analysis should produce *reference graphs* in terms of definition and reference labels. A reference graph is triple $G = \langle D, U, R \rangle$, where D and U are sets of labels identifying definitions and uses respectively, and $R \subseteq (U \cup D) \times D$ is a binary relation representing references.

Figure 2 shows the abstract syntax tree (AST) and reference graph of $Doors_1$. The dashed arrows represent reference tuples in R. For instance, the Ref node (“opened”) is labeled u1 and refers to d3, the label of a Name node (“opened”).

The reference graph provides two important pieces of information: namely, the AST nodes representing definitions of entities, and nodes that are references to such entities.

b) Matching Entities: The matching process takes the textual source of both models, their ASTs and their reference graphs as input. It first creates *entity projections*, P_1 and P_2 which are sequences of tuples $\langle x, c, l, d \rangle$, where x is the symbolic name of the entity, c its semantic category (e.g. State, Machine, etc.), l the textual line it occurs on and d its definition label (e.g., d_1). For instance, The entity projections for $Doors_1$ and $Doors_2$ are as follows:

$$P_1 = \begin{array}{l} [\,\langle \texttt{doors},\ \mathit{Machine}, 1, d_1 \rangle, \\ \langle \texttt{closed}, \mathit{State},\quad 2, d_2 \rangle, \\ \langle \texttt{opened}, \mathit{State},\quad 5, d_3 \rangle\ ] \end{array} \quad P_2 = \begin{array}{l} [\,\langle \texttt{doors},\ \mathit{Machine}, 1, d_4 \rangle, \\ \langle \texttt{closed}, \mathit{State},\quad 2, d_5 \rangle, \\ \langle \texttt{opened}, \mathit{State},\quad 6, d_6 \rangle, \\ \langle \texttt{locked}, \mathit{State},\quad 9, d_7 \rangle\ ] \end{array}$$

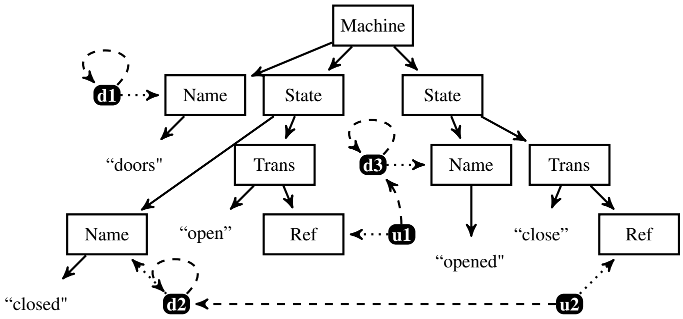


Fig. 2. AST with references of $Doors_1$. Solid arrows indicate containment in the AST. Dotted lines associate unique labels to AST nodes, and dashed lines are references.

The line numbers in the entity projections provide the key to using a traditional textual diff to determine whether an entity has been added or not. Recall that diff produces a patch describing which lines we added or removed. In the context of $Doors_1$ and $Doors_2$, for instance, the first three entries in both P_1 and P_2 all have line numbers that are not changed by the diff. Therefore the labels d_1, d_2, d_3 and d_4, d_5, d_6 are pairwise matched. Entity d_7 however was defined on line 9, and this is one of the lines marked as added by the textual diff. As result, d_7 is considered to represent a newly created entity. Recovering deleted entities works the other way round.

The reference graph provides the information on which nodes are actually semantic entities, and how entities refer to each other. Entity matching determines which entities exist in both revisions of a textual model. Together the reference graph and the entity matching represent the necessary information for applying existing model differencing algorithms such as [1].

IV. Discussion and Outlook

We have implemented a prototype of TMDIFF in Rascal, a meta programming language and environment for source code analysis and transformation [2]. As an initial experiment, we were able to reconstruct the complete history of file description models used in a DSL for digital forensics [5].

Future work is aimed at assessing how our diff-based matching strategy compares to existing approaches [3]. In particular, our strategy is not resilient against moving around of definitions, since traditional diff will not detect them as such. We are also investigating how our generic textual model differences can be used for reconciling co-evolving artifacts and migrating run-time states of textual DSL programs.

References

[1] M. Alanen and I. Porres. Difference and Union of Models. In *UML*, pages 2–17, 2003.

[2] P. Klint, T. van der Storm, and J. J. Vinju. RASCAL: A domain specific language for source code analysis and manipulation. In *SCAM*, pages 168–177. IEEE, 2009.

[3] D. S. Kolovos, D. Di Ruscio, A. Pierantonio, and R. F. Paige. Different Models for Model Matching: An Analysis of Approaches to Support Model Differencing. CVSM ’09, pages 1–6, Washington, DC, USA, 2009. IEEE Computer Society.

[4] W. Miller and E. W. Myers. A File Comparison Program. *Softw. Pract. Exper.*, 15(11):1025–1040, 1985.

[5] J. van den Bos and T. van der Storm. A Case Study in Evidence-based DSL Evolution. In *ECMFA’13*, pages 207–219. Springer, 2013.

A Static Concept Location Technique for Data-Intensive Systems: „Where Was This SQL Query Executed?”

Csaba Nagy, Anthony Cleve
PReCISE Research Center, University of Namur, Belgium
{csaba.nagy, anthony.cleve}@unamur.be

1 Introduction

An evolving software system is incrementally modified, changed by its developers during the development and maintenance phases [1]. Before the developers start working on a change they need to identify which parts of the source code implement the feature, and should be touched first during the change. In practice, what they do is a concept location task (also known as feature identification/location) which is „*the process that identifies where a software system implements a specific concept*” [2].

There are many existing approaches to support developers in concept location tasks starting from simple pattern matching (so-called ‘grep’ techniques) to more sophisticated methods like IR-based techniques or dependency analyzes [3]. However, none of the existing approaches consider when there is a database in the architecture, which adds further source artifacts or dependencies.

Here, we investigate a concept location approach for data-intensive systems, as applications with at least one database server in their architecture which is intensively used by its clients. Specifically, we introduce a static technique to identify the location(s) in the source code where a given SQL query was potentially sent to the database server.

2 Motivation

Identifying the location in the source code where a given SQL query was sent to the database is a regular debugging task of data-intensive systems. Typical scenarios are when queries need to be optimized for performance, or when they cause failures (e.g. a syntactic error or a deadlock issue). Complexity of the system or the use of ORM technologies can even complicate this tasks.

With dynamic analysis, it is possible to trace the query on the database side or on the client side too. At the database this is usually just a logging configuration, while on the client side they usually exploit that SQL queries are sent to the server via certain API calls which can be wrapped or hooked to catch the query.

However, dynamic analysis cannot help us in some situations. Suppose, that the user of the application experiences performance issues at the database; he identifies the query which causes the performance drop back in the log files of the database and sends us a bug report. Since the problem occurred at the database and was reported by it (client was not directly affected), we do not have a stack trace in the bug report. How can we spot out then, where the query was prepared in the source code? We must reproduce everything exactly as the user did which might be even impossible if we depend on the data stored in the database (perhaps we cannot even ask it for privacy reasons). In such situations, a static approach could provide us a great help in the concept location task.

3 Approach

Our approach, to identify the location in the source code where a given SQL query was sent to the database server, can be divided into three main steps (see Figure 1):

1. We extract the embedded SQLs from the source files with a technique which substitutes unrecognized code fragments with special identifiers [4]. The output of this step is a set of embedded SQLs which are prepared in the client code and potentially sent to the database.

2. We parse the extracted queries with a robust parser (which is able to handle the unrecognized code fragments). In the same step, we parse the database schema as well, and the queries that we are actually looking for. The output of the parser is an ASG (Abstract Syntax Graph) containing all the SQL statements.

3. We run a sort of tree-matching algorithm on the ASG to identify subtrees (queries) matching the trees of those statements that we are looking for. The output is a set of source code positions where the queries were potentially sent to the database server.

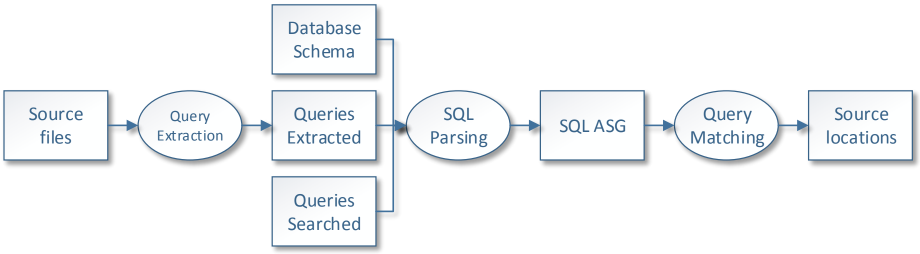


Figure 1: Overview of the approach

4 Current Results and Future Plans

We implemented our approach for systems written in Java and accessing the database through JDBC and/or Hibernate. To validate our approach, we test the implementation on the open source OSCAR EMR Clinical Management System. OSCAR accesses the database through JDBC and Hibernate and as a system with about 400 kLOC working with more than 400 tables, it is perfect to demonstrate the possibilities of our approach. Currently, we are in implementation/testing phases and able to extract JDBC queries, parse them supporting MySQL dialect and match the extracted queries to the concrete ones. We plan to extend our approach to handle Hibernate as well, where a key challenge is that a query can be specified as an HQL or Criteria query too, which is then compiled to the query language of the database server.

References

[1] V. Rajlich and P. Gosavi, "Incremental change in object-oriented programming," *IEEE Softw.*, vol. 21, no. 4, pp. 62–69, Jul. 2004.

[2] A. Marcus, V. Rajlich, J. Buchta, M. Petrenko, and A. Sergeyev, "Static techniques for concept location in object-oriented code," in *Proc. of the 13th International Workshop on Program Comprehension (IWPC'05)*. IEEE Computer Society, 2005, pp. 33–42.

[3] B. Dit, M. Revelle, M. Gethers, and D. Poshyvanyk, "Feature location in source code: a taxonomy and survey," *Journal of Software: Evolution and Process*, vol. 25, no. 1, pp. 53–95, 2013.

[4] L. Meurice, J. Bermudez, J. Weber, and A. Cleve, "Establishing referential integrity in legacy information systems - reality bites!" in *Proc. of 30th Interntional Conference on Software Maintenance and Evolution (ICSME)*. Victoria, BC, Canada: IEEE Computer Society, Oct. 2014.

A Template Method for Improvement of System Response Time

Authors: Rick Hoving, AFAS, Jan Martijn E.M. van der Werf, and Slinger Jansen, Utrecht University

With the constant improvement of computers and devices, the webclient in a client-server architecture can perform large portions of logic typically residing at server-side. When an architect chooses to upload and execute business logic on the webclient, the webserver requires fewer resources. Fewer resources result in a smaller server landscape. Although this sounds promising, the requirements thus placed on the webclient introduce many new challenges. Because of the wide variety in possible devices, an architect cannot predict the specific device on which the application runs. When the application performs more logic on the webclient, the user's device increasingly influences the user experience.

To provide the architect with information on the user experience of such a web application, we further elaborate on these characteristics. These characteristics on the client include Asynchronous Javascript And XML and the Document Object Model. Web applications use Asynchronous Javascript And XML for communication between client and server. A web application uses the Document Object Model to change the HTML document of the web page to communicate with the user. Because of the wide variety of possible devices, the architect cannot predict the specific device on which the user interacts with the application. However, the architect can measure the user experience when the user interacts with the application. To provide the architect with these measurements, we utilize the software operation knowledge framework.

As a metric of user experience, user-perceived latency heavily influences the client. Using the steps of human computer interaction depicted in Figure **??**, we provide with a formal definition of user-perceived latency. In user-perceived latency we define two types of actors, the user and the system. The architecture of the application cannot influence the user. However, the architecture can be used to influence the system. Therefore we divide user-perceived latency into two components, human interaction time (HIT) and system response time (SRT). During the human interaction time, the user interacts with the system. During the system response time, the system reacts to the users input, see Figure **??**. Because of our architectural view, we focus on the system response time. User experience is an element used to describe the quality of the system. Because the system response time is not a property of the application, we see the system response time as a metric.

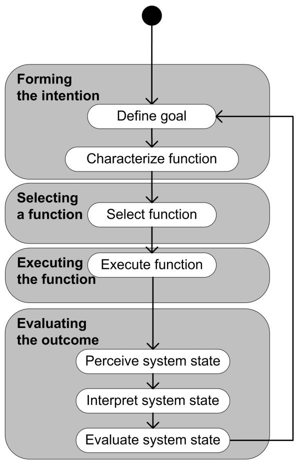


Figure 1: The Four Stages of Human Computer Interaction

When measuring the system response time, architectural erosion and architectural degradation result in uncertainties for the architect. These uncertainties result in the architect unable to predict the effect of changes in the software's architecture on the system response time. To measure the system response time and cope with these uncertainties, we create a framework based on the software operation knowledge framework. To cope with architectural erosion and architectural degradation, the software operation knowledge framework fit for system response time measurements includes a simulation step. The architect uses the simulation step to reproduce the system response time. The architect uses the reproduced system response time to predict the effect of architectural changes on the system response time with more accuracy.

We use the adjusted framework to provide the Template Method for Improvement of System Response Time. The method aims at gaining knowledge about the system response time and the effects of changes in the software's architecture on the system response time. The method aids in measuring, improving, and simulating the system response time. To improve the system response time, we perform data mining techniques and process mining techniques onto the obtained data. The architect uses presentation means to visualize the data and identify improvements in the architecture. After the architect improves the software's architecture and the software, they create and utilize a test environment to obtain an accurate prediction of the effect of these changes on the system response time.

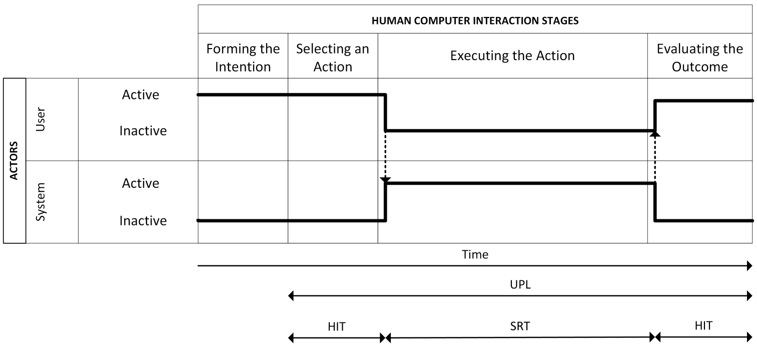


Figure 2: A Visualization of the Human Computer Interaction Process