SATToSE 2014 — Pre-proceedings

Advanced Techniques and Tools for Software Evolution

Extended Abstracts

	Wednesday 9 July 2014	Thursday 10 July 2014	Friday 11 July 2014
8:00–9:00	Opening		
9:00–10:00	Invited talk: Alexander Serebrenik	Invited talk: Tanja Vos	Invited talk: Marianne Huchard
10:00–10:30	Coffee and tea	Coffee and tea	Coffee and tea
10:30–12:00	Mining Session: #5, #4, #15, #20, #11	Analysis Session: #18, #9, #1, #7, #16, #17	Visualisation Session: #8, #2, #12
12:00–12:30			Hackathon results
12:30–13:30	Lunch	Lunch	Lunch
13:30–14:30	Transformation Session: #14, #10, #13	Tutorial: Anya Helene Bagge	Invited talk: Alfonso Pierantonio
14:30–15:00		Technology Showdown	Hackathon results
15:00–15:30	Hackathon intro		Metrics Session: #3, #6
15:30–16:00	Coffee and tea	Coffee and tea	
16:00–17:00	Invited talk: Leon Moonen	Social event	Coffee and tea
17:00–18:30	City tour		Closing
18:30–22:00	Hackathon		

Seminar Series, SATToSE 2014
L'Aquila, Italy, 9–11 July, 2014

Dipartimento di Informatica
Università degli Studi dell'Aquila
I–67100 L'Aquila, Italy

Vadim Zaytsev (Ed.)

Preface

SATToSE is the Seminar Series on Advanced Techniques & Tools for Software Evolution. The goal of SATToSE is to gather both undergraduate and graduate students to showcase their research, exchange ideas, improve their communication skills and attend technical sessions.

The 7th edition of SATToSE took place in L'Aquila (Italy) on 9–11 July 2014. Past editions of SATToSE saw presentations on software visualisation techniques, tools for co-evolving various software artefacts, their consistency management, run-time adaptability and context-awareness, as well as empirical results about software evolution. The attendees of SATToSE 2014 enjoyed 5 invited speakers, 19 technical presentations, a tutorial, technology showdown and hackathon sessions, as well as lively social events.

Many people contributed to the success of SATToSE 2014. I would like to truly acknowledge the Steering Committee that gave us the possibility to organise the 7th edition of the event in L'Aquila. I am also indebted to Vadim Zaytsev that managed to arrange a strong program, to the invited speakers and to the students themselves that contributed to make SATToSE 2014 a remarkable experience. Last but not least I would like to thank Ludovico Iovino and Romina Eramo that contributed to the local organisation.

9–11 July 2014

Davide Di Ruscio
SATToSE 2014

Organisation

SATToSE 2014 is hosted by Department of Information Engineering Computer Science and Mathematics (DISIM) of the University of L'Aquila, Italy.

General Chair	Davide Di Ruscio (University of L'Aquila, Italy)
Program Chair	Vadim Zaytsev (University of Amsterdam, The Netherlands)
Hackathon Chair	Alexander Serebrenik (Eindhoven University of Technology, The Netherlands)
Local organisation	Romina Eramo (University of L'Aquila, Italy) Ludovico Iovino (University of L'Aquila, Italy)

Steering Committee

Ralf Lämmel	(University of Koblenz-Landau)
Michael W. Godfrey	(University of Waterloo)
Marianne Huchard	(University of Montpellier 2)
Tom Mens	(University of Mons)
Oscar Nierstrasz	(University of Bern)
Coen De Roever	(Free University Brussels)
Vadim Zaytsev	(University of Amsterdam)

http://sattose.org

Contents

Lecture Abstracts

Presentation Abstracts

Mining

Transformations

Lecture Abstracts

Human Aspects of Software Engineering
Invited Talk Abstract

Alexander Serebrenik

Software Engineering and Technology
Faculteit Wiskunde en Informatica
Technische Universiteit Eindhoven
The Netherlands
a.serebrenik@tue.nl

Software engineering is inherently a collaborative venture, involving many stakeholders that coordinate their efforts to produce large software systems. While importance of human aspects in software engineering has been recognised already in the 1970s, emergence of open source software (late 1990s) and platforms such as Stack Overflow and GitHub (late 2000s) enabled application of empirical methods to study of human aspects of software engineering.

In the first part of the talk we present a selection of recent results pertaining to two main questions: who are the software developers and in what kind of activities they engage. The second part of the talk focuses on tools and techniques that have been used to obtain the aforementioned results.

Slides: http://www.slideshare.net/aserebrenik/sattose-talk

Assessment and Evolution of Safety-Critical Cyber-Physical Product Families

Invited Talk Abstract

Leon Moonen

Simula Research Laboratory
Norway
leon.moonen@computer.org

The research presented in this talk is part of an ongoing industrial collaboration with Kongsberg Maritime (KM), one of the largest suppliers of maritime systems worldwide. The division that we work with specialises in computerised systems for safety monitoring and automatic corrective actions on unacceptable hazardous situations. The overall goal of the collaboration is to provide our partner with software analysis tooling that provides source based evidence to support cost-effective software certification of evolving systems. In particular, we study a family of complex safety-critical embedded software systems that connect software control components to physical sensors and mechanical actuators.

A frequently advocated approach to manage the development of such complex software systems is to compose them from reusable components, instead of starting from scratch. Components may be implemented in different programming languages and are tied together using configuration files, or glue code, defining instantiation, initialisation and interconnections. Although correctly engineering the composition and configuration of components is crucial for the overall behaviour, there is surprisingly little support for incorporating this information in the static verification and validation of these systems. Analysing the properties of programs within closed code boundaries has been studied for some decades and is well-established.

Moreover, sharing components between software products introduces dependencies that complicate maintenance and evolution: changes made in a component to address an issue in one product may have undesirable effects on other products in which the same component is used. Therefore, developers not only need to understand how a proposed change will impact the component and product at hand; they also need to understand how it affects the whole product family, including systems that are already deployed. Given that these systems contain thousands of components, it is no surprise that it is hard to reason about the impact on a single product, let alone on a complete product family. Conventional impact analysis techniques do not suffice for large-scale software-intensive systems and highly populated product families, and engineers need better support to conduct these tasks.

In the talk, we will discuss the techniques we developed to support analysis *across* the components of a heterogeneous component-based system. We build upon OMG's Knowledge Discovery Metamodel to reverse engineer fine-grained

homogeneous models for systems composed of heterogeneous artifacts. Next, we track the information flow in these models using slicing, and apply several transformations that enable us to visualise the information flow at various levels of abstraction, trading off between scope and detail and aimed to serve both safety domain experts as well as developers. These techniques are implemented in a prototype tool-set that has been successfully used to answer software certification questions of our industrial partner. In addition, we discuss our ongoing research to build recommendation technology that supports engineers with the evolution of families of safety-critical, software-intensive systems. This technology builds on extensions of the previously discussed techniques to systematically reverse engineer abstract representations of software products to complete software product families, new algorithms to conduct scalable and precise change impact analysis (CIA) on such representations, and recommendation technology that uses the CIA results and constraint programming to find an evolution strategy that minimises re- certification efforts.

Test Automation at the User Interface Level
Invited Talk Abstract

Tanja E. J. Vos

Departamento de Sistemas Informaticos y Computación
Universidad Politecnica de Valencia
Spain
tvos@dsic.upv.es

Testing applications at the UI level is an important yet expensive and labour-intensive activity. Several tools exist to automate UI level testing. Capture replay tools rely on the UI structure and require substantial programming skills and effort. These tools implicitly make the assumption that the UI structure remains stable during software evolution and that such structure can be used effectively to anchor the UI interactions expressed in the test cases. Consequently, when test cases are evolved, adapted, parameterized or generalized to new scenarios, the maintenance cost can get real high and the competence required from programmers can become an obstacle. Visual testing tools take advantage of image processing algorithms to simulate the operations carried out manually by testers on the UI making UI testing as simple as that carried out step by step by humans. These visual testing approaches simplify the work of testers as compared to the structural testing approaches. However, they do rely on the stability of the graphical appearance of the UI, and require substantial computational resources for image processing. Changes to the application often also involve changes to the UI, hence also threatening the visual approach. Visual clues in the UI might mislead the image recognizer of visual testing tools, which are correspondingly subject to false positives (wrong UI element identification) and false negatives (missed UI elements). In this talk we will present the tool TESTAR, whose development was initiated under the FITTEST project. TESTAR automatically generates and executes test cases based on a structure that is automatically derived from the UI through the accessibility API. Since this structure is build automatically during testing, the UI is not assumed to be fixed and tests will run even though the UI evolves, which will reduce the maintenance problem that threatens the approaches mentioned earlier. The basic functionality of the tool will be presented, together with some case studies done in industry to evaluate the approach.

Relational Concept Analysis: Mining Multi-relational Datasets for Assisted Class Model Evolution

Invited Talk Abstract

Marianne Huchard

Université Montpellier 2 — Faculté Des Sciences
Montpellier, France
marianne.huchard@lirmm.fr

Formal Concept Analysis is a well established framework for extracting an ordered set of concepts from a dataset, called a Formal Context, composed of entities described by characteristics. This data analysis framework is currently applied to support various tasks, including information retrieval, data mining, or ontology alignment. It also has many applications in software engineering such as software understanding, extracting or maintaining class models, or software reengineering.

Relational Concept Analysis (RCA) is an extension of the FCA framework to multi-relational datasets, namely datasets composed of several categories of entities described by both characteristics and inter-entities links. RCA generates a set of concept lattices, precisely one for each category of entity. The concepts are connected via "relational attributes" that are abstractions of the initial links and traverse the lattice frontier. The concept lattice set is a particular view on the dataset, which reveals implication rules involving relationships, as well as relevant connections between classified groups of entities. In this talk, we introduce RCA and we explain its strengths and limits. Then we develop an exploratory approach for assisting a domain expert in class model evolution, more precisely for the class model building and for the follow-up of the class model abstraction level.

Non-determinism and Bidirectional Model Transformations

Invited Talk Abstract

Alfonso Pierantonio

Dipartimento di Ingegneria e Scienze dell'Informazione e Matematica
Università degli Studi dell'Aquila
Italy
alfonso.pierantonio@univaq.it

In Model-Driven Engineering (MDE), the potential advantages of using bidirectional transformations in various scenarios are largely recognized. Despite its relevance, bidirectional languages have rarely produced anticipated benefits. In explanation of this difficulty different arguments can be provided, in particular it has been widely discussed how bidirectional transformations need not be bijective. Any transformation which is not injective is non-deterministic, in the sense that more than one admissible solution is in principle possible. Unfortunately, most of the current languages generate only one model at time, possibly not the desired one, without the possibility to specify an update policy. This talk discusses these issues and tries to outline what are the challenges and the possible solutions.

Language, Models and Megamodels
Tutorial Abstract

Anya Helene Bagge

Institutt for Informatikk
Universitetet i Bergen
Norway,
anya@ii.uib.no

A *model* is an abstraction of a system, made to facilitate some kind of understanding or processing by humans or computers. Abstracting away from details allows us to deal with systems that would otherwise be too large or too complicated to deal with.

Models and modelling are of particular importance in software engineering for two reasons: First, the software itself is often meant to model some system; and second, software systems tend to be so large and complicated that they are themselves in need of modelling as part of the development process.

Modelling of systems and modelling of software naturally brings us to the modelling of modelling and of models. While *metamodels* are models of modelling languages (i.e., the abstractions used to describe models), a *megamodel* is a model of a system of models (i.e., a model where the elements are models). The design of software systems may involve many models, metamodels and artifacts – all of these can be captured in a megamodel, together with the relationships between them. The details of the individual models are abstracted away, leaving a bird's eye view of the roles and relationships of the models.

This tutorial will explain megamodelling in practice, with a particular focus on examples from software language engineering.

Presentation Abstracts

Co-Occurring Code Critics

Angela Lozano, Gabriela Arévalo, and Kim Mens

Vrije Universiteit Brussel
Pleinlaan 2
Brussels, Belgium
alozano@soft.vub.ac.be

Universidad Nacional de Quilmes
Roque Sáenz Peña 352
Buenos Aires, Argentina
gabriela.b.arevalo@gmail.com

Université Catholique de Louvain
Place Sainte Barbe 2
Louvain-la-Neuve, Belgium
kim.mens@uclouvain.be

Abstract. Code critics are controversial implementation choices (such as bad smells or code smells) but at a higher level of detail. Code critics are a recommendation facility of the Smalltalk-Pharo IDE. They aim to achieve standard idioms which allow for a better performance or for a better use of object-oriented building mechanisms. Code critics can be identified at the method- or class-level. We are analyzing in several applications which code critics tend to occur in the same source code entity to see to what extent it is possible to identify controversial implementation choices at a higher level of abstraction.

Keywords: Bad smells, code smells, code critics, Smalltalk, Pharo, empirical software engineering, co-occurrence

1 Introduction

In the context of design and coding any application, there is a plethora of implementation recommendations. Some of these recommendations are given at a high level of abstraction (e.g., low coupling and high cohesion) or while others are very specific level (e.g., classes should not have more than 6 methods). Most of the research on recommending the elimination of controversial implementation choices uses a top-down approach. That is, the recommendations given at a high level of abstraction are disassembled into concrete symptoms until a straightforward detection strategy is reached (e.g., Marinescu's design flaws [2], Gueheneuc/Moha's antipatterns [3]). However, it is difficult to argue that those concrete detection strategies and the way in which they are combined represent *all and only those entities* that the high level recommendation aims to convey. We propose to extract high-level recommendations from the analysis of specific recommendations. The recommendations extracted would not be affected by different interpretations (as opposed to the approaches to detect Fowler's bad smells[1] which differ on heuristics[1], metrics[2] and thresholds[3]). Moreover, this study would allows us to validate the need of the specific recommendations analyzed.

[1] Heuristics are incomplete by definition

[2] The definition of some metrics are also open to interpretation resulting in different tools that provide different results for the same metric.

[3] Thresholds tend to be absolute values that cannot be used across applications or relative values whose cut point is arbitrary.

1.1 Code critics

Code critics is a list of implementation choices in Smalltalk known for being 'ungraceful'. These critics may point out at defects or performance issues in the code. There are code critics only for methods or for classes. Each critic has a short name and a description that explains why that implementation choice could be harmful. In some cases, the code critic also proposes a refactoring.

Although code critics may contain many false positives, the IDE allows to 'turn off' manually any result. The results that have been turned off are saved within the image so that the developer does not have to browse the same false positive ever again. Each code critic belongs to a category that indicates its harmfulness. The categories are Unclassified, Style, Idioms, Optimization, Design Flaws, Potential Bugs, and Bugs.

2 Data collected

Although the code critics tool is designed to analyze the whole image on a selection of critics, as anything else in Smalltalk it can be run programmatically. We analyze all critics implemented except Spelling rules [4](i.e., 120 code critics) from which 27 apply to classes, and 93 apply to methods. We analyzed all packages contained in the image used for the latest distribution of Moose (i.e., Pharo 1.4). For each package (71 in total) we find all critics in methods/classes except for those that implement tests[5]. The result of this analysis is a set of boolean tables (two tables per package: one for its methods and another one for its classes) that indicate which source code entities had which critics (each row has a code critic while each column has a method or class of the package) (shown in Table 1).

These boolean tables are converted into distance tables that measured to what extent the entities affected by one code critic are also affected by another code critic. The distance between two code-critics are calculated by counting the number of source code entities (classes or methods) that were affected by only one of them, over the number of source code entities (classes or methods) that any of them affected. For instance, the distance between cc1 and cc2 is 1 (first cell in Table 2) because their results differ for two classes. We see in Table 1 that cc2 affected only class2, while cc1 affected only class3, out of the two classes that were affected by any of these critics: class2 and class3.

Based on the boolean Table (shown in Table 1) and the distance table (shown in Table 2) we proceed to discard pairs of critics that do not seem interesting

[4] Spelling rules check the spelling on the identifiers of classes, methods, variables, and comments. Given that these violations do not refer to the structure or design of the the source code we discard them because they are likely to generate noise in the results, and are non-critical for software development.

[5] Tests were excluded because critics to their code are likely to be false positives. For instance, duplicated code (which may occur due to calls to assert or other testing methods) does not necessarily create hidden links to other test methods. Moreover, test code tends to contain trial-and-error code which does not follow standard coding practices.

	class1	class2	class3
cc1	0	0	1
cc2	0	1	0
cc3	0	1	1
cc4	1	0	0
cc5	1	0	1
cc6	1	1	0
cc7	1	1	1

Table 1. Code critics (cc) per class for a fictitious package.

	cc1	cc2	cc3	cc4	cc5	cc6
cc2	1.0	-	-	-	-	-
cc3	0.5	0.5	-	-	-	-
cc4	1.0	1.0	1.0	-	-	-
cc5	0.5	1.0	0.6	0.5	-	-
cc6	1.0	0.5	0.6	0.5	0.6	-
cc7	0.6	0.6	0.3	0.6	0.3	0.3

Table 2. Distance among code critics shown in Table 1

for our analysis. Three criteria are used to discard pairs of code-critics. First, pairs with high distances (greater than 0.9) are discarded as they do not tend to co-occur and therefore are unlikely to represent a recommendation of a higher level of abstraction. Second, pairs that occur *always* in the same source code entities because they are likely to be different implementations of the same code critic. Third, all pairs for which one of the code-critics covers more than 90% of the source code entities analyzed because they will be automatically correlated with all other code-critics and these relations are likely to generate only noise.

3 Results

We have generated graphs depicting the frequency in which code-critics appear and the strength of their relation. The strength of the relation between a pairs of critics is defined by its frequency (i.e., number of packages where the pair of critics appears) and by the average of their distance (i.e., distance between a pair of code-critics for all packages analyzed). The pairs with lowest distance and highest frequency are analyzed first because they might reveal an undesirable implementation pattern of a higher level of abstraction. So far we have identified four patterns of relations between pairs of code critics. The first pattern occurs when the critics are redundant. This happens when the critics find similar problems. In particular the following pairs refer to code critics in which only the first one provides a refactoring for the critique:

- '*detect:ifNone: − > anySatisfy:*' vs. '*Uses detect:ifNone: instead of contains:*'
- '*Replace with allSatsify:, anySatisfy: or noneSatsify:*' vs. '*Uses do: instead of contains: or detect:'s*'

– *Rewrite super messages to self messages when both refer to same method*' vs. '*Sends different super message*'

The second pattern occurs when the critics positively contribute to another one, without being an implication (i.e., solving one critic does not solve the other). This happens when the critics have a common root cause. For instance,

– '*Excessive number of variables*' vs. '*Excessive number of methods*'
– '*Sends questionable message*' vs. '*Excessive number of methods*'

The third pattern occurs when the both critics need to be refactored. This happens when the critics have a common root cause. For instance,

– '*Instance variables not read AND written*'[6] vs. '*Variables not referenced*'[7]
– '*Subclass responsibility not defined*'[8] vs. '*References an abstract class*'[9]

The last pattern occurs when the code critics occur often together but they are more useful as a separate rule (as it is more specific). For instance, '*Inconsistent method classification*' vs. '*Unclassified methods*' which would be more useful as '*inconsistently unclassified methods*'.

Acknowledgments. Angela Lozano is financed by the CHaQ project of the Innovatie door Weteschap en Tecnologie.

References

1. M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, 1999.
2. R. Marinescu. Detecting design flaws via metrics in object oriented systems. In *Proc. of the Technology of Object-Oriented Languages and Systems (TOOLS)*, pages 173–182. 2001.
3. N. Moha, Y.-G. Gueheneuc, and P. Leduc. Automatic generation of detection algorithms for design defects. In *Proc. of the Int'l Conf. on Automated Software Engineering (ASE)*, pages 297–300. IEEE Computer Society, 2006.

[6] This critic should be divided to discriminate between variables readOnly, writeOnly, or notReferenced
[7] This critic hould refer to class variables only (instance variables would be caught by the noReferenced subcritic of the other code critic)
[8] This critic should be refined because as long as all leafs can see an implementation of the method it should not be a bad-smell.
[9] This critic should be refined into 'refers to an abstract class' and 'uses it as instance or with isKindOf':.

QwalKeko, a History Querying Tool

Reinout Stevens[1] and Coen De Roover[2]

[1] resteven@vub.ac.be Software Languages Lab, Vrije Universiteit Brussels, Belgium
[2] cderoove@vub.ac.be Software Engineering Laboratory, Osaka University, Japan

In this paper we discuss the inner working of QwalKeko[3][5], a history querying tool that allows querying Java projects stored in Git. The tool allows users to more easily query software projects and to reproduce studies on different software projects. To this end, QwalKeko converts a Git repository into a graph of versions, in which each version corresponds to the state of the stored software project after a particular commit. Successive versions are connected directly. This graph can be queried by using Qwal[4], an implementation of regular path expressions and Ekeko[5][3], a logic programming query language.

1 Ekeko

Ekeko is a Clojure library for applicative logic meta-programming against an Eclipse workspace. Ekeko has been applied successfully to answering program queries (e.g., "does this bug pattern occur in my code?"), to analyzing project corpora (e.g., "how often does this API usage pattern occur in this corpus?"), and to transforming programs (e.g., "change occurrences of this pattern as follows") in a declarative manner.

Ekeko provides a library of predicates that can be used to query programs. These predicates reify the basic structural, control flow and data flow relations of the queried Eclipse projects, as well as higher-level relations that are derived from the basic ones.

We limit our discussion to those predicates that reify structural relations computed from the Eclipse JDT. Binary predicate `(ast ?kind ?node)`, for instance, reifies the relation of all AST nodes of a particular type. Here, `?kind` is a Clojure keyword denoting the capitalized, unqualified name of `?node`'s class. Solutions to the query `(ekeko [?inv] (ast :MethodInvocation ?inv))` therefore comprise all method invocations in the source code.

Ternary predicate `(has ?propertyname ?node ?value)` reifies the relation between an AST node and the value of one of its properties. Here, `?propertyname` is a Clojure keyword denoting the decapitalized name of the property's `org.eclipse.jdt.core.dom.PropertyDescriptor` (e.g., `:modifiers`). In general, `?value` is either another `ASTNode` or a wrapper for primitive values and collections. This wrapper ensures the relationality of the predicate.

[3] https://github.com/ReinoutStevens/damp.qwalkeko
[4] https://github.com/ReinoutStevens/damp.qwal
[5] https://github.com/cderoove/damp.ekeko

2 Qwal

Qwal enables querying graphs using regular path expressions. Regular path expressions are an intuitive formalism for quantifying over the paths through a graph. They are akin to regular expressions, except that they consist of logic conditions to which regular expression operators have been applied. Rather than matching a sequence of characters in a string, they match paths through a graph along which their conditions holds.

A Qwal query is launched using the function `(qwal graph begin ?end [ & vars ] & goals)`. It takes as arguments a graph object, a begin node, a logical variable that is unified with the end node of the expression, a vector of local variables available inside the query and an arbitrary amount of goals. A graph object is a map that contains two logical rules `predecessors/2` and `successors/2`, which are called with a bound first argument and bind the second argument to either the predecessors or successors of the first argument. The goals in a query either specify conditions that must hold in the current node of the query, or they move throughout the graph changing the node against which conditions are checked. We provide an excerpt of the available goals:

`q=>` Moves the current version to one of its successors.
`q<=` Moves the current version to one of its predecessors.
`(qin-current & conditions)` Conditions need to hold in the current version. Conditions are regular `core.logic` predicates.
`(q* & goals)` Goals succeed an arbitrary, including zero, amount of times.
`(q=>* & goals)` Similar to `q*`, except an implicit `q=>` is added after goals. If goals is empty this skips an arbitrary number of nodes.

Every goal is either a function or a macro that returns a logic rule. Such a rule takes three arguments, namely the graph, the current node and an unbound variable that must be bound to the new current node of the query. Users can define their own goals by defining functions that adhere to this protocol.

3 QwalKeko

QwalKeko uses Qwal to navigate through a graph of versions, and Ekeko to specify what conditions need to hold in a particular version. QwalKeko features its own set of Qwal and Ekeko predicates that only make sense in the context of history querying.

QwalKeko defines the following two Qwal goals: `(in-git-info [c] & conditions)` and `(in-source-code [c] & conditions)`. Both evaluate conditions in the current version `c`. `in-git-info` only allows conditions that reason over information that can be retrieved from Git without checking out that particular commit, such as the commit message, author, timestamp and modified files. `in-source-code` also allows reasoning over Git information, but also does a checkout of the code and provides AST information. Currently, QwalKeko does not provide bindings for its queried projects. Eclipse can only generate bindings when a project is built

without errors, and automatically setting up a project is not an easy feat. Libraries are not always included in the repository, nor can we easily deduce which version of a library was used in a particular commit. QwalKeko used to use partial program analysis [2] but this was too slow for large-scale querying.

The following query finds the compilation units (the root note of an AST) of every modified file in all the versions of the queried software project:

```
1 (qwalkeko* [?info ?cu ?end]
2   (qwal graph root ?end []
3     (q=>*)
4     (in-source-code [curr]
5       (fileinfo|edit ?info curr)
6       (fileinfo|compilationunit ?info ?cu curr))))
```

The first line uses `qwalkeko*`, which has a similar function as `ekeko*` in that it configures the logic engine and specifies which variables will be the result of the query. The second line configures the qwal engine. Both `graph` and `root` are already bound, while `?end` will be bound to the end version. On the third line we skip an arbitrary number of versions. This pattern is functionaly equivalent to mapping the rest of the query over all the versions in the graph. In the last three lines we specify that we are interested in a modified file, as denoted by `fileinfo|edit/2`, and its corresponding compilation unit. A `fileinfo` object contains the path of a file and how it was changed (either added, removed or modified).

3.1 ChangeNodes

QwalKeko provides an implementation of a tree distilling algorithm named ChangeNodes[6]. It takes as input two AST nodes and outputs a minimal edit script that, when applied, transforms the first AST into the second one. The edit script will contain the following operations:

Insert A node is inserted in the AST
Delete A node is removed from the AST
Move A node is moved to a different location in the AST
Update A node is updated/replaced with a different node

The algorithm is based upon the work of Chawathe et. al [1] and has been used in ChangeDistiller [4]. The main difference between ChangeNodes and ChangeDistiller is that ChangeNodes works directly on top of the JDT nodes. ChangeNodes uses a language-aware representation, while ChangeDistiller uses a language-agnostic representation. ChangeNodes differs in its matching strategy. During the matching phase nodes from the original AST are matched with nodes in the target AST. Two nodes match when they are considered to be the same (even though their value may differ). When two AST nodes match ChangeNodes will also match all mandatory properties of these nodes We have not yet compared results from both implementations.

[6] https://github.com/ReinoutStevens/ChangeNodes

QwalKeko introduces a new predicate `(change ?change source target)`, which binds `?change` to a change operation between the source AST and target AST. Another predicate `changes/3` is similar, but binds the first argument to the whole list of operations between both ASTs.

The following code demonstrates how one can use this predicate:

```
1 (qwal graph root ?end [?left-cu ?right-cu ?change]
2   (in-source-code [curr]
3     (ast :CompilationUnit ?left-cu))
4   q=>
5   (in-source-code
6     (compilationunit|corresponding ?left-cu ?right-cu)
7     (change ?change ?left-cu ?right-cu)))
```

On line 3 it binds `?left-cu` to a compilation unit in the root version of the graph. It moves to one of the successors of that version on line 4. On lines 5–6 we retrieve the corresponding compilation unit using `compilationunit|corresponding/2`, which looks for a compilation unit in the same package that defines the same type. Finally we compute the changes between these two compilation units using `change/3`. We have successfully used QwalKeko to classify changes made to Selenium files. This work is currently under revision at ICSM.

4 Conclusion

We have provided a short overview of the capabilities of QwalKeko. QwalKeko allows querying over the history of software projects stored in Git. To this end, it combines the programming query language Ekeko with regular path expressions, extended with several history predicates. The most important one is `change/3`, which reifies change operations made to ASTs. It uses ChangeNodes, which implements a tree differencing algorithmn on top of JDT AST nodes. We have successfully used QwalKeko to identify changes made to Selenium files.

References

1. Chawathe, S.S., Rajaraman, A., Garcia-Molina, H., Widom, J.: Change detection in hierarchically structured information. In: Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD96). pp. 493–504 (1996)
2. Dagenais, B., Hendren, L.: Enabling static analysis for partial java programs. In: In Proceedings of the 23rd ACM SIGPLAN conference on Object oriented programming systems languages and applications (OOPSLA08) (2008)
3. De Roover, C., Stevens, R.: Building development tools interactively using the ekeko meta-programming library. In: Proceedings of the CSMR-WCRE Software Evolution Week (CSMR-WCRE14) (2014)
4. Fluri, B., Würsch, M., Pinzger, M., Gall, H.C.: Change distilling: Tree differencing for fine-grained source code change extraction. Transactions on Software Engineering 33(11) (2007)
5. Stevens, R., De Roover, C., Noguera, C., Jonckers, V.: A history querying tool and its application to detect multi-version refactorings. In: Proceedings of the 17th European Conference on Software Maintenance and Reengineering (CSMR13) (2013)

Evolution and dependencies of Hackage packages

Extended Abstract (Work in Progress) SATToSE 2014

Philipp Schuster and Ralf Lämmel

Software Languages Team
University of Koblenz-Landau, Germany
http://softlang.wikidot.com

1 Introduction

It is common practice to use a tool called Cabal to fetch and build libraries from a central repository for Haskell libraries called Hackage. As these libraries evolve, new versions of them are released as packages. Every package comes with metadata about its dependencies. Since not every package is compatible with every version of its dependencies, the metadata states a version range which includes packages known to work, but also future packages expected to work. The metadata also excludes packages known not to work and those expected not to work.

When package authors introduce a breaking change, then they release a package with an increased major version number that falls out of the range of all packages depending on it. It is only when authors of depending packages have ensured compatibility with the new version of the dependency that they in turn release a new version of their package that includes the new version of the dependency in the version range. This might be a breaking change in itself and if the number of reverse dependencies is high cascade into a lot of work for package maintainers.

It has been noted that not every change to a library that might be breaking necessarily does so. A different policy has been proposed to exclude upper bounds on the version ranges and only retroactively add them, when breakage actually occurs. This is a controversial topic that has spawned many discussions. The reason a breaking change might not affect a dependent package is that the broken functionality is not even used. The idea now is to do more fine-grained dependency analysis to find out how common this is. In order to do this we need not only information about dependencies between packages but between individual declarations.

The objective of this work is to support discussions around different versioning policies with real world data gathered from Hackage. It also explores possible ways to automate the detection of compatibility. More specifically, we formally define, based on our data model, the following queries:

1. Are two declarations used interchangeably?
2. Is an update of the dependencies of a certain package safe?
3. Are two declarations in conflict and cannot be used together?

We then run these queries on the gathered data to find all concrete instances of these situations.

Language-specific software repositories and even Hackage have been analysed before; see, for example, [3, 5, 2]. However the questions these tried to answer were different. There has also been work on conflicting software packages [1], but there it is assumed that conflicts are accurately reflected by the version constraints. Impact analysis on software repositories has been done before [4], but not across packages.

2 Methodology

The dependencies between declarations are established by one declaration mentioning a symbol the other declaration declares. For example, a symbol could be the name of a function or a data type. We also need the module the symbol comes from and its genre (e.g., constructor, type, method, etc.) for disambiguation.

We extract the following facts:

1. For each package its set of dependencies.
2. For each package all declarations with their abstract syntax tree.
3. For each declaration all declared symbols.
4. For each declaration all mentioned symbols.

```
dependency      ⊆ Package     × Package
declaration     ⊆ Package     × Declaration
mentionedsymbol ⊆ Declaration × Symbol
declaredsymbol  ⊆ Declaration × Symbol

ast             ⊆ Declaration × AST
modulename      ⊆ Symbol      × ModuleName
symbolname      ⊆ Symbol      × SymbolName
genre           ⊆ Symbol      × Genre
```

Fig. 1. Extracted Relations

The data is inserted into a database. An example of the kind of query we run against the database is given in Figure 2. This query is also used as part of more complex queries.

We want to know whether a declaration D uses a declaration E by mentioning a symbol S. This means there are two packages P and Q such that Q has to be a dependency of P. Then the declaration D from package P mentions the symbol S while the declaration E from package Q declares the same symbol S.

For reasons of scalability, at this stage of this work, we analyze a sample of Hackage. We chose the forty libraries with the greatest numbers of reverse dependencies because of their perceived relevance. We try to analyze all versions

```
uses(D,S,E) :-
    dependency(P,Q),
    declaration(P,D),
    mentionedsymbol(D,S),
    declaration(Q,E),
    declaredsymbol(E,S).
```

Fig. 2. Example Query

of these libraries with our custom processor which parses the source code and performs name resolution to find the mentioned symbols. Unfortunately not all of the packages and their dependencies can be parsed successfully. Table 3 has a summary of the numbers of packages, declarations, distinct abstract syntax trees, and symbols in our database, at the time of writing.

Packages	1149
Declarations	217007
Distinct abstract syntax trees	19167
Symbols	12407

Fig. 3. Numbers of entities in our database

3 Illustration

In our definition of a safe update two declarations are different, if the pretty printings of their abstract syntax trees are different. That is, we consider every change except reformatting to be a breaking change. This introduces false negatives because there are different abstract syntax trees that are semantically equivalent. One preliminary result is the detection of such abstract syntax trees.

Let's assume all version bounds only include valid versions. Then, if two declarations with the same abstract syntax tree might use two declarations with different abstract syntax trees we call the two different abstract syntax trees interchangeable. Figure 4 formally defines this.

If we run this query on the gathered data, we indeed find interchangeable but different abstract syntax trees. Inspection reveals that the differences can be classified into three kinds:

1. Changes in style with no semantic significance.
2. Backwards-compatible changes of interfaces.
3. Semantic changes that are probably benevolent (bugfixes).

Figure 5 shows two function definitions that are used interchangeably. They obviously have the same semantics; it is just that the argument was renamed.

```
itnerchangeable(E1,E2) :-
    uses(D1,S,E1),
    uses(D2,S,E2),
    ast(D1,AD1),ast(D2,AD2),
    AD1 = AD2,
    ast(E1,AE1),ast(E2,AE2),
    AE1 ≠ AE2.
```

Fig. 4. Query for interchangeable abstract syntax trees

```
unpack ps = build (unpackFoldr ps)

unpack bs = build (unpackFoldr bs)
```

Fig. 5. Two interchangeable function definitions

4 Concluding remarks

We want to find facts about the evolution and dependencies of Haskell packages. To do this, we have gathered data from Hackage, the largest repository for Haskell packages. We have then defined and executed queries against the data.

We have not yet implemented all the queries we would like to run. In particular the query to find safe updates still needs work. The number of packages we analyse will also be increased. The limiting factors here are build problems with our custom compiler and the duration of one extraction run.

References

1. Artho, C., Cosmo, R.D., Suzaki, K., Zacchiroli, S.: Sources of inter-package conflicts in debian. CoRR abs/1110.1354 (2011)
2. Bajracharya, S., Ossher, J., Lopes, C.: Sourcerer: An infrastructure for large-scale collection and analysis of open-source code. Sci. Comput. Program. 79, 241–259 (Jan 2014), http://dx.doi.org/10.1016/j.scico.2012.04.008
3. Bezirgiannis, N., Jeuring, J., Leather, S.: Usage of generic programming on hackage: Experience report. In: Proceedings of the 9th ACM SIGPLAN Workshop on Generic Programming. pp. 47–52. WGP '13, ACM, New York, NY, USA (2013), http://doi.acm.org/10.1145/2502488.2502494
4. Canfora, G., Cerulo, L.: Fine grained indexing of software repositories to support impact analysis. In: Proceedings of the 2006 International Workshop on Mining Software Repositories. pp. 105–111. MSR '06, ACM, New York, NY, USA (2006), http://doi.acm.org/10.1145/1137983.1138009
5. Raemaekers, S., Deursen, A.v., Visser, J.: The maven repository dataset of metrics, changes, and dependencies. In: Proceedings of the 10th Working Conference on Mining Software Repositories. pp. 221–224. MSR '13, IEEE Press, Piscataway, NJ, USA (2013), http://dl.acm.org/citation.cfm?id=2487085.2487129

Inheritance Usage in Java: A Replication Study

C. Aytekin, T. van der Storm
cigdem.aytekin2@student.uva.nl, storm@cwi.nl

Abstract

Inheritance is an important mechanism in object oriented languages. Quite some research effort is invested in inheritance until now. Most of the research work about inheritance (if not all) is done about the inheritance relationship between the classes. There is also some debate about if inheritance is good or bad, or how much inheritance is useful. Tempero et al. raised another important question about inheritance [TYN13]. Given the inheritance relationships defined, they wanted to know how much of these relationships were actually used in the system. To answer this question, they developed a model for inheritance usage in Java and analysed the byte code of 93 Open Source Java projects from Qualitas Corpus [TAD10]. The conclusion of the study was that inheritance was actually used a lot in these projects - in about two thirds of the cases for subtyping and about 22 percent of the cases for (what they call) external reuse. Moreover, they found out that downcall (late-bound self-reference) was also used quite frequently, about a third of inheritance relationships included downcall usage. They also report that there are other usages of inheritance, but these are not significant.

In this study, we replicate the study of Tempero et al. using Rascal meta-programming language. We use the inheritance model of the original study and also the open source projects from Qualitas Corpus, but we analyse the Java source code instead of the byte code. Our study is still in progress and therefore we do not report any results at the moment.

1 Research Questions of the Original Study

RQ1:	To what extent is late-bound self-reference relied on in the designs of Java Systems? (The terms late-bound self-reference and downcall are synonyms in the study.)
RQ2:	To what extent is inheritance used in Java in order to express a subtype relationship that is necessary to design?
RQ3:	To what extent can inheritance be replaced by composition?
RQ4:	What other inheritance idioms are in common use in Java systems?

2 Limitations of the Original Study

The limitations of the original study are as follows:

- The study is limited to Java classes and interfaces, exceptions, enums and annotations are excluded,
- The third party libraries are not analysed,
- The edges between system types and non-system types are not modelled,
- Heuristics are used when defining framework and generics attributes,
- The authors use the Java byte code as input to their analysis tool, byte code may in some cases incorrectly map to source code,

- They do make static code analysis and this may have impact on their down call results, the results may be overstating the reality

3 Results

For Research Question 1: They conclude that late-bound self-reference plays a significant role in the systems they studied - around a third (median 34 %) of CC edges involve down calls.

For Research Question 2: At least two thirds of all inheritance edges are used as subtypes in the program, the inheritance for subtyping is not rare.

For Research Question 3: The authors found that 22 % or more edges use external reuse (without subtyping) and 2 % or more use internal re-use (without subtyping or external reuse). They conclude that this result introduces opportunities to replace inheritance with composition.

For Research Question 4: They also report some other uses of Java inheritance (constant, generic, marker, framework, category and super), however the results show that big majority of edges (87 %) in their Corpus can already be explained with one of the subtype, external re-use, internal re-use uses and other usages do not occur frequently.

4 Definitions

4.1 System Type

A system type is created for the system under investigation. A non-system type or an external type, on the other hand, is used in the system, but is not defined in the system.

4.2 User Defined Attribute

The descendant ascendant pair in an inheritance relationship has user defined attribute if both of descendant and ascendant are system types.

4.3 CC, CI and II Attributes

The descendant-ascendant pair in an inheritance relationship in Java can have one of the three attributes: CC (Class Class) - both descendant and ascendant are classes, CI (Class Interface) - descendant is a class and ascendant is an interface or II (Interface Interface) - both descendant and ascendant are interfaces.

4.4 Explicit Attribute

The inheritance relationship is described directly in the code.

4.5 Internal Reuse

Internal reuse happens when a descendant type calls a method or accesses a field of its ascendant type.

4.6 External Reuse

External reuse is like internal reuse, except for that the access to a method or a field happens not within the descendant type itself, but it happens in another type, on an object of descendant type. According to the original study, the class in which the external reuse occurs may not have any inheritance relationship with the descendant or ascendant type.

4.7 Subtype

Subtype usage happens when an object of descendant type is supplied where an object of ascendant type is expected. Subtype usage can occur in four occasions: when assigning object(s), during parameter passing, when returning an object in a method or casting an object to another type. Contrary to internal and external reuse, the place where the subtyping occurs is not of any importance here.

There are two interesting cases of subtyping usage in Java (sideways cast and this changing type). Please see the original article [TYN13] for the definitions for these two specific cases.

4.8 Downcall

The terms downcall and late-bound self-reference have the same meaning in the original study. Downcall refers to the case when a method in the ascendant type (ascendant-method) makes a call to another method (descendant-method) which is overridden by the descendant type. When an object of descendant type calls the ascendant-method, the descendant-method of the descendant type will be executed.

4.9 Other Uses of Inheritance

Next to reuse, subtype and downcall, the authors also defined other uses of inheritance: Category, Constants, Framework, Generic, Marker and Super.

Category Category inheritance relationship is defined for the descendant ascendant pairs which can not be placed under any other inheritance definition. (We should also note that for this definition, ascendant type should be direct ascendant of the descendant type, i.e. no types are defined between the two types in the inheritance hierarchy.) In this case, we search for a sibling of the descendant which has a subtype relationship with the ascendant. If we can find such a sibling, we assume that the ascendant is used as a category class, and the descendant is placed under it for conceptual reasons.

Constants A descendant ascendant pair has constants attribute if the ascendant only contains constant fields (i.e., fields with `static final` attribute). The ascendant should either have no ascendant it self or if it has ascendants, the pair ascendant-(grand)ascendant should also have constants attribute.

Framework A descendant-ascendant pair will have the framework attribute if it does not have one of the external reuse, internal reuse, subtype or downcall attributes and the ascendant is a direct descendant of a third party type. Moreover, the first type should be direct descendant of the second type.

Generic Generic attribute is used for the descendant ascendant (for example : descendant type R, and ascendant type S) pairs which adhere to the following:

1. S is parent of R. (i.e. S is direct ascendant of R.)
2. R has at least one more parent, say, T.
3. There is an explicit cast from java.lang.Object to S.
4. There is a subtype relationship between R and java.lang.Object

Marker Marker usage for a descendant-ascendant pair occurs when an ascendant has nothing declared in it. Moreover, just like the constants definition, the ascendant should either have no ascendants itself, or if it has ascendants, ascendant-(grand)ascendant pairs should all have marker attribute. Ascendant should be defined as an interface and descendant may be a class or an interface.

Super A descendant-ascendant pair will qualify for super attribute if a constructor of descendant type explicitly invokes a constructor of ascendant type via `super` call.

5 Metrics

The metrics are explained elaborately in the website of the original study [TYN08]

6 Replication Study

6.1 Research Questions of the Replication Study

Our research questions directly refer to the four research questions of the original study 1 and can be summarized as: How do our results differ from those of the original study?

6.2 Differences in the Study Set-up

Source code versus byte code: The biggest difference between the original and replication studies is about the input to analysis work.

Differences between the content of the byte code and the source code: It may be possible that different set of classes or interfaces are included in binary and source forms. One may rightfully expect that many classes and interfaces would be included in both of the forms, however, there may be differences between the two.

Qualitas Corpus vs. Qualitas.class Corpus: The authors used the Qualitas Corpus [TAD10], we also use the Corpus, but the compiled version of it Qualitas.class Corpus [TMVB13b].

7 Code Analysis with Rascal

Code analysis in Rascal is straightforward and easy to understand. For example, getting the list of methods which are called more than n times in a project is as easy as the following in Rascal:

```
set[Declaration] pASTs =
        createAstsFromEclipseProject(|project://cobertura-1.9.4.1|, true);
map [loc, num]  mMap = ();
for (anAST <- pASTs) {
    visit (anAST) {
        case m1:\methodCall(_, _, _, _) : {
            if (m1@decl in  mMap)
                 {  mMap[m1@decl] =  mMap[m1@decl] + 1; }
            else {  mMap += (m1@decl : 1) ; }
        }
        case m2:\methodCall(_,_,_) : {
            if (m2@decl in  mMap)
                 {  mMap[m2@decl] =  mMap[m2@decl] + 1; }
            else { mMap += (m2@decl : 1); };
        }
    }
}
map [loc, num] fMethods =
       (aMethod :  mMap[aMethod] | aMethod <-  mMap,  mMap[aMethod] > 400 );
println("Frequently called methods:"); iprintln(fMethods);
```

References

[TAD10] Ewan Tempero, Craig Anslow, Jens Dietrich, Ted Han, Jing Li, Markus Lumpe, Hayden Melton, and James Noble. Qualitas corpus: A curated collection of java code for empirical studies. In 2010 Asia Pacific Software Engineering Conference (APSEC2010), pages 336–345, December 2010.

[TMVB13b] Ricardo Terra, Luis Fernando Miranda, Marco Tulio Valente, and Roberto S. Bigonha. Qualitas.class Corpus: A compiled version of the Qualitas Corpus. *Software Engineering Notes*, 38(5):1–4, 2013.

[TYN08] Ewan D. Tempero, Hong Yul Yang, and James Noble. Inheritance Use Data. Inheritance Use Data, 2008. URL: https://www.cs.auckland.ac.nz/ ewan/qualitas/studies/inheritance/.

[TYN13] Ewan D. Tempero, Hong Yul Yang, and James Noble. What programmers do with inheritance in java. In Giuseppe Castagna, editor, *ECOOP*, volume 7920 of Lecture Notes in Computer Science, pages 577–601. Springer, 2013.

Mining metrics for understanding metamodel characteristics

Juri Di Rocco

DISIM University of L'Aquila,
juri.dirocco@univaq.it

Abstract. Metamodels are a key concept in Model-Driven Engineering. Any artifact in a modeling ecosystem has to be defined in accordance to a metamodel prescribing its main qualities. Hence, understanding common characteristics of metamodels, how they evolve over time, and what is the impact of metamodel changes throughout the modeling ecosystem is of great relevance. Similarly to software, metrics can be used to obtain objective, transparent, and reproducible measurements on metamodels too. In this work, we present an approach to understand structural characteristics of metamodels. A number of metrics are used to quantify and measure metamodels and cross-link different aspects in order to provide additional information about how metamodel characteristics are related. The approach is applied on repositories consisting of more than 450 metamodels.

Keywords: Model Driven Engineering, metamodeling, metamodel metrics

1 Introduction

Metamodels are a key concept in Model-Driven Engineering [12]. Almost any artifact in a modeling ecosystem [6] has to be defined in accordance to a metamodel, which represents an ontological descriptions of application domains [5]. Metamodels are important because they formally define the modeling primitives used in modeling activities and represent the *trait-d'union* among all constituent components.

Despite the relevance of metamodels, little research has been undertaken on their empirical analysis. Understanding common characteristics of metamodels, how they evolve over time, and what is the impact of metamodel changes throughout the modeling ecosystem is key to success. Several approaches have been already proposed to analyse models [11] and transformations [1,13] with the aim of assessing quality attributes, such as understandability, reusability, and extendibility [4]. Similarly, there is the need for techniques to analyse metamodels as well in order to evalutate their structural characteristics and the impact they might have during the whole metamodel life-cycle especially in case of metamodel evolutions. To this end, some works [8,10] propose the adoption of metrics for analysing metamodels as typically done in software development by means of object-oriented measurements [7].

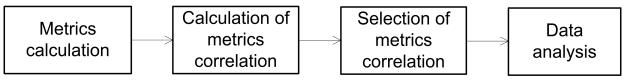


Fig. 1. Overview of the process for metamodel analysis

2 Measuring metamodels

In the follow, we describe the process we have applied to identify linked structural characteristics and to understand how they might change depending on the nature of metamodels. The first step of the proposed process consists of the application of metrics on a data set of metamodels. Concerning the applied metrics we borrowed those in [8] and added new ones by leading to a set of 28 metrics. The corpus of the analyzed metamodels has been obtained by retrieving metamodels from different repositories, i.e., EMFText Zoo [3], ATLZoo [2], Github, GoogleCode. Such metamodels have been downloaded from [9] for a total number of 466 metamodels belonging to different technical spaces and domains. Afterwards all the calculated metrics are correlated by using statistical tools. Finally, the collected data are analysed in order to cross-link struc- tural characteristics of metamodels (e.g., how the adoption of hier- archies changes with the number of metaclasses, and how the size of structural features typically changes depending on the introduc- tion of abstract metaclasses).

Correlation is probably the most widely used statistical method to detect cross-links and assess relationships among observed data. There are different techniques and indexes to discover and measure correlations. We have considered the Pearsons and Spearmans coefficients to measure the correlations among calculated metamamodel metrics.

Our purposes are analises how the most correlated metamodel metrics have been identified and selected. For each couple we have selected the coefficient index (between Pearson and Spearman) having the higher correlation value. In this section we discuss some relevant correlations we have identified as described in the previous section. This permits to draw interesting conclusions about how some structural metamodel char acteristics are coupled.

In this work, we proposed a number of metrics which can be used to acquire objective, transparent, and reproducible measurements of metamodels. The major goal is to better understand the main characteristic of metamodels, how they are coupled, and how they change depending on the metamodel structure. A correlation analysis has been performed to identify the most cross-linked metrics, which have, in turn, been computed over 450 metamodels summarized in Table 1

- the adoption of inheritance is proportional to the size of metamodels;
- the number of metaclasses with supertypes are inversely proportional to the average number of structural features;
- the number of metaclasses with supertypes is proportional to the number of metaclasses without attributes or references; finally,

Correlated Metrics	
↑ usage of inheritance	↑ metamodel size
↑ #metaclasses w/ supertypes	↓ avg. #features
↑ #metaclasses w/ supertypes	↑ #metaclasses w/o features

Table 1. Summary of correlated metrics

– isolated metaclasses are not commonly used apart from testing or educational purposes.

Threats to validity are present and cannot be neglected. In particular, the metamodels are mainly from academic repositories, since it is not easy to collect metamodels from industry. Even though complex metamodels like UML, MARTE, and BPMN are part of the analyzed corpus, we cannot claim that the results of our analysis apply to industrial metamodels in general. The difficulty is principally due to the proprietary nature of these artifacts representing a strategic and valuable asset for the holder. Nevertheless, we intend to overcome at some extent this problem and apply the approach also on this kind of metamodels: comparing the results with those obtained from the academic metamodels might reveal interesting differences and make the results more generally valid.

3 Future Work

One of the main goal in the next future is to extend the approach to analyse the characteristics of coupled modeling artifacts, e.g., how structural characteristics of metamodels affect those of model transformations or any metamodel-based artifact. An transformation will take a set of model as input and will produce a set of model as output. Both input and output models are conform to its metamodel. Therefore transformation domani and co-domain are metamodel, so there are reference from metamodel and trasformation. Likewise metamodel process analisys, we need to select corpus and metrics for transformation. Our analytical work wants to focus on the correlations that may exist between the calculated metrics on metamodels and those calculated on the transformations. The analysis process will be very similar to the one shown in this work. The long term goal of this work is to define an approach able to estimate or even predict the cost of developing or refining modeling artifacts by considering the structural characteristics of the corresponding metamodels.

References

1. van Amstel, M., van den Brand, M.: Quality assessment of atl model transformations using metrics. Proceedings of the 2nd International Workshop on Model Transformation with ATL (MtATL 2010), Malaga, Spain (June 2010) (2010)
2. ATLAS Group: ATL Transformations Zoo. http://www.eclipse.org/m2m/atl/atlTransformations/

3. ATLAS Group: EMFTEXT Concrete Syntaxes Zoo. http://emftext.org/index.php/EMFText_Concrete_Syntax_Zoo$
4. Bansiya, J., Davis, C.G.: A hierarchical model for object-oriented design quality assessment 28, 4 (Jan 2002)
5. Chandrasekaran, B., Josephson, J., Benjamins, V.: What are ontologies, and why do we need them? Intelligent Systems and their Applications, IEEE 14(1), 20–26 (1999)
6. Di Ruscio, D., Iovino, L., Pierantonio, A.: Evolutionary togetherness: how to manage coupled evolution in metamodeling ecosystems. In: Intl. Conf. on Graph Transformations (ICGT 2012). LNCS, vol. 7562. Springer (2012)
7. Harrison, R., Counsell, S., Nithi, R.: An evaluation of the mood set of object-oriented software metrics. IEEE Transactions on Software Engineering 24, 491–496 (1998)
8. James, W., Athansios, Z., Nicholas, M., Louis, R., Dimitios, K., Richard, P., Fiona, P.: What do metamodels really look like? Frontiers of Computer Science (2013)
9. James Williams: James r. williams corpus. http://www.jamesrobertwilliams.co.uk/mm-analysis/resources/metamodel-corpus-20130722.zip
10. Ma, Z., He, X., Liu, C.: Assessing the quality of metamodels. Frontiers of Computer Science 7(4), 558–570 (2013), http://dx.doi.org/10.1007/s11704-013-1151-5
11. Monperrus, M., Jézéquel, J.M., Champeau, J., Hoeltzener, B.: Model-Driven Software Development. IGI Global (Aug 2008), http://www.igi-global.com/chapter/measuring-models/26829/
12. Schmidt, D.: Guest Editor's Introduction: Model-Driven Engineering. Computer 39(2), 25–31 (2006)
13. Vignaga, A.: Metrics for measuring atl model transformations. Tech. rep. (2009)

A Taxonomy for Bidirectional Model Transformation and Its Application

Romeo Marinelli

DISIM - Università degli Studi dell'Aquila, Via Vetoio, Coppito I-67010, L'Aquila, Italy
romeo.marinelli@univaq.it

Abstract. In Model Driven Engineering bidirectional transformations are considered a core ingredient for managing both the consistency and synchronization of two or more related models. However, current languages still lack of a common understanding of its semantic implications hampering their applicability in practice. In this paper, relevant properties pertaining bidirectional model transformations are illustrated. It is based on the discussions of a working group on bidirectional model transformation of the Dagstuhl seminar on Language Engineering for Model-Driven Software Development (2005, 2008 and 2011) and characteristics of main existing tools for BX. This taxonomy can be used, among others, to help developers in deciding which model transformation language or tool is best suited to carry out a particular model transformation activity.

1 Introduction

Model Driven Engineering (MDE) is a software engineering discipline in which models are the primary artifacts and play a central role throughout the entire development process. Model transformations are the core MDE mechanism for building software from design to code, and hence have a significant impact on the software development process. They are used for different reasons and intents, e.g., to extract different views from a model (query), add or remove detail (refinement or abstraction), generate code from model (synthesis). One of the best ways to combat complexity of software development is through the use of abstraction, problem decomposition, and separation of concerns. The practice of software modeling has become a major way of implementing these principles. Model-driven approaches to systems development move the focus from third-generation programming language (3GL) code to models (in particular models expressed in UML and its profiles).

Working with multiple, interrelated models that describe a software system require significant effort to ensure their overall consistency. It follows that automating the task of model consistency checking and synchronization would greatly improve the productivity of developers and the quality of the models [15]. In addition to vertical and horizontal model synchronization, the burden of many activities of software development, could be significantly reduced through automation.

Many of these activities can be performed as automated processes, which take one or more source models as input and produce one or more target models as output, following a set of transformation rules [7]. We refer to this process as model transformation. For the model-driven software development vision to become reality, tools must

be able to support the automation of model transformations. Development tools should not only offer the possibility of applying predefined model transformations on demand, but should also offer a language that allows (advanced) users to define their own model transformations and then execute them on demand. Beyond transformation execution automation, it would also be desirable that tools could make suggestions as to which model transformations could be appropriately applied in a given context, but this aspect is very ambitious and out of the scope of this article.

Bidirectionality is a relevant aspects in model transformations: often it is assumed that during development only the source model of a transformation undergoes modifications, however in practice it is necessary for developers to modify both the source and the target models of a transformation and propagate changes in both directions [13, 15]. A number of approaches and languages have been proposed due to the intrinsic complexity of bidirectionality. Each one of those languages is characterized by a set of specific properties pertaining to a particular applicative domain [14]. Unfortunately, the existing approaches are affected by the intrinsic characteristics of model transformations, that have been only partially considered by research. In order to decide which model transformation approach is most appropriate for addressing a particular problem, a number of crucial questions need to be answered. The following subsections investigate a particular question, and suggest a number of objective criteria to be taken into consideration to provide a concrete answer to the question. Based on his requests, the developer can then select the model transformation approach that is most suited for his needs. In this paper we propose a taxonomy of bidirectional model transformation (BX). Such a taxonomy is particularly useful to help a software developer choosing a particular model transformation approach that is best suited for his needs. The taxonomy is based on the discussions of a working group of the Dagstuhl seminar on Language Engineering for Model-Driven Software Development 2005 and 2011 [2, 8]. In this paper we want to adapt and improve that taxonomy to bidirectional model transformations. Finally, we will apply these characteristics to the main tool for existing BX: TGG, QVT, Lenses, JTL and GRound-Tram.

2 Model transformation taxonomy

In order to decide which bidirectional model transformation approach is most appropriate for addressing a particular problem, a number of crucial questions need to be answered. This section investigate and suggest a number of objective criteria to be taken into consideration to provide a concrete answer to the question. Based on the answers, the developer can then select the model transformation approach that is most suited for his needs. This section provides the evaluation criteria for model transformations extending the work of Mens and Gorp [10, 11, 9]. The goal of model transformations is to automatically generate different views of a system from a source model and to support the code generation. Both the target model and the transformation rules can be evaluated according to some basic principles derived from software engineering. In general, a bidirectional transformation is required to keep all views synchronized with the underlying system by supporting and automatically propagating changes in both directions.

The BX taxonomy has been developed by means a comparision grid. It has been build by using both existing papers in literature [8, 3–5, 1, 12] and analyzing the main existing tools [6, 1].

The grid has been built considering parameters divided in three main category: general requirements (GR), functional requirements (FR) and non functional requirements (NFR).

-*general requirements GR*: level of automation, complexity of the transformation, visualization, level of industry application, maturity level;

-*functional requirements FR*: correctness of the transformations, inconsistency management, modularity (compositional/non compositional), traceability, change propagation, incrementality, uniqueness of transformation, termination of transformation, symmetric/asymmetric behavior of transformation, type of artifact, data model, endogenous/exogenous, transformation mechanisms, in-place/out-of-place transformations;

-*non functional requirements NFR*: extensibility/modifiability, usability and utility, scalability, robustness verbosity and conciseness, interoperability, reference platform (standardizzation), verificability and validity of a transformation.

3 Tools used

List of useful tools: eMoflon (TGG), EMORF (TGG), QVT-R (Medini), Boomerang (Lenses), JTL, GRound-Tram and BiFlux.

4 Grid application

Each tool will be analyzed by using the grid mentioned above.

5 Conclusions and future developments

As future developments, it can extend the comparision grid and to consider other tools to BX.

References

1. A. Cicchetti, D. Di Ruscio, R. Eramo, and A. Pierantonio. Jtl: a bidirectional and change propagating transformation language. In *3rd International Conference on Software Language Engineering (SLE)*, Oct. 2010. to appear.
2. K. Czarnecki, J. N. Foster, Z. Hu, R. Lämmel, A. Schürr, and J. F. Terwilliger. Bidirectional Transformations: A Cross-Discipline Perspective—GRACE meeting notes, state of the art, and outlook. In *ICMT2009 - International Conference on Model Transformation, Proceedings*, volume 5563 of *LNCS*, pages 260–283. Springer, 2009.
3. J. Foster, M. Greenwald, J. Moore, B. Pierce, and A. Schmitt. Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. *ACM Trans. Program. Lang. Syst.*, 29(3), 2007.

4. J. N. Foster. *Bidirectional Transformation Languages*. PhD thesis, University of Pennsylvania, 2010.
5. S. Hidaka, Z. Hu, K. Inaba, H. Kato, and K. Nakano. Groundtram: An integrated framework for developing well-behaved bidirectional model transformations. In P. Alexander, C. S. Pasareanu, and J. G. Hosking, editors, *ASE*, pages 480–483. IEEE, 2011.
6. S. Hidaka, Z. Hu, H. Kato, and K. Nakano. Towards a compositional approach to model transformation for software development. In S. Y. Shin and S. Ossowski, editors, *SAC*, pages 468–475. ACM, 2009.
7. S. Hildebrandt, L. Lambers, H. Giese, J. Rieke, J. Greenyer, W. Schäfer, M. Lauder, A. Anjorin, and A. Schürr. A survey of triple graph grammar tools. In *Second International Workshop on Bidirectional Transformations (BX 2013)*, volume 57, pages 1–18. EC-EASST, 2013.
8. Z. Hu, A. Schürr, P. Stevens, and J. F. Terwilliger. Bidirectional transformation "bx" (dagstuhl seminar 11031). *Dagstuhl Reports*, 1(1):42–67, 2011.
9. T. Mens. Model transformation: A survey of the state-of-the-art. In S. Gerard, J.-P. Babau, and J. Champeau, editors, *Model Driven Engineering for Distributed Real-Time Embedded Systems*. Wiley - ISTE, 2010.
10. T. Mens and P. V. Gorp. A taxonomy of model transformation. *Electronic Notes in Theoretical Computer Science*, 152:125–142, March 2006.
11. T. Mens, P. V. Gorp, D. Varro, and G. Karsai. Applying a model transformation taxonomy to graph transformation technology. *Electronic Notes in Theoretical Computer Science*, 152:143–159, March 2006.
12. A. Schürr. Specification of graph translators with triple graph grammars. In *in Proc. of the 20th Int. Workshop on Graph-Theoretic Concepts in Computer Science (WG '94), Herrsching (D*. Springer, 1995.
13. P. Stevens. A Landscape of Bidirectional Model Transformations. In R. Lämmel, J. Visser, and J. Saraiva, editors, *GTTSE 2007, Braga, Portugal, July 2-7*, volume 5235 of *Lecture Notes in Computer Science*, pages 408–424. Springer, 2008.
14. P. Stevens. Bidirectional model transformations in qvt: semantic issues and open questions. *Software and Systems Modeling*, 8, 2009.
15. Y. Xiong, D. Liu, Z. Hu, H. Zhao, M. Takeichi, and H. Mei. Towards automatic model synchronization from model transformations. In R. E. K. Stirewalt, A. Egyed, and B. Fischer, editors, *ASE*, pages 164–173. ACM, 2007.

Managing uncertainty in bidirectional transformations

Gianni Rosa

DISIM University of L'Aquila, Via Vetoio, 67100, Italy,
gianni.rosa@univaq.it

Abstract. In Model-Driven Engineering, the potential advantages of using bidirectional transformations are largely recognized. A key aspect is the non-deterministic nature of bidirectionality: consistently propagating changes from one side to the other is typically non univocal as more than one correct solution is admitted. In the paper, the problem of uncertainty in bidirectional transformations is discussed. In particular, we illustrate a uniform characterization of the representation of a family of cohesive models deriving from a transformation by means of models with uncertainty as already known in literature.

1 Introduction

In Model-Driven Engineering [11] (MDE), the potential advantages of using bidirectional transformations in various scenarios are largely recognized. As for instance, assuring the overall consistency of a set of interrelated models which requires the capability of propagating changes *back* and *forth* the transformation chain [12]. Despite its relevance, bidirectionality has rarely produced anticipated benefits as demonstrated by the lack of a leading language comparable, for instance, to ATL for unidirectional transformations due to the ambivalence concerning non-bijectivity. In fact, consistently propagating changes from one side to the other is typically non univocal as more than one correct solution is admitted and this gives place to a form of uncertainty.

Uncertainty is one of the main problem in software industry. Typically it occurs when the designer has not complete, consistent and accurate information required to make a decision during software development. In this paper we discuss about uncertainty indirectly generated as outcome of a model transformation process. In fact, when the transformation writer is not able to take some decisions, the specification can be left incomplete (e.g., constraints are missing) and more than one design alternatives may be obtained. Problems that originate uncertainty are common, from one hand, designer may be unsure about the mapping between some source and target elements, as a consequence the transformation is left incomplete and ambiguous mapping may cause the generation of multiple solution models each one representing a different design decision. On the other hand, during the transformation writing the designer may ignore that some information are missing and/or ambiguous. The designer can understand it only at execution time.

The paper is organized as follows. Section 2 introduces the concept of uncertainty and the main issues related to it by means of an example; In Section 3 we discuss how support uncertainty in bidirectional tranformation. Finally, Section 4 describes related work and draws some conclusions.

2 A motivating scenario

In order to better understand the above concepts, let us considering a typical scenario of round-trip engineering (RTE) [7]. The difficulty faced with RTE is the often neglected,

transformations are in general neither total nor injective. In other words, there are concepts in the source model that do not have a univocal correspondence in the target model and/or vice versa.

Take care the concept of uncertainty, in the following example we consider the *Collapse/Expand State Diagrams benchmark* defined in the *GRACE International Meeting on Bidirectional Transformations* [3]: starting from a hierarchical state diagram (involving some nesting) as the one reported in Figure 1(a), the bidirectional transformation yields a flat view provided in Figure 1(b). The main goal of the transformation is that any manual modifications on the (target) flat view should be back propagated and eventually reflected in the (source) hierarchical view. For instance, let us suppose the designer modifies the flat view by (see Δ change in Figure 1(c)): 1) adding the new state `Printing`, 2) adding the `print` transition that associates `Active` state to the latter, and 3) modifying the source of the `done` transition from the `Active` state to the `Printing` state. Then, in order to persist such a refinement to new executions of the transformation, the hierarchical state machine has to be consistently updated as illustrated in Figure 1(d).

We suppose that during the transformation writing, the designer has not finalized the exact behavior of the system, due to vague requirements given by the customer, such that an ambiguous mapping involving the new `print` transition is given. As a consequence, the back propagation of changes to hierarchical state machine gives place to an interesting situation: the new transition `print` can be equally mapped to each one of the nested states within `Active` as well as to the container state itself. In this way, more than one valid alternative may be proposed as output of the transformation (*non-determinism*), leaving the designers the choice for a suitable solution.

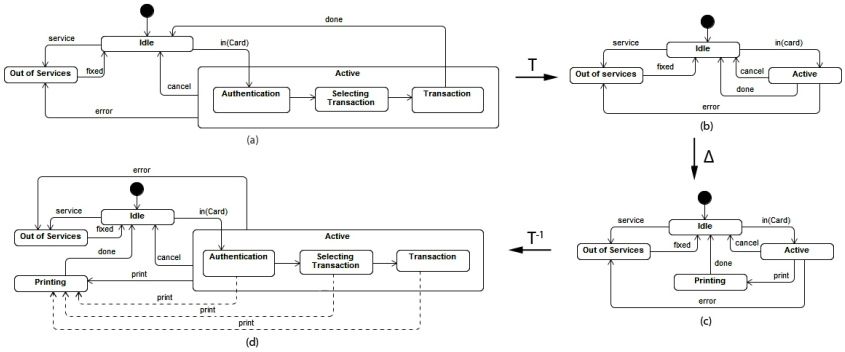


Fig. 1. Collapse/expand state diagrams in a round-trip process

3 Supporting uncertainty in bidirectional transformation

The need to express uncertainty about model content has been studied in different works [6, 9], where models seldom provide the means for expressing uncertainty. We are currently working on an approach to represent uncertainty as models, that can be taken as input by general purpose theories and tools in a MDE setting and that abstracts from the calculation method and allowing to harness the potential offered by generic modeling platforms (for instance [1]).

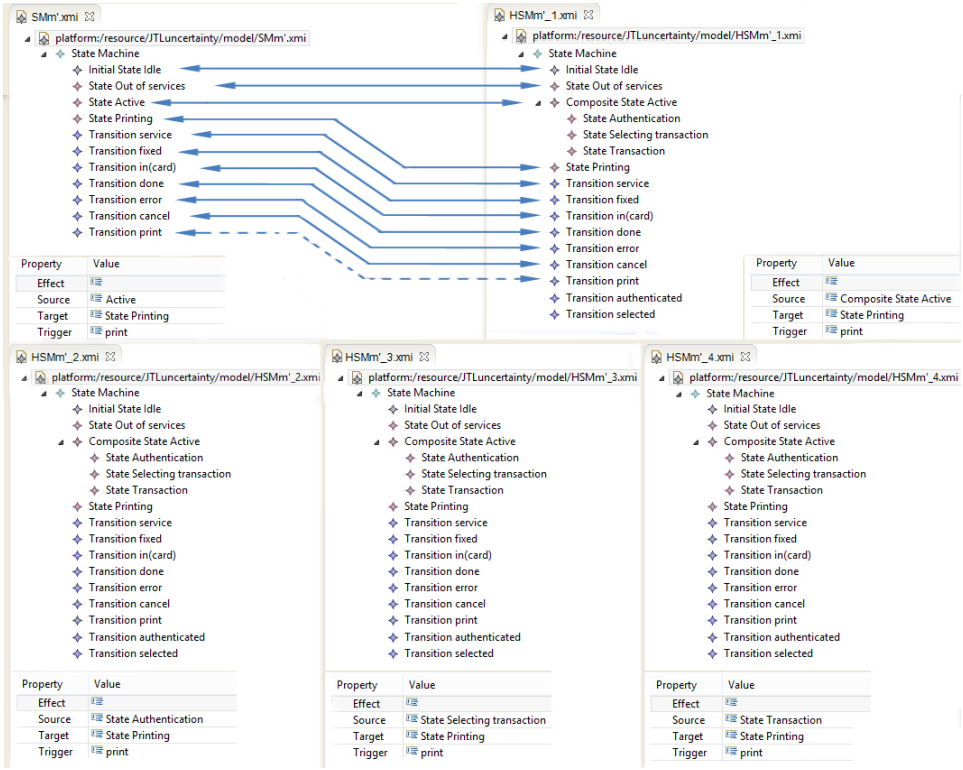


Fig. 2. The modified SM target model and the correspondent HSM source models

Considering a set of concrete model solutions generated by means of the JTL [2][1], conforms to a given metamodel *MM*, we aim to automatically derive a new metamodel *U-MM* able to specify uncertainty in models. In particular, *U-MM* has to provide the constructs able to express uncertain models consisting of a *base*-part model which contains the elements common to each concrete model and a set of *point of uncertainty* which collects the alternative *concrete* model elements. For instance, considering the set of alternatives depicted in the right side of Figure 2 may be expressed as a unique model containing uncertainty and conform to the *U-HSM* metamodel derived from the target *HSM* metamodel and able to represent all the concretizations resulting from JTL transformation engine.

4 Related work and conclusions

Uncertainty is one of the factors prevalent within contexts as requirements engineering [4], software processes [8] and adaptive systems [10]. Uncertainty management has been studied in many works, often with the aim to express and represent it in models.

[1] JTL is a constraint-based model transformation language specifically tailored to support bidirectionality

In [5] *partial model*s are introduced to allow designers to specify uncertain information by means of a base model enriched with annotations and first order logic. As discussed in this paper, modelers may need to encode ambiguities in their model transformation and obtain multiple design alternatives in order to choose among them. In contrast with this requirement, most existing bidirectional model transformation languages deal with non-determinism by requiring designers to write non-ambiguous mappings in order to obtain a deterministic result. Recently some interesting solutions have been proposed as based on lenses, unfortunately, the management of non-bijective problems is not clearly addressed, even though it could be theoretically supported by means of delta merging and conflict resolution.

Bidirectional model transformations represent at the same time an intrinsically difficult problem and a crucial mechanism for keeping consistent and synchronized a number of related models. In this paper, we tackle the problem of non-determinism in bidirectional transformations focusing on the concept of uncertainty, which represent one of the prevalent factors within software engineering. When modelers are not able to fix a design decision they may encode ambiguities in their model transformation specification, e.g. not providing additional constraints that would make the transformation deterministic. In this work we have made an attempt to help designers to give an uniform characterization of the solution in terms of models with uncertainty as already known in literature.

References

1. J. Bézivin, F. Jouault, P. Rosenthal, and P. Valduriez. Modeling in the Large and Modeling in the Small. In *Procs of European MDA Workshops*, volume 3599 of *LNCS*, pages 33–46. Springer, 2004.
2. A. Cicchetti, D. Di Ruscio, R. Eramo, and A. Pierantonio. JTL: a bidirectional and change propagating transformation language. In *SLE10*, 2010.
3. K. Czarnecki, J. N. Foster, Z. Hu, R. Lämmel, A. Schürr, and J. F. Terwilliger. Bidirectional Transformations: A Cross-Discipline Perspective - GRACE meeting notes, state of the art, and outlook. In *Procs. of ICMT2009*, volume 5563 of *LNCS*, pages 260–283. Springer, 2009.
4. C. Ebert and J. D. Man. Requirements uncertainty: influencing factors and concrete improvements. In *Procs. of ICSE*, pages 553–560. ACM Press, 2005.
5. M. Famelis, R. Salay, and M. Chechik. Partial models: Towards modeling and reasoning with uncertainty. In *ICSE*, pages 573–583, 2012.
6. M. Famelis, R. Salay, A. D. Sandro, and M. Chechik. Transformation of models containing uncertainty. In *MoDELS*, pages 673–689, 2013.
7. T. Hettel, M. Lawley, and K. Raymond. Model Synchronisation: Definitions for Round-Trip Engineering. In *Procs. of ICMT 2008*, 2008.
8. H. Ibrahim, B. H. Far, A. Eberlein, and Y. Daradkeh. Uncertainty management in software engineering: Past, present, and future. In *CCECE*, pages 7–12. IEEE, 2009.
9. R. Salay, M. Chechik, J. Horkoff, and A. D. Sandro. Managing requirements uncertainty with partial models. *Requir. Eng.*, 18(2):107–128, 2013.
10. P. Sawyer, N. Bencomo, J. Whittle, E. Letier, and A. Finkelstein. Requirements-aware systems: A research agenda for re for self-adaptive systems. In *RE*, pages 95–103. IEEE, 2010.
11. D. Schmidt. Guest Editor's Introduction: Model-Driven Engineering. *Computer*, 39(2):25–31, 2006.
12. S. Sendall and W. Kozaczynski. Model Transformation: The Heart and Soul of Model-Driven Software Development. *IEEE Software*, 20(5):42–45, 2003.

Automated chaining of model transformations with incompatible metamodels

Francesco Basciani *

Department of Information Engineering Computer Science and Mathematics
University of L'Aquila, Italy
francesco.basciani@graduate.univaq.it

Abstract. In Model-Driven Engineering (MDE) the development of complex and large transformations can benefit from the reuse of smaller ones that can be composed according to user requirements. Composing transformations is a complex problem: typically smaller transformations are discovered and selected by developers from different and heterogeneous sources. Then the identified transformations are chained by means of manual and error-prone composition processes. In this paper we propose an approach to automatically discover and compose transformations: developers provide the system with the source models and specify the target metamodel. By relying on a repository of model transformations, all the possible transformation chains are calculated. Importantly, in case of incompatible intermediate target and source metamodels, proper adapters are automatically generated in order to chain also transformations that otherwise would be discarded by limiting the reuse possibilities of available transformations.

1 Introduction

In MDE, model transformations play a key role and in order to enable their reusability, maintainability, and modularity, the development of complex transformations should be done by composing smaller ones [1]. The common way to compose transformations is to chain them [2,1,3,4,5], i.e., by passing models from one transformation to another.

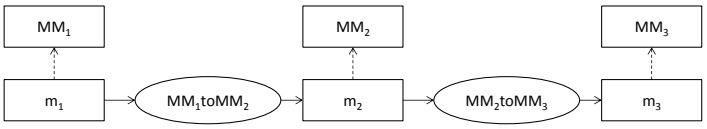


Fig. 1. Model transformation chain example

Figure 1 shows an explanatory model transformation chain. In particular, $\texttt{MM}_1\texttt{toMM}_2$ is a model transformation that generates models conforming to the target metamodel $\texttt{MM}_2$ from models conforming to $\texttt{MM}_1$. Additionally, $\texttt{MM}_2\texttt{toMM}_3$ is a model transformation that generates models conforming to $\texttt{MM}_3$ from models conforming to the source

* Davide Di Ruscio, Ludovico Iovino and Alfonso Pierantonio

metamodel $\texttt{MM}_2$. Since the input metamodel of $\texttt{MM}_2\texttt{toMM}_3$ is also the output metamodel of $\texttt{MM}_1\texttt{toMM}_2$, then these two transformations can be chained as shown in Fig. 1.

The definition of transformation chains rely on the concept of *compatible metamodels* [2] as defined below.

Definition 1 (metamodels compatibility). *Let MM_1 and MM_2 be two metamodels, then MM_1 and MM_2 are* compatible *if $MM_1 \subseteq MM_2$ or $MM_2 \subseteq MM_1$.*

Definition 2 (transformation composability). *Let $T_1 : MM_1 \rightarrow MM_2$ be a model transformation from the metamodel MM_1 to the metamodel MM_2, and let $T_2 : MM_3 \rightarrow MM_4$ be a model transformation from the metamodel MM_3 to the metamodel MM_4. Then, T_1 and T_2 are* composable *as $T_1 \circ T_2(T_2 \circ T_1)$ if $MM_4 \subseteq MM_1$ ($MM_2 \subseteq MM_3$).*

Unfortunately, restricting the definition of transformation chains only for the cases of compatible metamodels can reduce the number of chains that might be potentially obtained. Because of the metamodels compatibility concept previously defined, the transformations *Grafcet_to_PetriNet*$_{1.0}$ and *PetriNet*$_{2.0}$*_to_PNML* are not composable since *PetriNet*$_{1.0} \not\subseteq$ *PetriNet*$_{2.0}$ and *PNML* $\not\subseteq$ *Grafcet*. However, by analyzing the two versions of the PetriNet metamodel it is possible to notice that there are many commonalities that might be exploited to increase the possible transformation chains (will see the main differences in the following).

In this paper, we propose an approach that under certain conditions permits to chain model transformations defined on incompatible metamodels. This is done by means of an *adapter* transformation that can be automatically synthesized from a delta model representing the differences between the output and input metamodels of the transformations to be chained. By relying on a repository of model transformations, the system is able to automatically retrieve the model transformations that can be chained to satisfy the user request.

2 Automating Model Transformations Chaining

Our approach exploit and complement existing works by advancing the state-of-the-art in two different directions: *(i)* the user gives as input only the source model and the target metamodel, and the system automatically derives the possible chains that can satisfy the user request, *(ii)* under certain conditions the proposed approach is able to generate chains that include non-compatible transformations through synthetised metamodel adapters, as shown in Fig. 2.

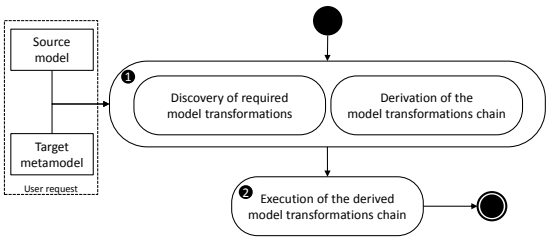


Fig. 2. Proposed model transformations chaining process

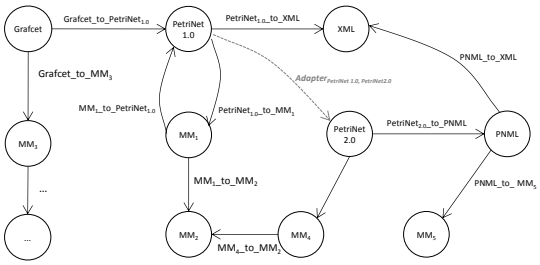


Fig. 3. Graph-based structure of a simple model transformations repository

Discovery of the required model transformations The whole activity ❶ in Fig. 2 is enabled by a novel repository of model transformations, which are stored in a directed graph-based structure as shown in Fig. 3. The nodes in the figure represent metamodels, whereas the arcs represent all the available model transformations in the repository.

Derivation of model transformation chains Representing all the available transformations as shown in Fig. 3 permits to deal with the problem of deriving a transformation chain from a source metamodel to a target one as the problem of finding paths between two nodes of a graph [9].

2.1 Managing transformations with incompatible metamodels

Considering the repository shown in Fig. 3 is not possible to satisfy the user that wants a chain from *Grafecet* to *PNML*. because PetriNet$_{1.0}$ and PetriNet$_{2.0}$ are not compatible. However, it is possible to add a new transformation ($Adapter_{PetriNet1.0,PetriNet2.0}$) that is able to adapt models conforming to PetriNet$_{1.0}$ so to enable their manipulation and transformation from *PetriNet$_{2.0}$_to_PNML*.

In order to discuss how to obtain the adapter transformation let us consider the differences between the PetriNet$_{1.0}$ and PetriNet$_{2.0}$ metamodels. In particular, the new version of the PetriNet metamodel has been obtained by operating the following changes on PetriNet$_{1.0}$:

δ_1: pull up of the attribute `name` to the new abstract metaclass `NamedElement`
δ_2: renaming of the metaclass `Net` as `PetriNet`

This discussion relates to the coupled-evolution problem that has been intensively investigate over the last years [10].

In such contexts, according to [11,12] metamodel manipulations can be classified by their corrupting or not-corrupting effects on corresponding artifacts as *non-breaking changes* (changes which do not break the conformance of models to the corresponding metamodel), *breaking and resolvable changes* (changes which break the conformance of models even though they can be automatically co-adapted) and *breaking and unresolvable changes* (changes which break the conformance of models which can not automatically co-evolved and user intervention is required).

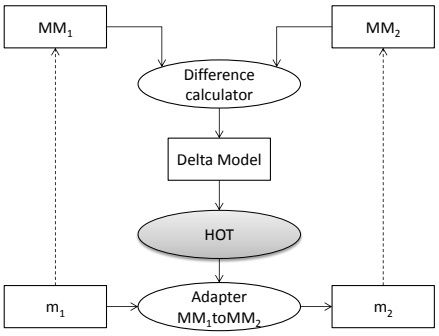


Fig. 4. Generation of the adapter transformation

We have conceived the approach shown in Fig. 4. Starting from a difference model [13] representing the differences between two incompatible metamodels, the approach is able to generate adapter transformations. The approach relies the higher-order transformation we have developed to deal with the problem of metamodel/model coupled evolution presented in [11] and subsequently refined to deal with other related co-evolution problems (e.g., see [14,15,16,17]).

The proposed approach can be applied in case of non-breaking and breaking and resolvable changes. In fact in such cases the adapter generation is completely automated without requiring user intervention. By considered the running example, since δ_1 is a non-breaking change, and δ_2 is breaking and resolvable, the approach in Fig. 4 can be applied. It is important to remark that *adapter transformations are generated when new transformations are added in the repository or deleted from it*. In particular, for each transformation addition, the corresponding source and target metamodels are taken as input by a similarity function [18,19] used to calculate a similarity value between such metamodels and all the others already stored in the repository. The similarity values are maintained in a table like the one shown in Table 1. If the similarity value between two considered metamodels is higher than a threshold (in our initial tests we have used 0,80 as threshold value) then such metamodels are further analyzed.

Implementation The implemented system consists of a J2EE application providing a Web-based front-end that users can conveniently adopt to *(i)* upload the source model, *(ii)* select the target metamodel among the available ones, *(iii)* select one of the proposed chains that are derived in a transparent manner for the user as discussed in the previous section (to help users in the selection, each chain is characterized by different attributes, like the number of single transformations that will be executed, the coverage with respect to the source metamodel, how many times the chain has been already executed, and its average execution time, these are just a few examples) , and *(iv)* remotely execute the selected chain. At the end of the process, the user can download the generated model.

	Grafcet	PetriNet$_{1.0}$	PetriNet$_{2.0}$	XML	PNML
Grafcet	1	0,2	0,30	0,29	0,26
PetriNet$_{1.0}$	-	1	**0,89**	0,20	0,28
PetriNet$_{2.0}$	0,30	**0,89**	1	0,30	0,30
XML	0,29	0,2	0,3	1	-
PNML	0,26	-	-	0,28	1

Table 1. Sample metamodel similarity values

3 Conclusions

In this paper we presented a novel approach to support the chaining of model transformations. Starting from a user request consisting of a source model, and the specification of a target metamodel, the system is able to calculate the possible chains satisfying the user request according to the transformation available in a proposed transformation repository. The main strengths of the approach proposed are related to the possibility of chaining transformations that would be discarded if only the notion of metamodel compatibility is considered.

References

1. Etien, A., Aranega, V., Blanc, X., Paige, R.F.: Chaining Model Transformations. In: Proceedings of the First Workshop on the Analysis of Model Transformations. AMT '12, New York, NY, USA, ACM (2012) 9–14
2. Etien, A., Muller, A., Legrand, T., Blanc, X.: Combining Independent Model Transformations. In: Proceedings of the 2010 ACM Symposium on Applied Computing. SAC '10, New York, NY, USA, ACM (2010) 2237–2243
3. Vanhooff, B., Baelen, S.V., Hovsepyan, A., Joosen, W.: Towards a Transformation Chain Modeling Language (2006)
4. Etien, A., Muller, A., Legrand, T., Paige, R.F.: Localized model transformations for building large-scale transformations. Software Systems Modeling (2013) 1–25
5. Wagelaar, D.: Composition Techniques for Rule-Based Model Transformation Languages. In Vallecillo, A., Gray, J., Pierantonio, A., eds.: Theory and Practice of Model Transformations. Volume 5063 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2008) 152–167
6. Jouault, F., Kurtev, I.: Transforming Models with ATL. In: Proceedings of the 2005 International Conference on Satellite Events at the MoDELS. MoDELS'05, Berlin, Heidelberg, Springer-Verlag (2006) 128–138
7. Aranega, V., Etien, A., Mosser, S.: Using Feature Model to Build Model Transformation Chains. In: Proceedings of the 15th International Conference on Model Driven Engineering Languages and Systems. MODELS'12, Berlin, Heidelberg, Springer-Verlag (2012) 562–578
8. Rivera, J.E., Ruiz-Gonzalez, D., Lopez-Romero, F., Bautista, J., Vallecillo, A.: Orchestrating ATL Model Transformations. In: Proc. of MtATL 2009, Nantes, France (2009) 34–46
9. Rubin, F.: Enumerating all simple paths in a graph. IEEE Transactions on Circuits and Systems **25** (1978) 641–642
10. Di Ruscio, D., Iovino, L., Pierantonio, A.: Coupled Evolution in Model-Driven Engineering. IEEE Software **29** (2012) 78–84
11. Cicchetti, A., Di Ruscio, D., Eramo, R., Pierantonio, A.: Automating Co-evolution in Model-Driven Engineering. In: Proceedings of the 2008 12th International IEEE Enterprise Distributed Object Computing Conference. EDOC '08, Washington, DC, USA, IEEE Computer Society (2008) 222–231
12. Wachsmuth, G.: Metamodel Adaptation and Model Co-adaptation. In Ernst, E., ed.: Proceedings of the 21st European Conference on Object-Oriented Programming (ECOOP'07). Volume 4069 of Lecture Notes in Computer Science., Springer-Verlag (2007)
13. Cicchetti, A., Di Ruscio, D., Pierantonio, A.: A Metamodel Independent Approach to Difference Representation. Journal of Object Technology **6** (2007) 165–185
14. Di Ruscio, D., Iovino, L., Pierantonio, A.: Managing the Coupled Evolution of Metamodels and Textual Concrete Syntax Specifications. In: Software Engineering and Advanced Applications (SEAA), 2013 39th EUROMICRO Conference on. (2013) 114–121
15. Di Ruscio, D., Lämmel, R., Pierantonio, A.: Automated Co-evolution of GMF Editor Models. In Malloy, B., Staab, S., Brand, M., eds.: Software Language Engineering. Volume 6563 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2011) 143–162
16. Di Ruscio, D., Iovino, L., Pierantonio, A.: Evolutionary Togetherness: How to Manage Coupled Evolution in Metamodeling Ecosystems. In: Proceedings of the 6th International Conference on Graph Transformations. ICGT'12, Berlin, Heidelberg, Springer-Verlag (2012) 20–37
17. Di Ruscio, D., Iovino, L., Pierantonio, A.: A Methodological Approach for the Coupled Evolution of Metamodels and ATL Transformations. In: Theory and Practice of Model Transformations. (2013)

18. Voigt, K.: Structural Graph-based Metamodel Matching. PhD thesis (2011)
19. Melnik, S., Garcia-Molina, H., Rahm, E.: Similarity flooding: a versatile graph matching algorithm and its application to schema matching. In: Proceedings. 18th International Conference on Data Engineering, 2002. (2002) 117–128

Similarity management via history annotation

Extended Abstract (Work in Progress) SATToSE 2014

Thomas Schmorleiz and Ralf Lämmel

Software Languages Team
University of Koblenz-Landau, Germany
http://softlang.wikidot.com

Abstract. To meet requirements software is often developed as a set of variants. In previous work [1], we have proposed an approach to manage variants by using a *virtual platform* and cloning-related operators. In this paper, we develop a tool for aggregating essential metadata for the central `propagate` operator. We discuss the aggregation process, aspects of the tool's user interface, and the implementation of `propagate`.

1 Introduction

Software is often developed as a set of variants to adapt it to conflicting user requirements, legal frameworks, or cultural constraints [1]. One approach to this is cloning, that is, copying artifacts from an existing variant to create a new variant. Cloning comes with both advantages and disadvantages [2]. While a clone can be created with minimal effort and developers are able to work independently, cloning often leads to redundancy, out-of-sync artifacts, and overall lack of control. A systematic approach for managing variants is *product line engineering* (PLE) [3]. However, transitioning from a cloning approach to PLE comes with migration issues and can lead to disruption within development teams. In previous work, we have proposed an approach to transition from cloning to a *virtual platform* which aims to achieve benefits of classical PLE while minimizing migration risks [1]. In particular, we have presented a set of operators for clone management.

In this paper, we present a tool for generating essential metadata for the central `propagate` operator. The operator pushes changes from an original variation to a cloned one or vice versa and thereby diminishes unintentional divergence. We use the *101companies* project as a running example [4]. The project aggregates implementations of a common feature model to demonstrate various software languages, technologies, and concepts. It is common practice to start the development of a new implementation by cloning an exiting one [1].

This paper makes the following contributions:

- A process for the extraction of similarities from a given Git repository.
- A web application for history annotation with cloning-related information.
- An approach to enable the `propagate` operator through annotations.

While other work on operators for clone management has been published in literature [5, 6], this paper focuses on the aggregation of metadata including

user-provided annotations for the implementation of a specific operator. Surveys on the state of the art of clone management point out necessary research for a complete clone management system [7].

This paper is organized as follows. Section 2 describes our approach to similarity management. We outline the overall process, explain relevant infrastructural components, and discuss relevant metadata. We further mention important aspects of the user interface, and discuss the propagation of changes. Section 3 discusses the current state of development and future work.

2 Similarity management

In this section, we describe our approach for managing similarities in a software repository.

2.1 Overall process

We have developed a web application which guides the overall process. Initially the user selects a Git repository from the local file system. This triggers a series of extraction tools:

1. **Script extraction.** Here we extract all operations performed throughout the history of the repository. At this point, the operations are creation, renaming, and deletion of files and committing changes.
2. **Variation extraction.** Next, we extract the names of variations from every commit point. This step also detects variation renaming.
3. **Fragment extraction.** For each commit point, each variant, and each file, we then extract a list of fragments, consecutive lines of code in a source file (see below).

The web application then provides various views of the repository (see below) to the user. The views aim to help the user to select a range of commits for which to perform further analysis:

4. **Similarity extraction.** For every selected commit point and every newly added fragment, we then extract all highly similar fragments at the commit point. We use *diff ratio* as the measure of similarity while the user sets a threshold for when two fragments should count as highly similar. As a result we store pairs of highly similar fragments.
5. **Conflict extraction.** For each found pair of highly similar fragments, we then detect if and when they diverged over time.

Finally, we show the similarities to the user in the web application. Similarities are grouped by variations and commit points, and, additionally presented as edges in the history of variations, pointing from a source fragment to a target fragment. The user then annotates the edges by declaring them to be either caused by cloning or by `ignore`. The second option is used by the web application

to no longer bother the user with the similarity. The result of the annotation is a graph where nodes are fragments in variations and edges are similarities that resulted from cloning. This cloning graph is the input of the propagation operator as described below.

2.2 Relevant infrastructure

While the web application triggers the various extractions, these in turn use other components. We use a Git API on a local repository to inspect diffs, checkout commits and read commit messages. For extracting fragments from a given source file we use as set of language-specific *fragment extractors* and *fragment locators* [8]. Fragment extractors take a source file as input and generate a set of classifier/name pairs, where a classifier is typically the name of a syntactic category of interest. For instance, for the programming language Haskell, a classifier can be a function or a data type definition. A fragment locator then returns a line range for a given source file and a classifier/name pair. We use Python to compute the text-based diff ratio as a similarity measure of two fragments.

2.3 Metadata

We differentiate between user-provided and computed metadata. The history annotations are user-provided. Each annotation is associated with a similarity edge between two fragments. The user annotates those edges that were caused by cloning. Annotations also consist of an intent, that is, a short text stating the reason why cloning and possibly changes were performed. In particular, differences may be documented as being justified by a deliberate variation. The intents will be used to decide whether change propagation can be performed automatically or requires confirmation by the user. The annotations are aggregated and a cloning graph is generated: a graph of fragments, each assigned to a variation and a file. Fragments are connected by those similarity edges which have been caused by cloning.

2.4 User interface

Due to the amount of data the user has to process it is necessary to follow good principles of user experience (UX) design. Users should make informed decisions with the help of different repository views. We currently provide an activity history of the repository and a history of variations.

The actual annotation is an iterative process. First the web applications displays the extracted similarities, group by variations and commit points, ranked by diff ratio. The user can set a threshold for the diff ratio to reduce the displayed similarities to a processable amount. The user then selects a similarity and is provided with additional information, e.g., diffs and commit messages. After annotating the edge the application shows the user detected divergences of the fragments and he or she has the option to request synchronization by merging changes. Then the next similarities are inspected.

2.5 Change propagation

The actual `propagate` operator takes the repository and the cloning graph as inputs. Changes can be pushed both in the direction of a cloning edge and from a clone to an original. The propagation might involve automatic or manual merging of changes made to the different variations. We also want to provide the operator as an extension of the Git command line tools.

3 Concluding remarks

We have developed a first version of a web application for similarity analysis and history annotation. The result of usage is a cloning graph which provides added value because it enables the application of a `propagate` operator.

We have implemented all discussed extractors except for the conflict extractor. We have a model for the cloning graph, but still have to develop a populator based on the annotations. We have refined the user interface in several iterations, but plan to add further views of the repository to help the user with the annotation step.

The tool enables the implementation of the `propagate` operator. We will look into options to extend the tool such that additional operators can be implemented. This may include an operator to push newly added feature implementations across variations or an operator for explicitly cloning an existing variation.

References

1. Michal Antkiewicz, Wenbin Ji, Thorsten Berger, Krzysztof Czarnecki, Thomas Schmorleiz, Ralf Lämmel, Stefan Stanciulescu, Andrzej Wasowski, and Ina Schaefer. Flexible product line engineering with a virtual platform. In *ICSE Companion*, pages 532–535, 2014.
2. Yael Dubinsky, Julia Rubin, Thorsten Berger, Slawomir Duszynski, Martin Becker, and Krzysztof Czarnecki. An exploratory study of cloning in industrial software product lines. In *CSMR*, 2013.
3. Klaus Pohl, Günter Böckle, and Frank J. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques.* Springer, 2005.
4. Jean-Marie Favre, Ralf Lämmel, Thomas Schmorleiz, and Andrei Varanovich. 101companies: a community project on software technologies and software languages. In *TOOLS*, 2012.
5. Julia Rubin and Marsha Chechik. A framework for managing cloned product variants. In *ICSE*, 2013.
6. Julia Rubin, Krzysztof Czarnecki, and Marsha Chechik. Managing cloned variants: A framework and experience. In *SPLC*, 2013.
7. Chanchal K. Roy, Minhaz F. Zibran, and Rainer Koschke. The vision of software clone management: Past, present, and future (keynote paper). In *CSMR-WCRE*, pages 18–33, 2014.
8. Jean-Marie Favre, Ralf Lammel, Martin Leinberger, Thomas Schmorleiz, and Andrei Varanovich. Linking documentation and source code in a software chrestomathy. In *WCRE*, pages 335–344, 2012.

Class model extraction from procedural code: Confronting a few ideas from the 2000's against the reality of an industrial system

Marianne Huchard*, Ines Ammar, Ahmad Bedja Boana, Jessie Carbonnel, Theo Chartier, Franz Fallavier, Julie Ly, Vu-Hao Alias Daniel Nguyen, Florian Pinier, Ralf Saenen, and Sebastien Villon

LIRMM, Université Montpellier 2 et CNRS
{marianne.huchard}@lirmm.fr
http://www.lirmm.fr/~huchard

Abstract. In this extended abstract, we report an ongoing experience conducted during a Master project on the migration of two industrial software systems. The project was proposed by a major IT service company (not cited here for confidentiality reasons) which would like to investigate a migration processing chain, in order to renovate legacy software composed of man-machine interfaces, databases and procedural source code. The aim of the renovation is to migrate to the object-oriented paradigm and generate new source code. Due to the limited time that the Master students had in their curriculum for this project, we restricted the study to the extraction of a class model. A processing chain was proposed and a few heuristics, taken in the literature, have been tested, with no really conclusive results. We report here the current status of the project in order to get feedback and new ideas to build for the future.

Keywords: Software reengineering, software migration, class model extraction, object identification

1 Context and problematics

Software renovation still remains a costly and time-consuming process for IT service companies, that can be viewed as a waste of resource, compared to the development of new software. Nevertheless, if timely and effective measures are not taken to regularly update the design, the source code, the documentation and all related artifacts, it may arrive the day where the software can no longer be understood by human, or compiled by the new compilers, or even ran on the new servers. The challenge is to maintain and migrate with a low cost the legacy software, before real problems arise. In this extended abstract, we report an

* The authors would like to thank the IT service companies that brought the renovation project and followed the work in progress and the master students (Luc Debène, Chaymae Regragui and Cedric Cambon) that made a tutorial for the use of Famix in the context of this project.

experience, in which we designed a processing chain for renovating a particular legacy software.

The two studied legacy software systems are part of a larger software suite, but can be analyzed independently. They are composed of code describing man/machine interface (HTML, VBScript/ASP, Javascript), Visual Basic Code (VB6), SQL procedures (SQL Server 2000) and two databases. The source code (VB and SQL) is composed of 909 functions, 30437 LOC for the largest software and 346 functions, 26042 LOC for the smallest software. One database contains 45 tables, while the other contains 103 tables.

Due to the limited time that the Master students had in their curriculum for this project, we restricted the study to the extraction of a class model. The students were divided in three groups, but they collaborated throughout the project. In the next sections, we develop the proposed approach and its current results (Section 2), then we conclude in Section 3.

2 The renovation approach

The global process The current study is concerned with specific programming languages, but we aim at proposing a generic processing chain which could be applied to other procedural and database programming languages (input) and other object-oriented language (output) of the same company. We decided to follow a classical processing chain such as that which is presented in Fig. 1, organized around an intermediate model.

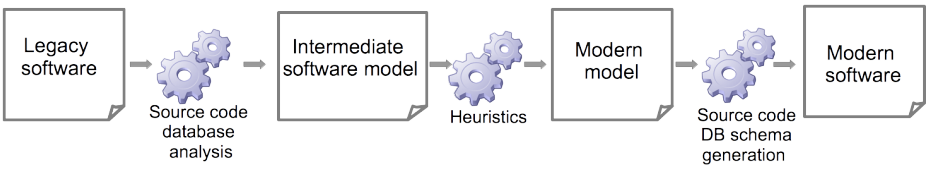


Fig. 1. The generic process

Fig. 2 shows the current instantiated process that we follow. We describe in next paragraphs the tested tools and methods of each step.

Intermediate model The choice of the intermediate model was partly guided by simplicity reasons (avoiding complex UML meta-model as proposed in UML2tools for example) and by the needs of the envisaged heuristics (study data access and function calls to determine connected sets of data and functions). In order to represent the source artifacts and the class model, we chose the FAMIX meta-model and its MSE serialization format [5]. For the representation of the output class model, this was quite natural, because of its language independency and its ability to describe the static structure of object-oriented software: main used

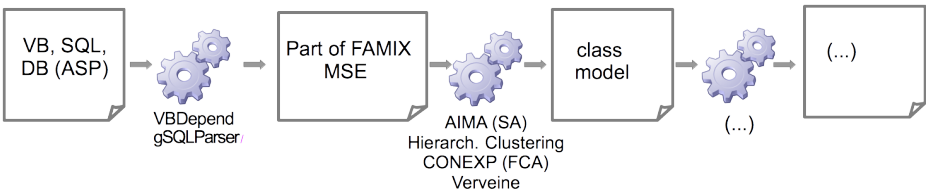


Fig. 2. The current process

concepts are meta-classse Class, Method, Attribute, Access and Invocation. For the representation of our input model, we partly used FAMIX in an unusual manner. Procedural code was easily represented using Function and Invocation. We met a problem for the representation of database elements. FAMIX being extensible, we thought that it would be the right approach to add meta-classes for representing data table and columns. Due to the delay, we provisionally abandon this track, but it remains a future work. Then, we used meta-classes Class and Attribute to respectively represent tables and columns, even if we don't consider the solution satisfactory. In this step, we found a difficulty in establishing a common strategy for the three student groups for the use of FAMIX, because initially, the groups were using different attributes of the meta-classes for representing the same information (*e.g.* using signature in Invocation meta-class, versus candidates).

Source code analysis Man-machine interface source code was partially manually analyzed, but in a first approach, we decided to abandon the track because of the little domain knowledge that this code seemed (at first sight) expose. This will be study into more details in future work. At the beginning, we would like to analyze VB code using Microsoft Visual studio, but it reveals to be impossible because of the old version of VB code. This again shows the importance to regularly update software source code. An evaluation version of VBdepend [1] was tested and allowed us to retrieve functions, invocations and parameters. As we were interested in finding the VB functions and the SQL functions that manipulated the database they called, some specific code has been developed because in the source code, this is done via a same function that takes as parameters the called SQL function and its parameters. SQL code has been analyzed with GSP [2] to extract which SQL procedure has which kind of access to which tables (and which columns of the tables). We met a few problems in using the two tools, but more than 99% of the VB functions and more than 91% of the SQL procedures were correctly analyzed. MSE entities coming from SQL analysis and from VB analysis are merged to give a unique MSE file for the remainder of the project.

Class model heuristics extraction Each student group had three or four papers about "object identification" (the term used in 2000's for talking about class

[1] http://www.vbdepend.com
[2] http://www.sqlparser.com

model extraction) and they had to choose one for implementation [9, 2–4, 7, 1, 6, 8, 10]. The three methods of [4, 6, 9] have been tried. The approach of [4] is an ad hoc method based on a hierarchical clustering technique that we had to entirely implement. The clusters are composed of columns "similarly accessed" by functions. In [6], they compare several meta-heuristics and we chose among them the simulated annealing method. The framework AIMA [3] has been used for the tests. It requires to implement a few Java interfaces by accessing the MSE files and computing a neighbor solution as well as cohesion and coupling metrics (on which an objective function is based) on the current solution (a set of candidate classes composed of data and functions). The approach of [9] uses Formal Concept Analysis to form concepts composed of functions and their accessed columns. After concept lattice building, an ad hoc algorithm merges some concepts and assigns the functions to the classes.

VerveineJ [4] is used in all implementations for finding the entities that are taken as input of heuristics and MSE files are generated that contain the output class models.

Current results The hierarchical clustering based implementation allows us to find different size solutions, depending the chosen level for stopping the clustering. It produces classes which mainly correspond to connected groups of tables of the initial databases, thus with an explainable logics. Its current issue concerns the methods assignment (only functions that access to a single class are assigned to that class), thus a complementary approach has to be proposed. The metrics to be used in simulated annealing approach and their respective weights were not sufficiently described, and for the moment we are not able to obtain the same results as those obtained by the author on the example given in the paper. The FCA approach produces many classes (compared to the database table number). Some obtained classes have a coherence for people of the company, but some seem to be incidental groups of attributes and methods, technically explained by accesses and invocations, but with no clear meaning. Besides, even if the concept lattice guarantees maximal factorization with no redundancies, the applied post-treatments generate a huge attribute and method duplication.

3 Conclusion

In this extended abstract, we reported an ongoing work that aims at extracting a class model from procedural code, relying on work mostly done in the 2000s. Some difficulties were met to apply the methods: some are dedicated to specific contexts (as COBOL code, or C code) and they are not so generic; some parameters of the methods are not very precisely described, leaving space for interpretation. As we also think that structural aspects, including data access and function invocation are not sufficient to determine meaningful classes, we plan to investigate software identifier analysis. Besides, we expect that man-machine

[3] http://code.google.com/p/aima-java/
[4] http://www.moosetechnology.org/tools/verveinej

interface code or execution trace could be an help to extract connected functionalities. Last, as these techniques have to be understood as a technical assistance for expert engineers, we would like to study how the user should intervene in the process.

References

1. Bhatti, M.U., Ducasse, S., Huchard, M.: Reconsidering classes in procedural object-oriented code. In: International Conference on Reverse Engineering (WCRE) (2008), http://rmod.lille.inria.fr/archives/papers/Bhat08b-WCRE2008-ObjectIdentification.pdf
2. Canfora, G., Cimitile, A., Lucia, A.D., Lucca, G.A.D.: A case study of applying an eclectic approach to identify objects in code. In: IWPC. pp. 136–143. IEEE Computer Society (1999)
3. Cimitile, A., Lucia, A.D., Lucca, G.A.D., Fasolino, A.R.: Identifying objects in legacy systems using design metrics. Journal of Systems and Software 44(3), 199–211 (1999)
4. van Deursen, A., Kuipers, T.: Identifying objects using cluster and concept analysis. In: Boehm, B.W., Garlan, D., Kramer, J. (eds.) ICSE. pp. 246–255. ACM (1999)
5. Ducasse, S., Anquetil, N., Bhatti, M.U., Hora, A.C., Laval, J., Girba, T.: Mse and famix 3.0 : an interexchange format and source code model family. Tech. Rep. Cutter-Deliverable 22, ANR 2010 BLAN 0219 02, RMod INRIA Lille-Nord Europe (November 2011), http://rmod.lille.inria.fr/archives/reports/Duca11c-Cutter-deliverable22-MSE-FAMIX30.pdf
6. Glavas, G., Fertalj, K.: Solving the class responsibility assignment problem using metaheuristic approach. CIT 19(4), 275–283 (2011)
7. Lucca, G.A.D., Fasolino, A.R., Guerra, P., Petruzzelli, S.: Migrating legacy systems towards object-oriented platforms. In: ICSM. pp. 122–129. IEEE Computer Society (1997)
8. Maletic, J.I., Marcus, A.: Supporting program comprehension using semantic and structural information. In: Müller, H.A., Harrold, M.J., Schäfer, W. (eds.) ICSE. pp. 103–112. IEEE Computer Society (2001)
9. Sahraoui, H.A., Lounis, H., Melo, W.L., Mili, H.: A concept formation based approach to object identification in procedural code. Autom. Softw. Eng. 6(4), 387–410 (1999)
10. Zou, Y., Kontogiannis, K.: Incremental transformation of procedural systems to object oriented platforms. In: COMPSAC. pp. 290–295. IEEE Computer Society (2003)

Towards cheap, accurate polymorphism detection

Nevena Milojković

Software Composition Group, University of Bern, Switzerland
http://scg.unibe.ch/

Abstract. Polymorphism, along with inheritance, is one of the most important features in object-oriented languages, but it is also one of the biggest obstacles to source code comprehension. Depending on the run-time type of the receiver of a message, any one of a number of possible methods may be invoked. Several algorithms for creating accurate call-graphs using static analysis already exist, however, they consume significant time and memory resources. We propose an approach that will combine static and dynamic analysis and yield the best possible precision with a minimal trade-off between used resources and accuracy.

1 Introduction

Developers often make assumptions about method invocations based on clues from source code. They need to understand their source code better. Knowing the call-graph structure at compile time would be of great benefit to the developers, and it would enhance the tools relying on static information.

While writing code, developers are often interested in the run-time behaviour of the system under development, especially in the run-time types of variables [6]. Whereas current IDEs mainly focus on the source code, and not on the dynamic behaviour, developers could benefit from both of them. Gathering the information from a running application is straightforward, presuming that the application can be instrumented and executed. However, in some cases it is not possible, or it could cause some additional costs.

Even though static and dynamic techniques exist to extract run-time information, both have shortcomings in performance, and in recall and precision. Our idea is to explore which static algorithms are affordable and give good, usable results, possibly combined with the information dynamically collected.

2 Description of the approach

In order to construct the most precise call-graphs possible in object-oriented systems, many algorithms have been developed over the years [3,1,7]. Shivers' k-CFA analysis [5] is a widely known family of control-flow analyses, accepted also in the object-oriented world, even though it was created primarily for functional languages. It has been established as being one of the most reliable analyses,

although one of the most expensive. 1- and 2-CFA are considered to be "heavy" analyses in OO languages, but feasible [4,2]. Other analyses, such as CHA and RTA analyses scale well, but, in most cases, they are not accurate enough. In general, the precision of one such algorithm is correlated with the number of nodes in the call-graph, *i.e.* the number of reachable methods, which represents a conservative approximation of a program behavior during run-time.

We intend to explore the trade-off between static and dynamic techniques to build accurate call graphs. Whereas static techniques generally produce false positives (theoretically possible but actually infeasible execution paths), dynamic techniques may produce false negatives (feasible paths that simply aren't covered). We wish to explore how static and dynamic techniques can be combined to yield higher precision at a reasonable cost. To obtain dynamic information we will execute tests where the method and the source code coverage are large enough to give us consistent results. In cases where it is possible to predict all potential inputs, the preferred way would be to run the application, under the assumption that it will finish in reasonable time.

We propose a tool that will return more precise information whenever it is possible to do so. For example, static analysis in some cases gives too ambiguous results, *i.e.* a really large set of methods possibly used at a certain call site. In such cases, our program-understanding tool could offer dynamically collected information which provides a more focused set, often more relevant for developers. Another approach would be to use an inexpensive static analysis in order to determine the call sites where polymorphism is possible, and then analyze only these call sites dynamically, thus reducing cost.

References

1. David F. Bacon and Peter F. Sweeney. Fast static analysis of c++ virtual function calls. *SIGPLAN Not.*, 31(10):324–341, October 1996.
2. Martin Bravenboer and Yannis Smaragdakis. Strictly declarative specification of sophisticated points-to analyses. *SIGPLAN Not.*, 44(10):243–262, October 2009.
3. Jeffrey Dean, David Grove, and Craig Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In W. Olthoff, editor, *Proceedings ECOOP '95*, volume 952 of *LNCS*, pages 77–101, Aarhus, Denmark, August 1995. Springer-Verlag.
4. David Van Horn Matthew Might, Yannis Smaragdakis. Resolving and exploiting the k-CFA paradox. In *PLDI*, pages 305–315, 2010.
5. Olin Grigsby Shivers. *Control-flow Analysis of Higher-order Languages of Taming Lambda*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, 1991. UMI Order No. GAX91-26964.
6. Jonathan Sillito, Gail C. Murphy, and Kris De Volder. Questions programmers ask during software evolution tasks. In *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, SIGSOFT '06/FSE-14, pages 23–34, New York, NY, USA, 2006. ACM.
7. Frank Tip and Jens Palsberg. Scalable propagation-based call graph construction algorithms. In *Proceedings of the 15th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '00, pages 281–293, New York, NY, USA, 2000. ACM.

Detecting Refactorable Clones Using PDG and Program Slicing

Extended Abstract

Ammar Hamid*

Universiteit van Amsterdam

Abstract.

Code duplication in a program can make understanding and maintenance more difficult. This problem can be reduced by detecting duplicated code, refactoring it into a separate new procedure, and replacing all the occurrences by calls to the new procedure. This paper is an evaluation and extension of the paper of Komondoor and Horwitz [3], which describes the approach and implementation of a tool, that based on Program Dependence Graph (PDG) [4] and Program Slicing [7]. The tool can find non-contiguous (clones whose components do not occur as contiguous text in the program), reordered, intertwined, refactorable clones, and display them. PDG and Program Slicing provide an abstraction that ignores arbitrary sequencing choices made by programmer, and instead captures the most important dependences (data and control flow) among program components. In contrast to some approaches that used the program text, control-flow graph, and AST, all of which are more closely tied to the lexical structure which is sometimes producing irrelevant result.

1 Approach

To detect clones in a program, we represent each procedure using its PDG. In PDG, vertex represents program statement or predicate, and edge represents data or control dependences. The algorithm performs four steps (described in the following subsections):

Step 1: Find relevant procedures
Step 2: Find pair of vertices with equivalent syntactic structure
Step 3: Find clones
Step 4: Group clones

1.1 Find relevant procedures

We are only interested in finding clones for procedures that are reachable from the main program execution. The reason for this, is that we can safely remove unreachable procedures from our program and therefore there is no need to detect clones for it. We do this by getting a system initialization vertex and do a forward-slice with data and control flow. This will return all PDGs (including user defined and system PDGs) that are reachable from the main program execution. From that result, we further filter those PDGs to find only the user defined ones, ignoring system libraries.

* email: ammarhamid84@gmail.com

1.2 Find pair of vertices with equivalent syntactic structure

We scan all PDGs from the previous step to find vertices that has type `expression` (e.g. int a = b + 1). From those expression vertices, we try to match their syntactic structure with each other. To find two expressions with equivalent syntactic structures, we make use of Abstract Syntax Tree (AST). This way, we ignore variable names, literal values, and focus only on the structure, e.g. `int a = b + 1` is equivalent with `int k = l + 1`, where both expression has the same type, which is `int`).

1.3 Find clones

From a pair of equivalent structure expressions, we do a backslice call to find their predecessors and compare them with each other. If the AST structures of their predecessors are the same then we store it in the collection of clones found. Because of this step, we can find non-contiguous, reordered, intertwined and refactorable clones. Refactorable clones in this case mean that the found clones are meaningful and it should be possible to move it into a new procedure without changing their semantic.

1.4 Group clones

This is the step where we make sense of the found clones before displaying them. As an example, using CodeSurfer [2], the vertex for a while-loop doesn't really show that it is a while loop but rather showing its predicate, e.g. `while(i<10)` will show as a control-point vertex `i<10`. Therefore, it is important that the found clones is mapped back to the actual program text representation and group them together before displaying them. Moreover, it is important that the programmer can understand and take action on the reported clones.

2 Evaluation

We are using CodeSurfer (version 2.3) to create PDG representation for the C-program to be analyzed. In CodeSurfer, we use the API, written in Scheme [1], to access those PDGs and perform all operations in the previous sections programmatically. The progress so far, with running this approach on a sample program, shows that it is pretty accurate and satisfying result. See example below:

Procedure 1	**Procedure 2**
`int foo(void) {`	`int bar(void) {`
`*  int i = 1;`	`*  int a = 1;`
`  bool z = true;`	`  bool w = false;`
`  int t = 10;`	`  int t = 10;`
`*  int j = i + 1;`	`*  int s;`
`*  int n;`	`*  int b = a + 1;`
`*  for (n=0; n<10; n++) {`	`*  for (s=0; s<10; s++) {`
`*  j = j + 5;`	`*  b = b + 5;`
`  }`	`  }`
`*  int k = i + j - 1;`	`*  int c = a + b - 1;`
`  return k;`	`  return c;`
`}`	`}`

The clones found are indicated with *. In this example, it clearly shows that not everything that has the same structure or the same syntax are reported as clones (e.g. `int t = 10;`). The reason that some vertices with equivalent syntactic structures are not included is that because they are not used anywhere and therefore can be left out safely. Only the clones which are meaningful and can be refactored into a new procedure are reported.

3 Related studies

Komondoor and Horwitz [3] proposed the use of PDG for clone detection. They were able to find isomorphic subgraphs of the PDG by implementing a program slicing technique that is using a combination of backward slicing and forward slicing. Their initial step is to find set of pair with syntactically equivalent node pairs and performed backward slicing from each pair with a single forward slicing for matching predicates nodes.

Cider [6] can detect an interprocedural clone, using Plan Calculus [5]. This algorithm can detect code clones regardless of various refactorings that may have been applied to some of the copies but not to others.

References

[1] H. Abelson, R. K. Dybvig, C. T. Haynes, G. J. Rozas, I. Adams, N. I., D. P. Friedman, E. Kohlbecker, J. Steele, G. L., D. H. Bartley, R. Halstead, D. Oxley, G. J. Sussman, G. Brooks, C. Hanson, K. M. Pitman, and M. Wand. Revised[5] Report on the Algorithmic Language Scheme. *Higher-Order and Symbolic Computation*, 11(1):7–105, 1998.

[2] GrammaTech. CodeSurfer. http://www.grammatech.com/research/technologies/codesurfer.

[3] R. Komondoor and S. Horwitz. Using Slicing to Identify Duplication in Source Code. In *Proceedings of the 8th International Symposium on Static Analysis*, SAS '01, pages 40–56. Springer-Verlag, 2001.

[4] K. J. Ottenstein and L. M. Ottenstein. The Program Dependence Graph in a Software Development Environment. In *Proceedings of the First ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, SDE 1, pages 177–184. ACM, 1984.

[5] C. Rich and R. C. Waters. *The programmer's apprentice*. ACM Press frontier series. ACM, 1990.

[6] M. Shomrat and Y. A. Feldman. Detecting Refactored Clones. In G. Castagna, editor, *Proceedings of the 27th European Conference on Object-Oriented Programming (ECOOP)*, volume 7920 of *Lecture Notes in Computer Science*, pages 502–526. Springer, 2013.

[7] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, 1984.

Analysis of developer expertise of APIs

Extended Abstract (Work in Progress) SATToSE 2014

Hakan Aksu and Ralf Lämmel

Software Languages Team
University of Koblenz-Landau, Germany
http://softlang.wikidot.com

Abstract. We analyze the version history of software projects to determine API and domain expertise of developers. In particular, we analyze the commits to a repository in terms of affected API usage. On these grounds, we can associate APIs (or parts thereof) with developers and we can thus assess API experience of developers. In transitive closure, we can also assess domain experience because of the way how (parts of) APIs are associated with programming domains.

1 Preamble

This extended abstract describes the topic of the MSc thesis by the first author. Work on this thesis is currently at the phase of methodology planning and related work research.

2 Motivation

One task of executives and project managers in IT companies or departments is to hire suitable developers and to assign them to suitable problems. Thus, developer skills must be determined. To this end, interviews, questionnaires, assignments, and publicly available information (e.g., on topcoder or stackoverflow) may be used. All existing methods are known to be "problematic". In this extended abstract, we propose an additional technique; it directly leverages previous work experience of developers in a systematic manner. That is, we analyze existing evidence for developer expertise based on the version history of existing projects. More specifically, we analyze the commits to a repository in terms of affected API usage. On these grounds, we can associate APIs (or parts thereof) with developers and we can thus assess API experience of developers. In transitive closure, we can also assess domain experience.

3 Milestones

– We review related work and best practices of MSR (mining software repositories), see, e.g., [2], to agree on methods for processing version history and discovering traceability links between commits, code, and developers.

- We leverage our prior work on API usage analysis [7, 3] and more related work on the subject [10, 9] to translate code changes into API usage data. This may also concern parts (facets) of APIs and programming domains.
- We leverage best practices on corpus usage and engineering in MSR, see, e.g., [8, 7, 5], to select suitable open-source projects as the corpus to be used in our research. A challenge is here that the analysis cannot generally assume all versions to be buildable (resolvable).
- We devise appropriate summarization and visualization techniques to be applied on the results of our analysis so that we derive an understandable and informative developer profile regarding API and domain expertise. A challenge is here that we need to map changes to skills.

4 Related work

This effort relates broadly to these research areas:

- Analysis of API usage; see, e.g., [10, 3, 7, 9]
- Analysis of changes along evolution; see, e.g., [6, 1].
- Analysis of developer activity; see below.

Our project combines all three areas with some emphasis on the last one, as far as the need for new techniques is concerned. That is, we aim at analyzing and interpreting developer activity in terms of changes along evolution based on indicators of API usage.

In [11], interactions of distributed open-source software developers are analyzed. Data mining techniques are utilized to derive developer roles. The underlying modeling and data mining techniques may be also be applicable, to some extent, to our problem in that the concepts of developer roles and (API-related) developer skills are not completely different. The concrete open-source projects of this work (ORAC-DR and Mediawiki) may also provide a starting point for our corpus.

In [4], statistical author-topic models are applied to a subset of the Eclipse 3.0 source code. The authors state that this technique provides an intuitive and automated framework with which to mine developer contributions and competencies from a given code base. The resulting information can serve as summary of developer activities and a basis for developer similarity analysis. The technique may be also be applicable, to some extent, to our problem, if we manage to identify API "topics" on the grounds of API definitions (types and method signatures) and extra metadata about domains and API facets [7].

References

1. Canfora, G., Cerulo, L., Penta, M.D.: Identifying Changed Source Code Lines from Version Repositories. In: MSR. p. 14. IEEE (2007)
2. Chaturvedi, K.K., Singh, V.B., Singh, P.: Tools in Mining Software Repositories. In: ICCSA (6). pp. 89–98. IEEE (2013)
3. Lammel, R., Linke, R., Pek, E., Varanovich, A.: A Framework Profile of .NET. In: WCRE. pp. 141–150. IEEE (2011)
4. Linstead, E., Rigor, P., Bajracharya, S.K., Lopes, C.V., Baldi, P.: Mining Eclipse Developer Contributions via Author-Topic Models. In: MSR. p. 30. IEEE (2007)
5. Pek, E.: Corpus-based Empirical Research in Software Engineering. Ph.D. thesis, University of Koblenz-Landau, Department of Computer Science (2014), available online at http://softlang.uni-koblenz.de/PekThesis.pdf
6. Robbes, R.: Mining a Change-Based Software Repository. In: MSR. p. 15. IEEE (2007)
7. Roover, C.D., Lämmel, R., Pek, E.: Multi-dimensional exploration of API usage. In: ICPC. pp. 152–161. IEEE (2013)
8. Tempero, E.D., Anslow, C., Dietrich, J., Han, T., Li, J., Lumpe, M., Melton, H., Noble, J.: The Qualitas Corpus: A Curated Collection of Java Code for Empirical Studies. In: APSEC. pp. 336–345. IEEE (2010)
9. Wang, J., Dang, Y., Zhang, H., Chen, K., Xie, T., Zhang, D.: Mining succinct and high-coverage API usage patterns from source code. In: MSR. pp. 319–328. IEEE (2013)
10. Xie, T., Pei, J.: MAPO: mining API usages from open source repositories. In: MSR. pp. 54–57. IEEE (2006)
11. Yu, L., Ramaswamy, S.: Mining CVS Repositories to Understand Open-Source Project Developer Roles. In: MSR. p. 8. IEEE (2007)

The *SoLaSoTe* ontology for software languages and technologies

Extended Abstract (Work in Progress) SATToSE 2014

Ralf Lämmel, Martin Leinberger, and Andrei Varanovich

Software Languages Team
University of Koblenz-Landau, Germany
http://softlang.wikidot.com

1 Introduction

Ontologies are increasingly used in software engineering for analysis, design, implementation, documentation, testing, and maintenance of software systems [1]. In this extended abstract, we sketch the *SoLaSoTe* ontology for software languages and software technologies, as they are used in software engineering. *SoLaSoTe* serves for *knowledge representation* and management and integration in the broad context of software languages and software technologies—as opposed to more specific categories of ontologies (such as domain ontologies, task ontologies, application ontologies, or (very) high-level ontologies [2]).

More specifically, the *SoLaSoTe* ontology represents knowledge as follows:

- *Classification* of languages and technologies as well as related concepts.
- *Dependencies* between languages and technologies.
- *Concept-based characterization* of languages and technologies.
- *Links to existing knowledge resources* for languages and technologies.
- *Traceability* for language and technology usage in *shared* software systems.

As a result, the *SoLaSoTe* ontology provides benefits as follows:

- Unambiguous terminology in the "domain" of languages and technologies.
- Identification of commonalities and differences of entities in ditto domain.
- Systematic demonstration of languages and technologies.
- Integration of otherwise scattered knowledge resources.

The continuous development of *SoLaSoTe* is linked to the *101companies* project [3] (or "101" for short) in that we leverage 101's software chrestomathy [4] with relevant components as follows:

- A wiki used to document all involved entities.
- Semantic web-like properties in said wiki to manage structured knowledge.
- A software repository to hold associated, shared software systems.
- A feature model to standardize said software systems.

In the rest of the extended abstract, we sketch the schema of *SoLaSoTe*, the process for the validation of the ontology (as represented on a semantic wiki) as well as some indications for using the ontology in queries (reasoning).

Top-level classification of entities

– Entity	*Everything in the scope of the ontology*
• Language	*Software languages such as Java or XML*
• Technology	*Software technologies such as JUnit or Eclipse*
• Concept	*Software concepts such as Visitor or Unit testing*
• Feature	*Features of 101's imaginary system*
• Contribution	*Implementations of 101's imaginary system*
• Contributor	*Contributors of code and documentation*
• Theme	*Containers of related contributions*
• Vocabulary	*Containers of domain-specific terms*
• Resource	*External resources such as standards and specifications*

Semantic properties grouped by subject entity

Entity		
instanceOf	Entity	*An instance/type relationship*
isA	Entity	*A specialization relationship*
partOf	Entity	*A whole-part relationship*
dependsOn	Entity	*Dependence relationship*
mentions	Entity	*Nonspecific reference in documentation*
sameAs	URL	*Equivalence relative to external resource*
similarTo	URL	*Similarity relative to external resource*
linksTo	URL	*Nonspecific reference to external resource*
documentedBy	Contributor	Authorship of documentation
memberOf	Vocabulary	*Membership in vocabulary of terms*
Contribution		
uses	Language	*Language usage*
uses	Technology	*Technology usage*
uses	Concept	*Concept usage*
implements	Feature	*Feature implementation*
developedBy	Contributor	*Developer of contribution*
reviewedBy	Contributor	*Reviewer of contribution*
memberOf	Theme	*Membership in theme of contributions*
basedOn	Contribution	*Indication of reuse*
varies	Contribution	*Indication of variation*
moreComplexThan	Contribution	*Indication of complexity*
Resource		
describes	Language	Language definitions, et al.
describes	Technology	API specifications, et al.
describes	Concept	Textbook, white papers, et al.
Technology		
uses	Language	*Language usage*
uses	Technology	*Technology usage*
uses	Concept	*Concept usage*
implements	Language	*Parsers, compilers, interpreters, et al.*
implements	Resource	*Compliance with a standard, et al.*
supports	Concept	*Support of a protocol, et al.*

Fig. 1. The schema of *SoLaSoTe* (with some omissions)

2 *SoLaSoTe*'s schema

The essence of the schema is shown in Fig. 1. The classification objective of *SoLaSoTe* relies on a layer of top-level classes that are simply rooted by *Entity*; see the upper part of the figure. Specialization and instantiation relationships can be applied for the classification of languages, technologies, and concepts, while additional relationships capture composition and dependence; see the properties *instanceOf*, *isA*, *partOf*, and *dependsOn* for entities. There is a family of properties concerned with the association of the ontology's entities with external resources; see *sameAs*, *similarTo*, and *linksTo*. Yet other properties concern authorship of documentation for entities and membership of entities (terms) in (sub-) vocabularies.

SoLaSoTe relies on the shared software systems of 101; they are called *contributions*; they are contributed (developed, reviewed, documented) by contributors (persons). The schema identifies a number of properties that are clearly designed to semantically connect contributions and other kinds of entities. In particular, a contribution links to languages, technologies, concepts that it *uses*. Also, a contribution links to features that it implements. To this end, a well-defined feature model helps with the standardization of the shared software systems. For what it matters, contributions are also interrelated to indicate reuse, variation of technology or design choices, and differences in complexity.

Fig. 2. Semantic properties of a contribution in 101's chrestomathy

3 *SoLaSoTe*'s validation process

The ontology (in terms of all the subtypes of *Entity* and all the actual properties) is expressed on a semantic wiki. Regular links give rise to "mentions" properties of Fig. 1. All the other properties are explicitly expressed within the documentation in designated metadata section per page. For instance, Fig. 2 shows the properties for a Java-based contribution, which exercises database technologies as well as SQL while implementing a number of 101's features.

The (simplified) process of validation commences as follows:

– Extract RDF triples from the semantic wiki:

  - *Entity* becomes a root class.
  - *Language*, *Technology*, etc. become subclasses of *Entity*.
  - The *isA* properties give rise to *rdfs:subClassOf* properties.
  - The *instanceOf* properties give rise to *rdf:type* properties.
  - All other semantic properties are adopted, as is.
- Analyze the integrity of the RDF triples:
  - All resources have an *rdf:type* property.
  - The subjects and objects of properties agree with the schema.
  - No properties other than those of the schema are used.
  - An instance is never specialized (as in OWL DL).

These integrity constraints are expressed as SPARQL queries that are obtained by a simple interpretation of the schema.[1] We skip over some details here such as dealing with cardinalities for providing warnings regarding symptoms of incompleteness in the ontology.

4 Querying *SoLaSoTe*

The ontology is not only useful for representing knowledge, but we can also query it to infer knowledge not represented explicitly. Here are a few query scenarios:

Paradigm-specific concepts Given a small set of programming paradigms, find the concepts that appear to be (more or less) uniquely associated with each paradigm—by means of collecting concepts being mentioned in the documentation of contributions, which are using programming languages of the different paradigms.

Simple baseline implementation Find the contribution that uses a given language and exercises a given concept such that there is no other contribution with less features, languages, technologies, and concepts involved.

Knowledge holder shortage Identify languages and technologies that are used infrequently by contributions without a proportional frequency of contributors who appear to be knowledgeable for these languages and technologies.

References

1. Emdad Ahmed. Use of ontologies in software engineering. In *SEDE*, pages 145–150. ISCA, 2008.
2. José R. Hilera Francisco Ruiz. Ontologies for software engineering and software technology. In *Using Ontologies in Software Engineering and Technology*, pages 49–102. Springer, 2006.
3. Jean-Marie Favre, Ralf Lämmel, Thomas Schmorleiz, and Andrei Varanovich. 101companies: A community project on software technologies and software languages. In *TOOLS (50)*, volume 7304 of *LNCS*, pages 58–74. Springer, 2012.
4. Ralf Lämmel. Software chrestomathies. *Science of Computer Programming*, 2013. In print.

[1] The approach is inspired by Stardog's ICV: http://docs.stardog.com/icv/

A three-level formal model for software architecture evolution

Abderrahman Mokni[+], Marianne Huchard*, Christelle Urtado[+], Sylvain Vauttier[+], and Huaxi (Yulin) Zhang[‡]

[+]LGI2P, Ecole Nationale Supérieure des Mînes Alès, Nîmes, France
*LIRMM, CNRS and Université de Montpellier 2, Montpellier, France
[‡] INRIA / ENS Lyon, France
{Abderrahman.Mokni, Christelle.Urtado, Sylvain.Vauttier}@mines-ales.fr,
huchard@lirmm.fr, yulinz88@gmail.com

1 Introduction

Software evolution has gained a lot of interest during the last years [1]. Indeed, as software ages, it needs to evolve and be maintained to fit new user requirements. This avoids to build a new software from scratch and hence save time and money. Handling evolution in large component-based software systems is complex and evolution may lead to architecture inconsistencies and incoherence between design and implementation. Many ADLs were proposed to support architecture change. Examples include C2SADL [2], Wright [3] and π-ADL [4]. Although, most ADLs integrate architecture modification languages, handling and controlling architecture evolution in the overall software lifecycle is still an important issue. In our work, we attempt to provide a reliable solution to the architecture-centric evolution that preserves consistency and coherence between architecture levels. We propose a formal model for our three-level ADL Dedal [5] that provides rigorous typing rules and evolution rules using the B specification language [6]. The remainder of this paper is organized as follows: Section 2 gives an overview of Dedal. Section 3 summarizes our contributions before Section 4 concludes and discusses future work.

2 Overview of Dedal the three-level ADL

Dedal is a novel ADL that covers the whole life-cycle of a component-based software. It proposes a three-step approach for specifying, implementing and deploying software architectures in a reuse-based process.
The abstract architecture specification is the first level of architecture software descriptions. It represents the architecture as designed by the architect and after analyzing the requirements of the future software. In Dedal, the architecture specification is composed of component roles and their connections. Component roles are abstract and partial component type specifications. They are identified by the architect in order to search for and select corresponding concrete components in the next step.

The concrete architecture configuration is an implementation view of the software architecture. It results from the selection of existing component classes in component repositories. Thus, an architecture configuration lists the concrete component classes that compose a specific version of the software system. In Dedal, component classes can be either primitive or composite. *Primitive component classes* encapsulate executable code. *Composite component classes* encapsulate an inner architecture configuration (*i.e.* a set of connected component classes which may, in turn, be primitive or composite). A composite component class exposes a set of interfaces corresponding to unconnected interfaces of its inner components.
The instantiated architecture assembly describes software at runtime and gathers information about its internal state. The architecture assembly results from the instantiation of an architecture configuration. It lists the instances of the component and connector classes that compose the deployed architecture at runtime and their assembly constraints (such as maximum numbers of allowed instances).

3 Summary of ongoing research

3.1 Dedal to B formalization

Dedal is a relatively rich ADL since it proposes three levels of architecture descriptions and supports component modeling and reuse. However, the present usage of Dedal is limited since there is no formal type theory for Dedal components and hence there is no way to decide about component compatibility and substitutability as well as relations between the three abstraction levels. To tackle with this issue, we proposed in [7] a formal model for Dedal that supports all its underlying concepts. The formalization is specified in B, a set-theory and first order logic based language with a flexible and simple expressiveness. The formal model is then enhanced with invariant constraints to set rules between Dedal concepts.

3.2 Intra-level and inter-level rules in Dedal

Intra-level rules in Dedal consist in substitutability and compatibility between components of the same abstraction level (component roles, concrete component types, instances). Defining intra-level relations is necessary to set the architecture completeness property:

An architecture is complete when all its required functionalities are met. This implies that all required interfaces of the architecture components must be connected to a compatible provided interface.

Inter-level rules are specific to Dedal and consist in relations between components at different abstraction levels as shown in Figure 1. Defining inter-level relations is mandatory to decide about coherence between abstraction levels.

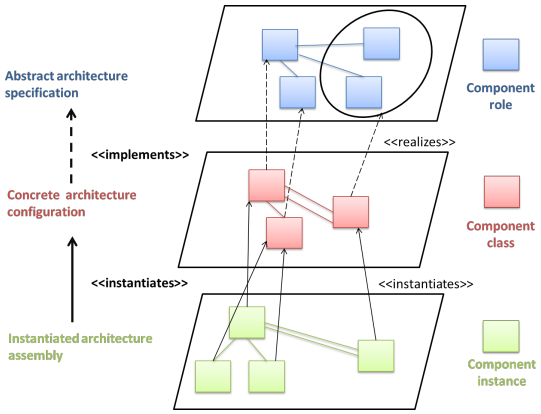


Fig. 1. Inter-level relations in Dedal

For instance, the conformance rule between a specification and a configuration is stated as follows:

A configuration C implements a specification S if and only if all the roles of S are realized by the concrete component classes of C.

3.3 Evolution rules in Dedal

An evolution rule is an operation that makes change in a target software architecture by the deletion, addition or substitution of one of its constituent elements (components and connections). Each rule is composed of three parts: the operation signature, preconditions and actions. Specific evolution rules are defined at each abstraction level to perform change at the corresponding formal description. These rules are triggered by the evolution manager when a change is requested. Firstly, a sequence of rule triggers is generated to reestablish consistency at the formal description of the initial level of change. Afterward, the evolution manager attempts to restore coherence between the other descriptions by executing the adequate evolution rules. Figure 2 presents the corresponding condition diagram of the proposed evolution process.

4 Conclusion and future work

In this paper, we give an overview of our three-level ADL Dedal and its formal model. At this stage, a set of evolution rules is proposed to handle architecture change during the three steps of software lifecycle: specification, implementation and deployment. The rules were tested and validated on sample models using a B model checker. As future work, we aim to manage the history of architecture changes in Dedal descriptions as a way to manage software system versions. Furthermore we are considering to automate evolution by integrating Dedal and evolution rules into an eclipse-based platform.

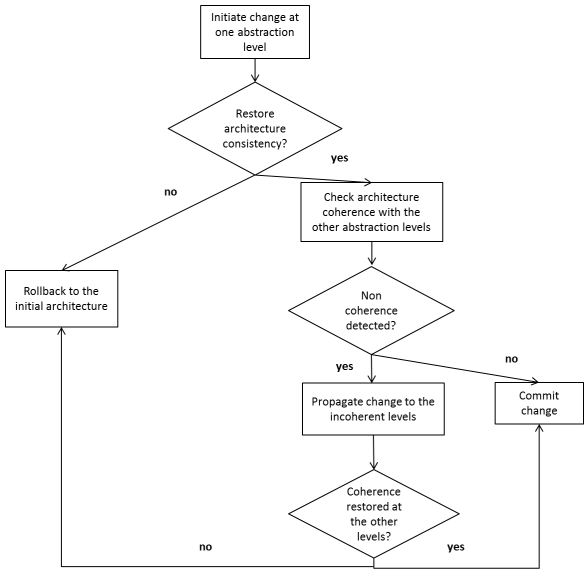


Fig. 2. Condition diagram of the evolution process

References

1. Mens, T., Serebrenik, A., Cleve, A., eds.: Evolving Software Systems. Springer (2014)
2. Medvidovic, N.: ADLs and dynamic architecture changes. In: Joint Proceedings of the Second International Software Architecture Workshop and International Workshop on Multiple Perspectives in Software Development (Viewpoints '96) on SIGSOFT '96 Workshops, New York, USA, ACM (1996) 24–27
3. Allen, R., Garlan, D.: A formal basis for architectural connection. ACM TOSEM **6**(3) (July 1997) 213–249
4. Oquendo, F.: Pi-ADL: An architecture description language based on the higher-order typed Pi-calculus for specifying dynamic and mobile software architectures. SIGSOFT Software Engineering Notes **29**(3) (May 2004) 1–14
5. Zhang, H.Y., Urtado, C., Vauttier, S.: Architecture-centric component-based development needs a three-level ADL. In: Proceedings of the 4th ECSA. Volume 6285 of LNCS., Copenhagen, Denmark, Springer (August 2010) 295–310
6. Abrial, J.R.: The B-book: Assigning Programs to Meanings. Cambridge University Press, New York, USA (1996)
7. Mokni, A., Huchard, M., Urtado, C., Vauttier, S., Zhang, H.Y.: Fostering component reuse: automating the coherence verification of multi-level architecture descriptions. Submitted to ICSEA 2014 (2014)

Adaptable Visualisation Based On User Needs

Leonel Merino

Software Composition Group, University of Bern — http://scg.unibe.ch

Abstract. Software developers often ask questions about software systems and software ecosystems that entail exploration and navigation, such as who uses this component?, and where is this feature implemented? Software visualisation can be a great aid to understanding and exploring the answers to such questions, but visualisations require expertise to implement effectively, and they do not always scale well to large systems. We propose to automatically generate software visualisations based on software models derived from open source software corpora and from an analysis of the properties of typical developers queries and commonly used visualisations. The key challenges we see are (1) understanding how to match queries to suitable visualisations, and (2) scaling visualisations effectively to very large software systems and corpora. In the paper we motivate the idea of automatic software visualisation, we enumerate the challenges and our proposals to address them, and we describe some very initial results in our attempts to develop scalable visualisations of open source software corpora.

1 Introduction

A visualisation has the advantage that it can present in a comprehensive manner a great deal of information, which makes it a good candidate for software analysis. Analyzing many systems together enables the exploration of trends and comparisons in software evolution. The users of a visualisation, whether researchers or developers, may differ in their needs. Some of them are going to be interested in visualising software quality metrics, while others will want to explore software evolution or maybe to detect code smells. To understand their specific needs we will build a taxonomy of the user needs. The taxonomy should define the means that will be provided to the user for interacting with the visualisation (e.g. to inspect entities, to launch new visualisations based on the selected entity), and the type of information he is going to receive (e.g. method attributes, invocation sender). The taxonomy should produce a classification of user needs, such as most complex methods, extent of polymorphism, method invocations or class hierarchy length.

1.1 Automatic Software Visualisation

We want to provide a query mechanism to adapt automatically the visualisation to the specific user needs. The mechanism should provide means to specify the

kind of attributes to be explored (e.g. the kind of entity, the associated properties, and the corresponding relations). For instance, we could specify that we want to visualise at a package level a view with the method invocations to others packages in the system. The mechanism should automatically select the most appropriate visualisation that satisfies our needs by choosing the right shapes, colours, layout and navigation. The visualisation should allow the user to explore and to interact with entities, to inspect them and to launch new visualisations based on the entity selected. For instance, in a visualisation that shows methods, classes, packages and systems, if we select a system it could provide means to launch a new visualisation with the evolution history of the versions of that system. Although related work supports this kind of interactions, they have to be specified manually.

1.2 Visualisations for Large Systems and Corpora

From a scalability point of view, one of the most difficult challenges in software visualisation is representing corpora of software systems. A corpus can contain hundreds of software systems, so the requirements on memory and processing power can be much higher than those for visualising individual software systems. Also the effort involved in generating the visualisation is higher: once a corpus is downloaded, we have to extract models from all the included systems. By querying attributes of the model we can extract facts about the systems. The visualisation should also provide means to navigate and to interact with those entities and relations.

2 Early results

As a first approach we wanted to cope with the visualisation of large systems and corpora. Open-source software fits very well this kind of analysis since it gives us access the source code of complete systems. Qualitas Corpus [4] and SqueakSource [2] are two available corpora. Pangea [3] is a tool that enables running language independent analyses on corpora of object-oriented software. It provides Moose [1] models for Qualitas Corpus and SqueakSource systems allowing us to create Moose images for every system and interact with them by running scripts. Qualitas Corpus comprises 112 open source Java systems, 58,557 classes, almost 15 MLOC and 754 versions, while SqueakSource contains 28 Smalltalk systems.

For our visualisation we needed a tool to depict many components in a comprehensive manner, in the sense that fine and coarse grained entities can be distinguished at a sight. Although there are many tools for creating visualisations of software, not many are suitable for large systems. Even fewer tools can support visualising many systems at the same time or more than one corpus at the time.

Figure 1 presents five systems of Qualitas Corpus: *AspectJ, ArgoUML, ANTLR, AOI, Axion* comprising more than 1.5MLOC. Since in this example we

wanted to visualise the use of polymorphism at a fine-grained level, a TreeMap is an appropiate layout. We used red color to indicate the presence of polymorphism, this is done by an heuristic that marks as polymorphic the methods of interfaces that have more than one implementation. We colored every level of the hierarchy such as method, class and package. The intensity of the color is related to the number of lines of code involved in their polymorphic methods.

The main issues that we foresee are memory use and processing time. This visualisation took a bit more than 8 minutes to be rendered and required to load almost 300MB of Moose models. We implemented this as a proof-of-concept and we did not do any optimisation yet. Since our intent is to provide a visualisation of different corpora at the same time we will need to overcome these contraints.

Acknowledgments

We gratefully acknowledge the financial support of the Swiss National Science Foundation for the project "Agile Software Assessment" (SNSF project No. 200020-144126/1, Jan 1, 2013 - Dec. 30, 2015). This work has been partially funded by CONICYT BCH/Doctorado Extranjero 72140330.

References

1. Stéphane Ducasse, Michele Lanza, and Sander Tichelaar. Moose: an Extensible Language-Independent Environment for Reengineering Object-Oriented Systems. In *Proceedings of CoSET '00 (2nd International Symposium on Constructing Software Engineering Tools)*, June 2000. URL: `http://scg.unibe.ch/archive/papers/Duca00bMooseCoset.pdf`.
2. Adrian Lienhard and Lukas Renggli. Squeaksource — smart monticello repository. European Smalltalk User Group Innovation Technology Award, August 2005. Won the 2nd prize. URL: `http://scg.unibe.ch/archive/reports/Lien05b.pdf`.
3. SCG: Pangea 2.0. URL: `http://scg.unibe.ch/research/pangea`.
4. E. Tempero, C. Anslow, J. Dietrich, T. Han, Jing Li, M. Lumpe, H. Melton, and J. Noble. The qualitas corpus: A curated collection of java code for empirical studies. In *Software Engineering Conference (APSEC), 2010 17th Asia Pacific*, pages 336 –345, December 2010. `doi:10.1109/APSEC.2010.46`.

Fig. 1. Visualisation of an heuristic of polymorphism in five systems: *AspectJ, ArgoUML, ANTLR, AOI, Axion*. More than 1.5MLOC at a glance.

A tale about software profiling, debugging, testing, and visualization

Alexandre Bergel

http://bergel.eu
Pleiad Lab, Department of Computer Science (DCC), University of Chile

Programming as a modern activity. When I was in college, I learned programming with C and Pascal using a textual and command-line programming environment. At that time, about 15 years ago, Emacs was popular due to its sophisticated text editing capacities. The `gdb` debugger allows one to manipulate the control flow including the step-into, step-over, and restart operations. The `gprof` code execution profiler indicates the share of execution time for each function, in addition to the control flow between each method.

Nowadays, object-orientation is compulsory in university curriculum and mandatory for most software engineering positions. Eclipse is a popular programming environment that greatly simplifies the programming activity in Java. Eclipse supports sophisticated options to search and navigate among textual files. Debugging object-oriented programs is still focused on the step-into, step-over and restart options. Profiling still focuses on the method call stack: the JProfiler and YourKit profilers happily output resource distributions along a tree of methods.

Sending messages is largely perceived as a major improvement over *executing functions*, which is the key to polymorphism. Whereas programming languages have significantly evolved over the last two decades, most of the improvements on programming environments do not appear to be a breakthrough. Navigating among software entities often means searching text portions in text files. Profiling is still based on methods executions, completely discarding the notion of objects. Debugging still comes with its primitive operations on stack; again ignoring objects.

This paper presents the research line carried out by the author and his collaborators on making programming environments closer to the object-oriented paradigm. Most of the experiences and case studies summarized below have been carried out in Pharo[1] – an object-oriented and dynamically typed programming language.

Profiling. Understanding the root of a performance drop or improvement requires analyzing different program executions at a fine grain level. Such an analysis involves dedicated profiling and representation techniques. Two recognized profilers – JProfiler and YourKit – both fail at providing adequate metrics and visual representations, conveying a false sense of the performance variation root.

We have proposed *performance evolution blueprint*, a visual support to precisely compare multiple software executions [1]. The performance evolution

[1] `http://pharo.org`

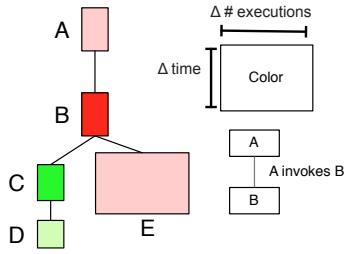


Fig. 1: Performance evolution blueprint

blueprint is summarized in Figure 1. A blueprint is obtained after running two executions. Each box is a method. Edges are invocations between methods (a calling method is above the called methods). Height of a method is the difference of execution time between the two executions. If the difference is positive (i.e., the method is slower), then the method is shaded in red, otherwise it is green. The width of a method is the absolute difference in the number of executions, thus always positive. Light red / pink color means the method is slower, but its source code has not changed between the two executions. If red the method is slower and the source code has changed. Light green indicates a faster non-modified method. Green indicates a faster modified method.

Our blueprint accurately indicates roots of performance improvement or degradation. We have developed Rizel, a code profiler to efficiently explore performance of a set of benchmarks against multiple software revisions.

Testing. Testing is an essential activity when developing software. It is widely acknowledged that a test coverage above 70% is associated with a decrease in reported failures. After running the unit tests, classical coverage tools output the list of classes and methods that are not executed. Simply tagging a software element as covered may convey an incorrect sense of necessity: executing a long and complex method just once is potentially enough to be reported as 100% test-covered. As a consequence, a developer may receive an incorrect judgement as to where to focus testing effort.

By relating execution and complexity metrics, we have identified essential patterns to characterize the test coverage of a group of methods [2]. Each pattern has an associated action to increase the test coverage, and these actions differ in their effectiveness. We empirically determined the optimal sequence of actions to obtain the highest coverage with a minimum number of tests. We present TEST BLUEPRINT, a visual tool to help practitioners assess and increase test coverage by graphically relating execution and complexity metrics. Figure 2 is an example of a test blueprint. Two versions of the same class is represented. Inner small boxes represent methods. The size of a method indicates its cyclomatic complexity. Taller a method is, more complex it is. Edges are invocations between methods, statically determined. Red color indicates uncovered methods. The figure shows

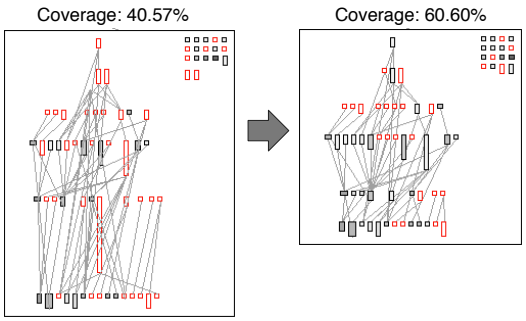


Fig. 2: Test blueprint

an evolution of a class in which complex uncovered methods have been broken down into simpler methods.

Debugging. During the process of developing and maintaining a complex software system, developers pose detailed questions about the runtime behavior of the system. Source code views offer strictly limited insights, so developers often turn to tools like debuggers to inspect and interact with the running system. Unfortunately, traditional debuggers focus on the runtime stack as the key abstraction to support debugging operations, though the questions developers pose often have more to do with objects and their interactions.

We have proposed *object-centric debugging* as an alternative approach to interacting with a running software system [3]. By focusing on objects as the key abstraction, we show how natural debugging operations can be defined to answer developer questions related to runtime behavior. We have presented a running prototype of an object-centric debugger, and demonstrated, with the help of a series of examples, how object-centric debugging offers more effective support for many typical developer tasks than a traditional stack-oriented debugger.

Visual programming environment. Visualizing software-related data is often key in software developments and reengineering activities. As illustrated above in our tools, interactive visualizations play an important intermediary layer between the software engineer and the programming environment. General purpose libraries (e.g., D3, Raphaël) are commonly used to address the need for visualization and data analytics related to software. Unfortunately, such libraries offer low-level graphic primitives, making the specialization of a visualization difficult to carry out.

Roassal is a platform for software and data visualization. Roassal offers facilities to easily build domain-specific languages to meet specific requirements. Adaptable and reusable visualizations are then expressed in the Pharo language. Figure 3 illustrates two visualizations of a software system dependencies. Each class is represented as a circle. On the left-hand side, gray edges are inheritance (the top superclass is at the center) and blue lines are dependencies between

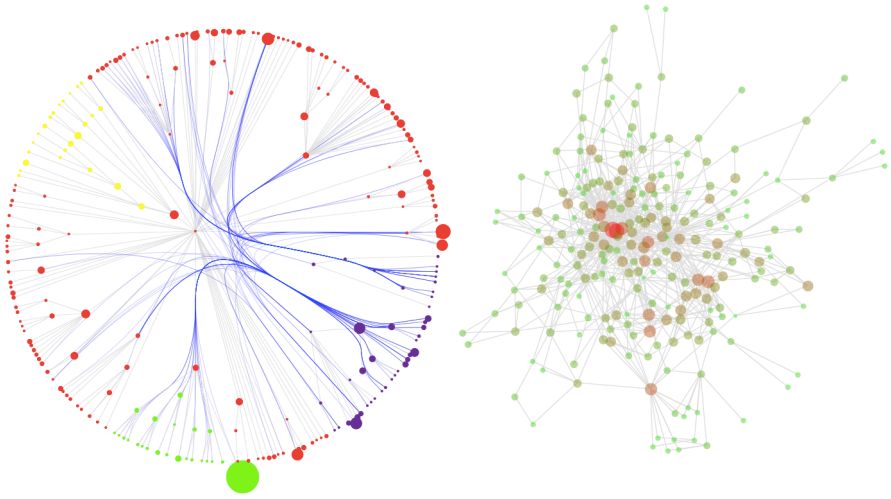


Fig. 3: Visualization of a software system

classes. Each color indicates a component. On the right-hand side, edges are dependencies between classes whereas class size and color indicate the size of the class. Roassal has been successfully employed in over a dozen software visualization projects from several research groups and companies.

Future work. Programming is unfortunately filled with repetitive and manual activities. The work summarized above partially alleviates this situation. Our current and future research line is about making our tools not only object-centric, but domain-centric. We foresee that being domain specific is a way to make tools closer to practitioners, and therefore more accepted.

References

1. J. P. S. Alcocer, A. Bergel, S. Ducasse, M. Denker, Performance evolution blueprint: Understanding the impact of software evolution on performance, in: A. Telea, A. Kerren, A. Marcus (Eds.), VISSOFT, IEEE, 2013, pp. 1–9.
2. A. Bergel, V. P. na, Increasing test coverage with hapao, Science of Computer Programming 79 (1) (2012) 86–100. doi:10.1016/j.scico.2012.04.006.
3. J. Ressia, A. Bergel, O. Nierstrasz, Object-centric debugging, in: Proceeding of the 34rd international conference on Software engineering, ICSE '12, 2012. doi:10.1109/ICSE.2012.6227167.
   URL http://scg.unibe.ch/archive/papers/Ress12a-ObjectCentricDebugging.pdf

Survivability of Software Projects in GNOME – A Replication Study

Tom Mens[1], Mathieu Goeminne[1], Uzma Raja[2], and Alexander Serebrenik[3]

[1] Software Engineering Lab, Department of Computer Science, Faculty of Sciences
University of Mons, Belgium
Email: { tom.mens | mathieu.goeminne }@umons.ac.be
[2] Department of Information Systems, Statistics and Management Science
University of Alabama, Tuscaloosa, USA
Email: uraja@cba.ua.edu
[3] Department of Mathematics and Computer Science
Eindhoven University of Technology, The Netherlands
Email: a.serebrenik@tue.nl

Abstract. This extended abstract contains a summary of our submitted ICSME 2014 paper that reports on a replication study of Open Source Software project survivability. The original study was conducted on 136 SOURCEFORGE projects and proposed a predictive model of project inactivity based on a multidimensional measure of Viability. We replicated the study by considering 183 projects from the GNOME ecosystem and by re-operationalizing the measure of Viability in order to accommodate the unique characteristics of interacting projects belonging to the GNOME ecosystem. Results of our replication study reveal that the Viability Index model has significant ability to identify projects that become inactive compared to the ones that remain active in the ecosystem.

1 Introduction

Software ecosystems represent a successful model for developing collection of software systems sharing a common goal that is the subject of a growing interest from the research community, as can be witnessed by a recent systematic literature review [7]. Messerschmitt and Szyperski [8] define a software ecosystem as "a collection of software products that have some given degree of symbiotic relationships," and Lungu [6] defines it as "a collection of software projects which are developed and evolve together in the same environment." In such ecosystems, communities of developers collaborate together, often on a voluntary basis, while users and developers of the software can submit bug reports and requests for changes [3].

Since maintaining software projects requires considerable effort and investment, understanding which projects are more likely to fail can be useful to take preventive and corrective actions to improve these projects, to discontinue their development, or to take other decisions at the ecosystem level. In our paper, we carry out a replication study of the predictive model of project survivability for

SourceForge [9], by adapting the model and evaluating its predictive power in the context of the Gnome ecosystem. To facilitate reproducibility of our replication study, all the extracted data used for our analysis as well as the tooling and metrics used have been made available on
https://bitbucket.org/mgoeminne/gnome-survivability/downloads.

2 Experimental Setup

In [9], *Viability* was defined as the basis of a predictive model to assess open source project survivability. Viability aims to reflect the ability of a project to grow and maintain its structure in the presence of perturbations. Viability was defined as a complex, multidimensional measure involving three dimensions: *Vigor*, *Resilience* and *Organization*.

Vigor represents the ability of a project to grow over a period of time. In our replication, we defined $V(p)$ the Vigor of a Gnome project p to take into account the cumulative growth of a project and its team.

Resilience reflects the ability of a project to recover from internal or external perturbations, which are interpreted as changes in the project's operating environment. We defined $R(p)$ the resilience of a Gnome project p as the number of distinct contributors involved in its bug tracker divided by the mean time to resolve issues for this project.

Organization represents the amount of structure exhibited by the interaction between software project contributors. In our replication study, we operationalized the Organisation of a Gnome project p, called $O(p)$, by using the Simpson index, a well-known measure of ecosystem diversity [10].

Finally, we operationalize *Viability* by defining a viability index *VI*.

$$VI\,(\mathrm{p}) = \alpha + \beta_1 V(p) + \beta_2 R(p) + \beta_3 O(p) \tag{1}$$

We decided to determine lack of project survival for Gnome projects in terms of inactivity in its version repository. We consider a project as being inactive if there were no commits to its version repository during the 365 days before the date on which we extracted the source code repository (i.e., on January 2013).

197 Gnome projects use an both an official Git source code repository and an official Bugzilla-based bugtracking system. We extracted the data contained in these data sources by using CVAnaly2 and Bicho, We extracted the official Git repositories and the official mailing list of these project by using two tools belonging to the MetricsGrimoire toolkit (see metricsgrimoire.github.io). During data extraction, we carried out identity merging [2,4]. The extracted data was stored in two relational SQL databases.

3 Analysis

Using the viability index *VI* described above, we statistically analyse whether we can use V, R and O as effective predictors of active and inactive Gnome projects.

The original data was largely skewed (very high values for very few projects). We also observed a difference in scale of the three variables. For better interpretability of the results, we therefore performed logarithmic transformation on our variables. This transformation allows the three variables to have a comparable range of values [5]. We conducted a Shapiro-Wilk normality test that failed to reject the null hypothesis, thereby providing support that the data is normal for the three considered measures.

To test the predictive power of the predictor variables V, R and O, we need to test how well they can predict the outcome. In this study, a project will be classified as either active or inactive. We refer to this outcome as $VI = 0$ or 1. Since this is a binary result, we cannot use traditional techniques like least square regression that only allow for numeric prediction. Logistic Regression (LR) is a statistical technique that determines the impact of multiple independent variables simultaneously to predict a binary or a categorical outcome [1, p. 67]. The logarithmic transformation we carried out does not affect the outcome of LR. Therefore, it appears to be a well-suited technique.

LR analysis identifies if the predictor variables are significant in accurately categorizing the outcome as high (VI=1) or low (VI=0). It also computes the relative odds ratio for the predictor variables. Odds ratios [1, p. 28] are a statistical mechanism to gauge the impact of a variable on the probability of the outcome being 0 or 1. In our case, we establish that using the predictor variables has a statistically significant improvement in predicting the outcome.

Next, we use the Goodness of Fit for determining whether the predictor variables have enough information or whether we need a more complex model including additional dimensions of viability. We reject the null hypothesis that states that adding new variables (or interactions or polynomials of existing predictors) significantly improves the model.

Finally, we show that each of the individual predictor variables we use significantly improves the prediction of VI.

4 Discussion and Future Work

We established in the previous section that all three dimensions of *Viability* are significant and sufficient in discriminating active and inactive projects in the Gnome ecosystem. However, we observe that the results for Gnome differ from the original SourceForge study. For SourceForge projects it was noticed that all three dimensions are significant for a project to be viable, yet a project could have low values on one or more dimensions and still be successful based on other dimensions. In contrast, *active* Gnome projects predominantly have higher values of V, R and O. This difference could be because of the differences in the nature of SourceForge and Gnome. While SourceForge projects are independent (for most cases), the Gnome ecosystem is relatively networked compared to SourceForge. We also observed some differences for the O metric that may be attributed to the fact that we have used Simpson's index as a measure of diversity, whereas the original study used an entropic measure of

Mutual Information. Further studies need to be conducted to investigate the impact of changes in organization.

From a statistical point of view, our findings support the original study in terms of statistical significance of all three dimensions of *Viability*. Like the original study, we found *Vigor* to be the most significant indicator of project success. A unit change in *Vigor* has the biggest impact on a positive outcome. However, we also observe that both *Resilience* and *Organization* have odds ratio on the same order of magnitude as *Vigor*. Since the original study did not use scaled values of the predictor variables, it is hard to infer whether the differences in the impact of the three predictors is a reflection of scale differences or a difference in how the predictors impact the outcome.

The misclassification rate of our study is still within acceptable range, however, we are investigating the cause of weaker performance of our model for *inactive* projects. This might be indicative of the need to improve our metrics.

Acknowledgment

This work is partially supported by research projects FRFC PDR T.0022.13, financed by the F.R.S.-FNRS, and Action de Recherche Concertée AUWB-12/17-UMONS-3, financed by the Ministère de la Communauté française – Direction générale de l'Enseignement non obligatoire et de la Recherche scientifique, Belgium.

References

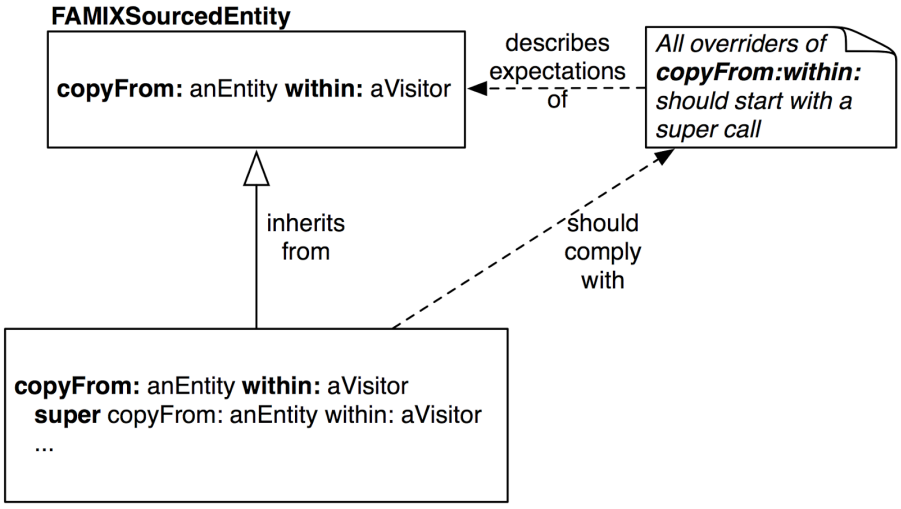
1. Alan Agresti. *An Introduction to Categorical Data Analysis*. Wiley-Interscience, 2nd edition, 2007.
2. Mathieu Goeminne and Tom Mens. A comparison of identity merge algorithms for software repositories. *Science of Computer Programming*, 78(8):971–986, August 2013.
3. Mathieu Goeminne and Tom Mens. *Software Ecosystems: Analyzing and Managing Business Networks in the Software Industry*, chapter Analyzing ecosystems for open source software developer communities. Edward Elgar, 2013.
4. Erik Kouters, Bogdan Vasilescu, Alexander Serebrenik, and Mark G. J. van den Brand. Who's who in Gnome: using LSA to merge software repository identities. pages 592–595. IEEE, 2012.
5. Loet Leydesdorff and Stephen Bensman. Classification and powerlaws: The logarithmic transformation. *Journal of the American Society for Information Science and Technology*, 57:1470–1486, 2006.
6. Mircea Lungu. Towards reverse engineering software ecosystems. pages 428–431, 2008.
7. Konstantinos Manikas and Klaus Marius Hansen. Software ecosystems: A systematic literature review. 86(5):1294–1306, May 2013.
8. D.G. Messerschmitt and C. Szyperski. *Software ecosystem: Understanding and indispensable technology and industry*. MIT Press, 2003.
9. Uzma Raja and Marietta J. Tretter. Defining and evaluating a measure of open source project survivability. 38(1):163–174, 2012.
10. E.H. Simpson. Measurement of diversity. *Nature*, 163(688), 1949.

Usage Contracts

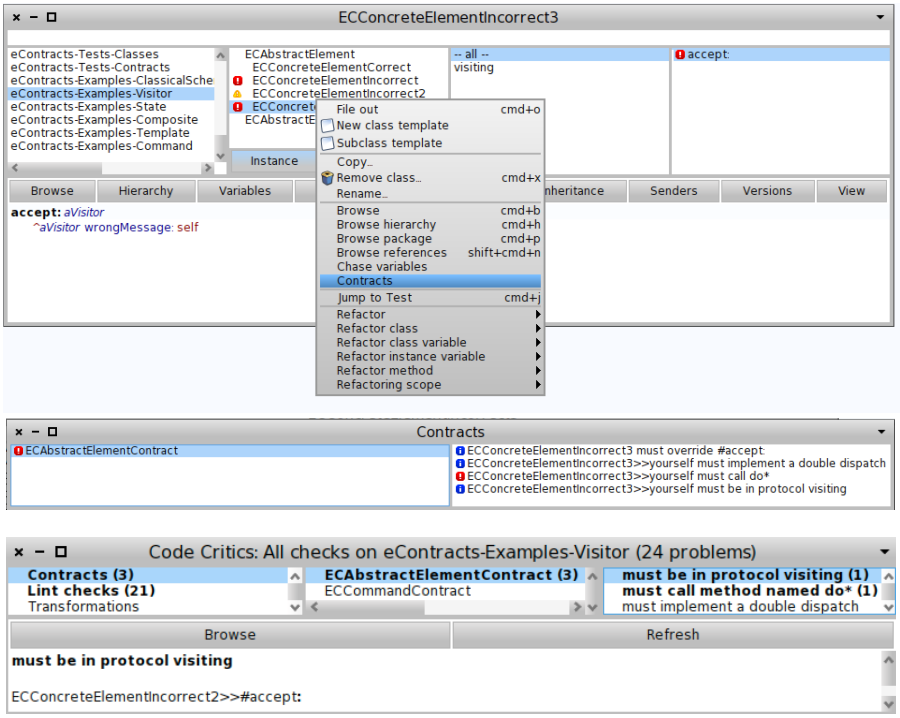
Kim Mens[1], Angela Lozano[1,2], and Andy Kellens[2]

[1] Université Catholique de Louvain
Place Sainte Barbe 2
Louvain-la-Neuve, Belgium
[2] Vrije Universiteit Brussel
Pleinlaan 2
1050 Brussels - Belgium
kim.mens@uclouvain.be, alozano@soft.vub.ac.be, akellens@vub.ac.be

Developers often encode design knowledge through structural regularities such as API usage protocols, coding idioms and naming conventions. As these regularities express how the source code should be structured, they provide vital information for developers (re)using that code. Adherence to such regularities tends to deteriorate over time when they are not documented and checked explicitly.



Our *uContracts* tool and approach allows codifying and verifying such regularities as 'usage contracts'. The contracts are expressed in an internal domain-specific language that is close to the host programming language, the tool is tightly integrated with the development environment and provides immediate feedback during development when contracts get breached, but the tool is not coercive and allows the developer to decide if, when and how to correct the broken contracts (the tool just highlights the errors and warnings in the integrated development environment). In spirit, the approach is very akin to unit testing, except that we do not test behaviour, but rather verify program structure.



The tool, of which some screenshots can be found above, was prototyped in an older version of the Pharo dialect of the Smalltalk programming language.

Author Index