CEUR Workshop Proceedings
http://ceur-ws.org
ISSN 1613-0073

CEUR-WS.org/Vol-1354
urn:nbn:de:0074-1354-1

Copyright © 2015 for the individual papers by the papers' authors. Copying permitted for private and academic purposes. This volume is published and copyrighted by its editors.

SATToSE 2014
Seminar on Advanced Techniques and Tools for Software Evolution

Post-proceedings of the Seventh Seminar on Advanced Techniques and Tools for Software Evolution

L'Aquila, Italy, 9–11 July 2014.

Edited by

Davide Di Ruscio *
Vadim Zaytsev **

* Department of Information Engineering, Computer Science and Mathematics, University of L'Aquila, Italy
** Institute of Informatics, University of Amsterdam, The Netherlands

Table of Contents

- SATToSE 2014: The Post-proceedings (editorial) 1–5
  Davide Di Ruscio, Vadim Zaytsev
- Profiling, Debugging, Testing for the Next Century 6–12
  Alexandre Bergel
- "What Programmers Do with Inheritance in Java" - Replicated on Source Code 13–24
  Çiğdem Aytekin
- Explora: Infrastructure for Scaling Up Software Visualisation to Corpora
  Leonel Merino, Mircea Lungu, Oscar Nierstrasz 25–36
- Detecting Refactorable Clones by Slicing Program Dependence Graphs
  Ammar Hamid, Vadim Zaytsev 37–48
- A Critique on Code Critics 49–59
  Angela Lozano, Gabriela Arévalo, Kim Mens
- User Interface Level Testing with TESTAR; What about More Sophisticated Action Specification and Selection? 60–78
  Sebastian Bauersfeld, Tanja E. J. Vos
- Qualifying Chains of Transformation with Coverage-based Evaluation Criteria 79–89
  Francesco Basciani, Juri Di Rocco, Davide Di Ruscio, Ludovico Iovino, Alfonso Pierantonio
- Describing the Correlations between Metamodels and Transformations Aspects 90–101
  Juri Di Rocco, Davide Di Ruscio, Ludovico Iovino, Alfonso Pierantonio
- A Three-level Formal Model for Software Architecture Evolution 102–111
  Abderrahman Mokni, Marianne Huchard, Christelle Urtado, Sylvain Vauttier, Huaxi (Yulin) Zhang
- Representing Uncertainty in Bidirectional Transformations 112–121
  Romina Eramo, Alfonso Pierantonio, Gianni Rosa
- Towards a Taxonomy for Bidirectional Transformation 122–131
  Romina Eramo, Romeo Marinelli, Alfonso Pierantonio
- Languages, Models and Megamodels (a tutorial) 132–143
  Anya Helene Bagge, Vadim Zaytsev

2015-04-30: submitted by Vadim Zaytsev, metadata incl. bibliographic data published under Creative Commons CC0
2015-05-02: published on CEUR-WS.org |valid HTML5|

SATToSE 2014: The Post-proceedings Editorial

Davide Di Ruscio[†] and Vadim Zaytsev[‡]

[†]University of L'Aquila, Italy, davide.diruscio@univaq.it
[‡]Universiteit van Amsterdam, The Netherlands, vadim@grammarware.net

Venue

SATToSE is the Seminar Series on Advanced Techniques and Tools for Software Evolution. Its previous editions has happened in Waulsort (Belgium, 2008), Côte d'Opale (France, 2009), Montpellier (France, 2010), Koblenz (Germany, 2011, 2012) and Bern (Switzerland, 2013). Its seventh edition took place in L'Aquila (Italy) on 9–11 July 2014. Each edition of SATToSE witnesses presentations on software visualisation techniques, tools for coevolving various software artefacts, their consistency management, runtime adaptability and context-awareness, as well as empirical results about software evolution.

The goal of SATToSE is to gather both undergraduate and graduate students to showcase their research, exchange ideas, improve their communication skills, attend and contribute technology showdown and hackathons.

The highlights of the programme included five invited lectures (given by Marianne Huchard, Leon Moonen, Alfonso Pierantonio, Alexander Serebrenik, Tanja Vos), an interactive tutorial (by Anya Helene Bagge) and a hands-on hackathon (by Alexander Serebrenik). The detailed programme, as well as the pre-proceedings volume with moderated but not yet reviewed abstracts and drafts can be found on our website: http://sattose.org/2014.

Selection process

We would like to express our gratitude to the following people (listed in lexicographic order) who provided the reviews for the submissions of this volume:

- Gabriela Arévalo
- Çiğdem Aytekin
- Anya Helene Bagge
- Antonio Cicchetti
- Juri Di Rocco
- Davide Di Ruscio
- Mike Godfrey
- Mathieu Goeminne
- Ammar Hamid
- Michiel Helvensteijn
- Ludovico Iovino
- Andy Kellens
- Xavier Le Pallec
- Angela Lozano
- Mircea Lungu
- Tom Mens
- Oscar Nierstrasz
- Bogdan Vasilescu
- Sylvain Vauttier
- Tanja Vos

The call for post-proceedings contributions was communicated to all participants after the event — only some decided to pursue the finalisation of their contribution for the post-proceedings, others often solicited more co-authors, changed the title, included more results. As a result, we have received 12 submissions: one of them was a keynote add-on, one was a tutorial write-up, the rest were being extended versions of pre-proceedings abstracts.

Each submitted report has been reviewed by at least three different peers. All submissions with a conflict of interest with one of the editors (co-authored by them or their colleagues) were handled by the other editor. The emphasis was put on clear problem definitions and descriptions of advanced aspects of the techniques contemplated in the solution, as opposed to the finality of the obtained results. Thus, most submissions are intermediate reports on ongoing work or summaries of previously developed tools and papers. The authors received enough time after the review to take the feedback into account and finalise their submissions.

Organisation

- **General Chair:** Davide Di Ruscio (University of L'Aquila)
- **Program Chair:** Vadim Zaytsev (Universiteit van Amsterdam)
- **Hackathon Chair:** Alexander Serebrenik (TU Eindhoven)
- **Steering Committee Chair** Ralf Lämmel (Universität Koblenz)
- **Local Committee:**
  - Romina Eramo (University of L'Aquila)
  - Ludovico Iovino (University of L'Aquila)
- **Steering Committee:**
  - Michael W. Godfrey (University of Waterloo)
  - Marianne Huchard (Université Montpellier 2)
  - Tom Mens (University of Mons)
  - Oscar Nierstrasz (University of Bern)
  - Coen De Roever (Free University Brussels)
  - Vadim Zaytsev (Universiteit van Amsterdam)
- **Post-proceedings Editors:**
  - Davide Di Ruscio (University of L'Aquila)
  - Vadim Zaytsev (Universiteit van Amsterdam)

Contents of the volume

- ◇ *Profiling, Debugging, Testing for the Next Century*
  The methods of software engineering have improved significantly over half the century of its existence: we now have advanced languages that help to shape our thinking about both the problem and the solution; we have compilers and other grammarware that assists us in the most laborious tasks of translating programs written or drawn in software languages to machine code; we have various programming paradigms and schools of thought that allow us to represent our solutions in a concise yet efficient way. Programming environments have also grown, but the author of this contribution claims that the difference between an old Emacs and a modern Eclipse is less ground-breaking than the difference between, say, RPG and Swift. This report focuses on possible advanced techniques of profiling, testing, debugging and visualisation, prototyped in Pharo.
- ◇ *"What Programmers Do with Inheritance in Java", Replicated on Source Code*
  Inheritance is one of the key techniques of the object-oriented paradigm. As such, it has been thoroughly researched — this paper replicates one of such studies, published at ECOOP 2013. The original study found that the defined inheritance relationships are extensively used in the code, mostly for subtyping and reuse purposes, and that late-bound self-reference occurs frequently. This replication confirmed and complemented the results by running the same experiments on the source code as opposed to the byte code used in the original paper.
- ◇ *Explora: Infrastructure for Scaling Up Software Visualisation to Corpora*
  Software analyses often receives considerable profits from visualisation techniques, which help overcome the intangible nature of software by letting the user interact with concrete representations of various aspects of it. Some visualisations are also models which abstract from less important aspects and focus on the essentials. Yet, software corpora, quite often used in empirical software analysis — a software reverse engineering domain that requires useful and scalable visualisations — are not well-supported, as the authors of this report claim. This contribution presents Explora, an infrastructure that is specifically targeted at visualising software corpora (at this point, in Smalltalk and Java).
- ◇ *Detecting Refactorable Clones by Slicing Program Dependence Graphs*
  Code duplication is a well-known and well-researched code smell that can sometimes be linked to increased maintenance costs and creating difficulties in understanding a software system. One of the solutions addressing the issue involves detecting duplicated code, refactoring it into a separate function and replacing all the clones with appropriately parametrised calls to the new function. This report describes a confirmatory replication of a CodeSurfer-based tool for detecting such refactorable code clones based on the combination of program dependence graphs and backward program slicing.

◇ *A Critique on Code Critics*
Pharo (a Smalltalk IDE) has a recommendation facility called "code critics" which is used for signalling code controversial and suspicious implementation choices and for promoting good style language idioms. For instance, they report code smells and encourage correct use of object orientation. In this paper, code critics are re-evaluated based on a case study of the code of a big system with 51 packages (Moose). It was observed that some code critics tend to occur together and therefore should be grouped in issues of a higher level of abstraction; others occur only in presence of others up to the point of being overshadowed by them; some are even triggered by the same patterns and offer competing solutions. The approach is promising and can be applied broadly to any quality-driven program analysis.

◇ *User Interface Level Testing with TESTAR*
Software testing is exercised at many levels, and if done at the Graphical User Interface (GUI) level, allows to create very realistic test cases due to close emulation of user behaviour. This report offers an extension of the TESTAR tool by the same authors and their collaborators. TESTAR implements a model-driven approach to test automation which generates test cases based on a model which is automatically derived from the GUI through the accessibility API. The extension revolves around action ranking and using machine learning techniques in order to execute sensible and sophisticated sequences of GUI actions instead of blindly making a random selection of clicks and keystrokes that are visible and unblocked in each state.

◇ *Qualifying Chains of Transformation with Coverage-based Evaluation Criteria*
Development of complex and large model transformations can be optimised by composition of reusable smaller ones. Yet, composing them is a complex problem: typically smaller transformations are discovered and selected by developers from different and heterogeneous sources, and then chained by composition processes that are semi-automated at best. Without considering the semantic properties of a transformation, it is difficult for the user to make the right choice between two possible proposed chains. This report contains a proposal to classify such chains with respect to the coverage of the metamodels involved in the transformation.

◇ *Describing the Correlations between Metamodels and Transformations Aspects*
The paper considers two main concepts in MDE: a metamodel as a model that all regular models in the ecosystem conform to; and a model transformation as a description of changes in models. Surfacing, formalising and maintaining relations between the artefacts of these two kinds is a big subdomain of modelware. This report presents an approach to understand structural characteristics of metamodels by looking at how model transformations depend on metamodels of their source and target models.

- *A Three-level Formal Model for Software Architecture Evolution*
  Architecture Description Languages are the focus of this paper: a brief overview of them is given, and one (Dedal) is explain in full detail. It formally defines the architecture of a software system on three levels: the abstract specification, the concrete configuration and the instantiated architecture assembly. Software evolution is expressed in Dedal in static invariants that bind descriptions together and in evolution rules that trigger changes at each level. The running example in the paper is a home automation software that is used to guide the reader through explanations of evolution in Dedal. The authors consider integrating the methods presented here, in an IDE platform to help developers embrace software evolution further.
- *Representing Uncertainty in Bidirectional Transformations*
  Bidirectional transformations (BX) are a largely recognised quickly growing field of MDE. In BX, we usually deal with both ways of propagating information between two (or more) software artefacts — such changes are typically non-univocal in the sense that more than one correct solutions could be admitted and tolerated. However, most existing BX frameworks do not represent uncertainty explicitly. In this report, the authors insist that they should and show that it is quite possible in their framework, by re-explaining the non-deterministic nature of bidirectionality and presenting a case study with a family of cohesive models with uncertainty.
- *Towards a Taxonomy for Bidirectional Transformation*
  Software model consistency and synchronisation are often achieved by designing and implementing bidirectional transformations. This report demonstrates an early attempt to take a step towards a taxonomy of BX by illustrating a set of relevant properties of such mappings. The contribution is of an analytical nature: existing literature is analysed and characteristics of existing approaches are put into perspective.
- *Languages, Models and Megamodels*
  Software modelling is considered to be a somewhat difficult or even obscure subdomain of software engineering, with some of its topics such as megamodelling leaning towards the obscure part. However, it can be explained in relatively simple terms and given to software evolution researchers, can prove to be a useful advanced technique of expressing relations between software artefacts. During SATToSE 2014, we have held an interactive tutorial on this topic, which this report tries to summarise and complement.

Profiling, debugging, testing for the next century

Alexandre Bergel

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

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

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 for 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 curricula 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[2] and YourKit[3] profilers, the most popular and widely spread profilers in the Java World, output resource distributions along a tree of methods.

Sending messages is 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, largely discarding the notion of objects. Debugging still comes with its primitive operations based on stack manipulation; again ignoring objects. Naturally, some attempts have been made to improve the situation: Eclipse offers several navigation options; popular (and expensive) code profilers may rely on code instrumentation to find out more about the underlying objects; debuggers are beginning to interact with objects [1,2]. However, these attempts remain largely marginal.

Profiling. Great strides have been made by the software performance community to make profilers more accurate (*i.e.,* reducing the gap between the actual applica-

[1] http://pharo.org
[2] http://www.ej-technologies.com/products/jprofiler/overview.html
[3] http://www.yourkit.com

tion execution and the profiler report). Advanced techniques have been proposed such as variable sampling time [3] and proxies for time execution [4,5]. However, much less attention has been paid to the visual output of a profile. Consider JProfiler and YourKit, two popular code profilers for the Java programming language: profile crawling is largely supported by text searches. We address this limitation with *Inti*.

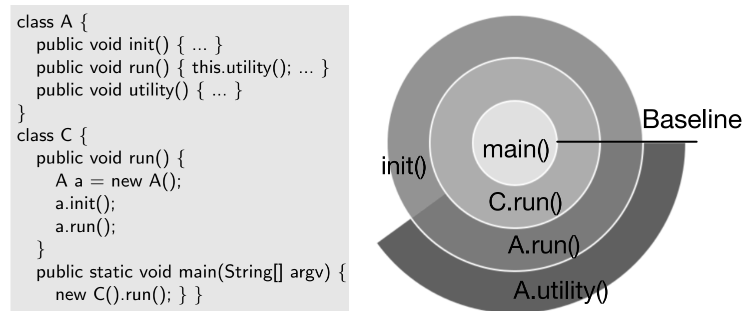


Fig. 1: Sunburst-like visualization

Inti is a *sunburst*-like visualization dedicated to visualizing CPU time distribution. Consider the code and arc-based visualization given in Figure 1. The code indicates a distribution of the computation along five different methods. Each method frame is presented as an arc. The baseline represents the starting time of the profile. The angle of each arc represents the time distribution taken by the method. In this example, `C.main` and `C.run` have an angle of 360 degrees, meaning that these two methods consume 100% of the CPU time. Methods `A.init` consumes 60% and `A.run` 40% (these are illustrative numbers). The distance between an arc and the center of the visualization indicates the depth of the frame in the method call stack. A nested method call is represented as a stacked arc.

Inti exploits the sunburst representation in which colors are allocated to particular portion of the sunburst. For example, colors are picked to designate particular classes, methods or packages in the computation: the red color indicate classes belonging to a particular package (*e.g.,* Figure 2).

The benefits of Inti are numerous. Inti is very compact. Considering the number of physical spaces taken by the visualization, Inti largely outperforms the classical tree representation: Inti shows a larger part of the control flow and CPU time distribution in much less space. Details about each stack frame are accessible via tooltip. Hovering the mouse cursor over an arc triggers a popup window that indicates the CPU time consumption and the method's source code.

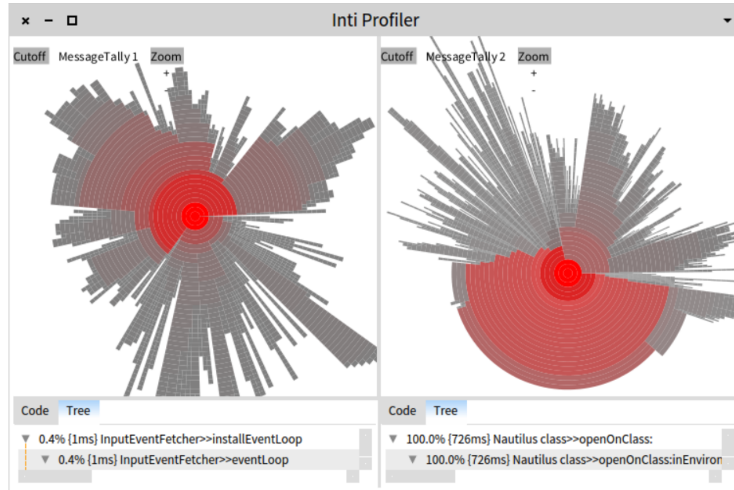


Fig. 2: Sunburst-like profile

Delta 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. JProfiler and YourKit both fail at providing adequate metrics and visual representations, conveying a false sense of the root cause of the performance variation.

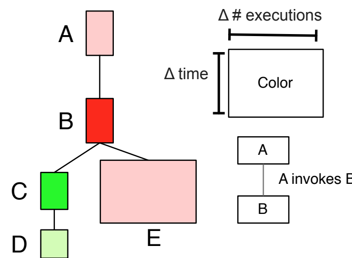


Fig. 3: Performance evolution blueprint

We have proposed *performance evolution blueprint*, a visual tool to precisely compare multiple software executions [6]. The performance evolution blueprint is summarized in Figure 3. 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). The 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; if the difference is negative (*i.e.,*

the method is faster), then the method 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: Figure 3 indicates that method B is likely to be responsible for the slowdown since the method is slower and has been modified. We 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 efforts.

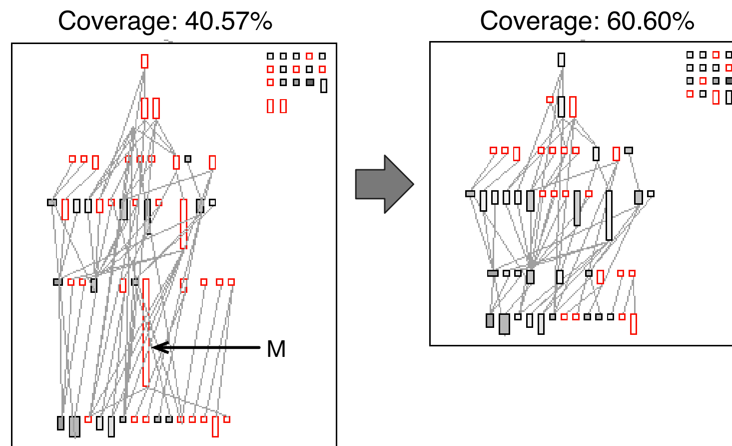


Fig. 4: Test blueprint

By relating execution and complexity metrics, we have identified essential patterns to characterize the test coverage of a group of methods [7]. 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 4 is an example of a test blueprint, obtained as the result of the test execution. Consider Method `M`: the definition of this method is relatively complex, which is indicated by the height of the box representing it. `M` is shared in red, meaning it has not been covered by the unit test execution. Covering this method and reducing its complexity is therefore a natural action to consider.

Two versions of the same class are represented. Inner small boxes represent methods. The size of a method indicates its cyclomatic complexity. The taller a method is, the more complex it is. Edges are invocations between methods, statically determined. Red color indicates uncovered methods. The figure shows 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. 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 [8].

We have proposed *object-centric debugging* as an alternative approach to interacting with a running software system [9]. 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 5 illustrates two visualizations of a software system's dependencies. Each class is represented as a circle. On the left side, gray edges are inheritance (the top superclass is at the center) and blue lines are dependencies between classes. Each color indicates a component. On the right side, edges are dependencies between classes, whereas class size and color indicate the size of the class. Roassal

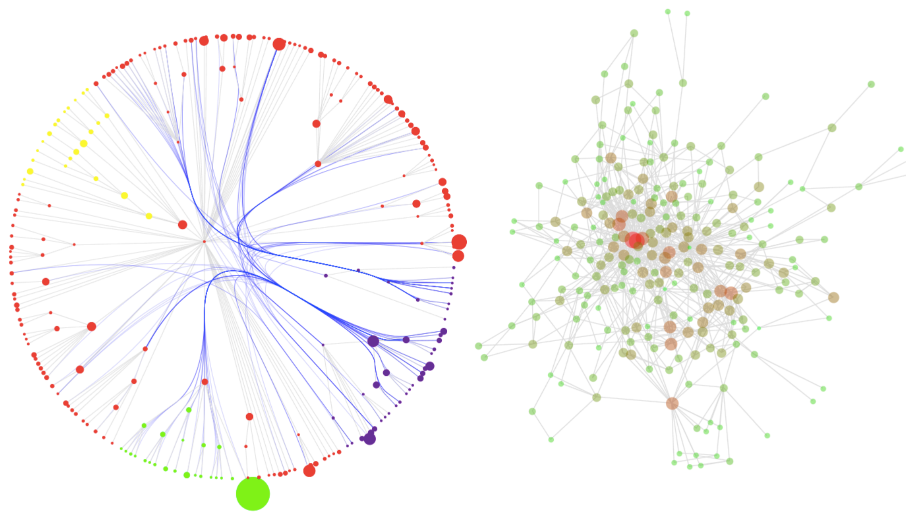


Fig. 5: Visualization of a software system's dependencies

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, 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 reduce the cognitive gap between what the tools present to the programmers, and what the programers expect to see from the tools.

References

1. A. Lienhard, T. Gîrba, O. Nierstrasz, Practical object-oriented back-in-time debugging, in: Proceedings of the 22nd European Conference on Object-Oriented Programming (ECOOP'08), Vol. 5142 of LNCS, Springer, 2008, pp. 592–615, ECOOP distinguished paper award. `doi:10.1007/978-3-540-70592-5_25`.
   URL `http://scg.unibe.ch/archive/papers/Lien08bBackInTimeDebugging.pdf`
2. G. Pothier, E. Tanter, J. Piquer, Scalable omniscient debugging, Proceedings of the 22nd Annual SCM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'07) 42 (10) (2007) 535–552. `doi:10.1145/1297105.1297067`.
3. T. Mytkowicz, A. Diwan, M. Hauswirth, P. F. Sweeney, Evaluating the accuracy of java profilers, in: Proceedings of the 31st conference on Programming language design and implementation, PLDI '10, ACM, New York, NY, USA, 2010, pp. 187–197. `doi:10.1145/1806596.1806618`.
   URL `http://doi.acm.org/10.1145/1806596.1806618`

4. A. Camesi, J. Hulaas, W. Binder, Continuous bytecode instruction counting for cpu consumption estimation, in: Proceedings of the 3rd international conference on the Quantitative Evaluation of Systems, IEEE Computer Society, Washington, DC, USA, 2006, pp. 19–30. doi:10.1109/QEST.2006.12.
URL http://portal.acm.org/citation.cfm?id=1173695.1173954
5. A. Bergel, Counting messages as a proxy for average execution time in pharo, in: Proceedings of the 25th European Conference on Object-Oriented Programming (ECOOP'11), LNCS, Springer-Verlag, 2011, pp. 533–557.
URL http://bergel.eu/download/papers/Berg11c-compteur.pdf
6. 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.
7. A. Bergel, V. Peña, Increasing test coverage with hapao, Science of Computer Programming 79 (1) (2012) 86–100. doi:10.1016/j.scico.2012.04.006.
8. J. Sillito, G. C. Murphy, K. 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, ACM, New York, NY, USA, 2006, pp. 23–34. doi:10.1145/1181775.1181779.
URL http://people.cs.ubc.ca/~murphy/papers/other/asking-answering-fse06.pdf
9. 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

"What Programmers Do with Inheritance in Java" - Replicated on Source Code

Ç. Aytekin

cigdemaytekin.872@gmail.com, University of Amsterdam

Abstract. Inheritance is an important mechanism in object-oriented languages and it has been subject of quite some research. Tempero, Yang and Noble made a research [Tempero13] about the usage of the inheritance in Java open source systems and found that the defined inheritance relationships are also used quite considerably in the code, mostly for subtyping and reuse. They also found that late-bound self-reference occurs frequently. They analysed the byte code of the projects. We replicated their study to verify the results by carrying out the same study on the source code. For most of the metrics introduced in their inheritance model we found similar results. We found some suspected false positives in the original study for late-bound self-reference and (external) reuse, but there are not many of them. Except for these cases, our study verifies the correctness of the original study results.

1 Introduction

We externally replicated a study done by Tempero et al. [Tempero13] about inheritance usage. Inheritance is an important mechanism in object-oriented programming. The majority of the studies about inheritance concentrated on the declaration of the inheritance relationships. The original study brought a new perspective on inheritance research. They investigate the *usage* of inheritance in their study, in their own words: "having made the decision to use inheritance at the design level, what benefits follow from the use of inheritance?"

The authors defined an inheritance usage model and analysed the byte code of 93 open source Java projects from Qualitas Corpus, which is a curated collection of open source Java projects [Tempero10]. Their results show that the defined inheritance relationships in projects are frequently used, especially for what they call subtyping and reuse.

Our purpose is to replicate the original study with one major difference: we analyse the source code and not the byte code. We have chosen for replication because of two reasons. Firstly, replication plays a very important role in verification of the results of empirical studies in general. Also in the field of software engineering empirical research this is the case. Brooks et al. explain this in [Brooks08]. Secondly, despite the importance of replication, there are very few replication studies so far, as also shown by Sjoberg et al. in [Sjoberg05]. Replication studies are answers to this need, and so is ours.

Our analysis results are similar to those of original study. But they are not the same. For down-call, we report that 27 % late-bound self-reference (original study reports 34 %) For subtype, we report that at least 61 % of inheritance relations show this usage, whereas original study reports 69 %. Original study reports 22 % of external reuse and 2 % of internal reuse, whereas we report 4 % and 20 % respectively. Our results also show that reuse and subtype explain most of the usages of inheritance and other uses are not significant, just like the original study

We see the following reasons for the differences between the results: the differences between the set-up of two studies, and our limitation about analysing external methods. Our limitation about external method analysis is explained in subsection 5.3. However, we also suspect some false positives in the original study for down-call and external reuse. The reason why we think that there are some false positives is explained in the discussion section (section 8)

This article starts with introducing some of the empirical studies about inheritance. Section 3 contains the definitions of the inheritance model used . In section 4 the original study is explained. Section 5 presents the replication study. The Rascal implementation is explained in section 6. The results of the replication study is presented in section 7. The results are discussed in section 8, followed by the list of threats to the validity of our work (section 9). Finally we conclude with section 10.

2 Related Work

A thorough discussion about the notion of inheritance is given by Taivalsaari in [Taivalsaari96]. Three main usages defined in the original study refer to this article: subtype usage, reuse and down-call (late-bound self-reference). We will define these concepts in detail, but here is a brief explanation in advance. Subtype usage occurs when a child type supplied when parent type is expected, reuse occurs when a child type uses a field or a method defined in its parent and down-call occurs when a method call in a class is directed to a type which is down in the inheritance hierarchy instead of a method in the class itself.

There is a group of empirical studies about inheritance which analyse the code of projects. Tempero and Noble, this time together with H. Melton, carried out the study [Tempero08] using an earlier version of Qualitas Corpus. The study concentrated mainly on how types are defined with respect to inheritance in Java open source projects. In another study, Nasseri, Counsell and Shepperd investigated the evolution of inheritance in Java open source systems (OSS) [Nasseri08]. Lämmel et al. analysed a corpus of .NET projects for the usage of .NET API in the source code [Lammel11]. This last study is also about inheritance usage but from the perspective of API usage.

Another group of empirical studies worked with programmers to observe the effect of inheritance on software quality. An early study of Mancl and Havanas from 1990 [Mancl90], focuses on the effects of using C++ programming language on software maintenance. Similarly, Daly, Brooks, Miller, Roper and Wood also

made an experiment [Daly96] with programmers about the inheritance and investigated if the programs written with inheritance were more easily maintained than the programs written without inheritance. Cartwright replicated the study done by Daly et al., but ended up with opposite results [Cartwright98].

3 Definitions

The authors of the original study proposed a model for inheritance usage. The most important usages of inheritance in their model are subtype, reuse and down-call. In addition to these, they also describe other uses of inheritance, which also occur, but much less frequent than subtype, reuse and down-call.

If a pair of classes has inheritance relationship, it is modelled as an ordered pair of descendant and ascendant. If there is also a usage between the descendant and the ascendant, then this pair is given the attribute which qualifies that certain usage. How often a usage occurs is not taken into consideration here. For example, if a descendant reuses a piece of code from the ascendant, how many times this reuse occurs in the code does not matter.

Here are the definitions of system type, external method, user defined attribute, CC, CI and II attributes, the explicit attribute and indirect reuse. These definitions are important for understanding the scope and the metrics of the original study:

- **System Type** A type (Java class or interface) is a system type if it is defined in the system under investigation. The rest of the types, which are used in the system, but are not defined in the systems are called non-system types. Non-system types are typically defined in the external libraries on which the system under investigation depends on.
- **External method** Similar to non-system types, the methods which are defined outside of the system under investigation are called external methods, or non-system methods.
- **User Defined Attribute:** The descendant ascendant pair in an inheritance relationship has user defined attribute if both of descendant and ascendant are system types. A system type is created for the system under investigation
- **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), CI (Class Interface) and II (Interface Interface).
- **Explicit Attribute** The inheritance relationship is described directly in the code. Inheritance relation between a child and its direct parent is explicit, whereas a child and its grand parent is not explicit, but implicit.
- **Indirect Reuse** If the inheritance use occurs between the types in a pair which has not explicit attribute, this usage gets the *indirect* attribute. Let us assume that class `GC` extends class `C` and class `C` extends class `P`. If an external reuse occurs between GC and P, all the pairs between GC and P (in this case <GC,C> and <C,P>), are counted as having external reuse attribute (*indirect external reuse*). This is done for the external reuse and subtype

attributes only. For other inheritance usages like down-call or internal reuse, this is not done.

The authors count only explicit user defined pairs in their research.

The inheritance usages we most frequently see in the open source Java projects are subtype, external reuse and internal reuse, which are defined as follows:

- **Subtype:** Subtype inheritance usage happens when a child type is supplied where the parent type is expected. Subtype occurrence can be seen during assignment, casting, parameter passing and returning a parameter in a method declaration in Java. Moreover the enhanced for loop (for each construct) and ternary operator can also contain subtype usage.
- **Internal Reuse:** Reuse occurs if an object of child type accesses a field or calls a method of a parent type. If the reuse happens in the class definition of the child class, this is called internal reuse.
- **External Reuse:** If the reuse happens outside of the class definition of the child class, this is called external reuse.

Down-call is one of the most important concepts of the original study and one research question is solely about down-call. Here is an illustrative example of down-call:

```
class P {
  void p() {
    q();
  }
  void q() {}
}
```

```
class C extends P {
  void q() {}
}
```

In the example, the `p()` method of class `P`, calls method `q()` which is also overridden by its child type. If method `p()` is called on an object of child type, the `q()` method of child type is called instead of the one of parent type. The original study counts the potential down-calls only, in other words, they do not search for a call of method `p()` on an object of child type. In the replication study we did the same, but we find that this approach is open to discussion.

In addition to the most frequent usages, other uses of inheritance are also defined in the inheritance model. Because these usages do not occur frequently, we will only give brief definitions of these concepts. *Category* usage occurs if a parent type has more than one child and at least one of the children have subtype relation with the parent. The siblings which has no other usage with the parent receive category attribute. If an ascendant has only Java constant fields in it, the descendant-ascendant pairs get the *constant* attribute. *Framework* attribute, on the other hand, is given to the descendant-ascendant pairs where ascendant is a child of a non-system type.

The *generic* attribute qualifies a frequently used pattern in raw collections between an ascendant and a descendant. *Marker* usage qualifies the pairs for which ascendants are empty interfaces and descendants implement them for the reason of conceptual classification. Finally, the *super* attribute are given to pairs

in which the descendant explicitly issues a `super()` call to the constructor of the ascending type.

4 The Original Study

4.1 Research Questions

After introducing an inheritance usage model, the original study concentrates on four research questions:

1. To what extent is late-bound self-reference (down-call) relied on in the designs of Java Systems? (the ratio of pairs for which down-call is seen to the total number of class-class inheritance pairs.)
2. To what extent is inheritance used in Java in order to express a subtype relationship that is necessary to design? For class-class pairs, this is the ratio of pairs with subtype usage to the total number of *used* pairs. The *used* pairs are the inheritance pairs for which subtype or reuse is seen. The subtype usage between class-interface and interface-interface pairs are also measured.
3. To what extent can inheritance be replaced by composition? The inheritance relationships which involve internal or external reuse, but which do not involve subtype use, are candidates for such a replacement.
4. What other inheritance idioms are in common use in Java systems? (This is answered by considering the results of various other metrics about other inheritance usages like category, constant, framework, etc.)

4.2 Implementation

The authors developed a Java byte code analysis tool which is based on SOOT framework [ValleeRai99]. With this tool they analysed the byte code distributions of 93 Java projects from the 20101126 release of the Qualitas Corpus [Tempero10]. The study is very well documented in the study web site [Tempero08Web] The article and the study web site have been immensely useful for us when conducting this replication study. When we needed additional information, we e-mailed the authors and we got detailed answers from the first author (E. Tempero). These answers improved our understanding and enabled us to deliver a replication study with better quality.

The original study has some limitations about inheritance model which we also use. The analysis is limited to classes and interfaces, exceptions, enums and annotations are excluded. Moreover only the types which are declared in the project are analysed, and not the third party libraries. Unlike our study, the original study has also some limitations which originate from byte code analysis. In a few cases, for example, source code may not map correctly to byte code.

4.3 Results

For the first question, they conclude that down-call plays a significant role - around a third (median 34 %) of all class-class pairs involve down calls. For the second question, they saw that least two thirds of all inheritance pairs are used as subtypes in the program. About replacing inheritance with composition, the authors found that 22 % or more pairs use external re-use (without subtyping) and 2 % or more use internal re-use (without subtyping or external reuse) which signals opportunities to replace inheritance with composition. For the last question, they report some other uses of Java inheritance (constant, generic, etc.) however the results show that big majority of inheritance pairs (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.

5 Replication Study

5.1 Research Questions

Our research questions directly refer to the four research questions of the original study. For each question we would like to know how our results differ from those of the original study.

5.2 Differences in the Study Set-up

Our study has some differences from the original study in the study set-up. When comparing the results, it is necessary to consider these differences:

- **Source code versus byte code:** The biggest difference between the original and replication studies is about the input to analysis work. The authors analyse the byte code, while we analyse the source code of Java projects.
  Before we started the replication study, we have decided to choose for the source code analysis. There were two reasons for that: firstly, we did not intend to perform an exact replication, we wanted to answer the same research questions from a different perspective, namely by analysing the source code. And secondly, we wanted to use an existing robust tool (or meta-programming language) for analysis. Rascal is proven to be a robust tool for Java source code analysis, but it does not support Java byte code analysis at the moment.
- **Differences between the content of the byte code and the source code:** When we analysed the source code and compared the contents of the source code and byte code distributions, we saw that the set of types which are contained in both distributions differ from each other quite considerably, even for the same versions of the projects.
- **Qualitas Corpus vs. Qualitas.class Corpus:** The authors used the Qualitas Corpus [Tempero10], we also use the Corpus, but the compiled version of it, namely Qualitas.class Corpus [Terra13]. In the original study 93 open

source Java projects are analysed. We could not analyse 3 projects because of errors. From the 90 projects we did analyse, 65 have the same version as the original study and 25 have different versions. The meta-language we used to analyse the source code, Rascal, will analyse the source code correctly when the source code compiles. Therefore it was important for us that the source code compiled correctly. We had two alternatives: we could either use the original corpus and invest time on resolving dependencies of the source code to external libraries and fixing the compilation problems ourselves, or we could use the compiled Corpus (Qualitas.class Corpus), for which this work was already done. We have chosen the second option because of the time limitations, and this meant that we had to analyse different versions of 25 projects.

5.3 Limitations of the Replication Study

The differences between the content of the code analysed is a limitation of our study, as explained in the previous subsection in detail. This difference makes comparison of the results less straightforward.

Another limitation which has impact on our results is about the analysis of non-system methods. We have limited information about external methods. We only analyse the system types, and the external methods are defined in non-system types, i.e. outside of our analysis boundaries. For external reuse, this limitation results in fewer number of external reuse cases for indirect external access (we only mark the child and the immediate parent with external reuse attribute, whereas the original study marks the whole chain between the child and the ascendant which declares the method). For subtype, this limitation affects our analysis during parameter passing. When an external method is called, the parameter types are not totally available for us. We use a very limited heuristic to analyse these calls and it is highly likely that we miss some subtype cases.

There are some more limitations to our study which we think will not have major impact on the results. To name a few: Internal reuse attribute is not analysed for class-interface and interface-interface pairs, for interface-interface pairs category attribute is not analysed, method parameters that are given as ternary operators are not analysed for subtype, etc.

6 Implementation of the Replication Study

We have written a Java source code analysis program in Rascal. Rascal is a meta-programming language which has various features to make (among others) analysis of Java source code easy. Rascal is fully integrated in Eclipse IDE.

With the following simple Rascal code example, we would like to give an idea about how Java source code analysis is done by Rascal:

```
1 public void run() {
2   set[Declaration] pASTs = createAstsFromEclipseProject(|project://cobertura
        -1.9.4.1|, true);
3   pM3 = createM3FromEclipseProject(|project://cobertura-1.9.4.1|);
```

```
4     for (anAST <- pASTs) {
5       visit (anAST) {
6         case m1:\methodCall(_, receiver:_, _, _) : {
7           set[loc] defClassSet = {aClass | <aClass, aMethod> <- pM3@containment
                , isClass(aClass), aMethod == m1@decl};
8           if (!isEmpty(defClassSet)) {
9             loc definingClass = getOneFrom(defClassSet) ;
10            println("The method <m1@decl> is defined in: <definingClass>" );
11            println("Type of receiver is: <receiver@typ>");
12          }
13        }
14      }
15    }
16  }
```

Listing 1.1. Sample Rascal code which analyses a method call

`createAstsFromEclipseProject()` in line 2 is a Rascal method returns all ASTs (Abstract Syntax Trees) for a given Java project. Rascal also creates the M3 model for a given project with method `createM3FromEclipseProject()` as we see in line 3. M3 model contains information about the project from various aspects. This information can be accessed via annotations in Rascal. Some examples of the annotations are: `@extends` annotation (which lists the parent child pairs for classes and interfaces), `@implements` annotation (similar to extends, but for class interface pairs), `@declarations` annotation (which lists the location where different items in the project are declared). In our example, we access to `containment` annotation of project M3 in line 7 to retrieve the class in which the method was declared.

The information in M3 are stored as binary relations (ordered pairs), and Rascal also enables access to binary relations by comprehensions, as we again see in line 7. Once the ASTs are built, it is also possible to visit each node of an AST via the Rascal construct `visit` - line 5 in the example above. The `case` statements (line 6) in the `visit` construct are used for selecting the AST node we are interested in, in this case a `methodCall()`. Once we selected the node we want, it is also possible to retrieve further information about the node itself, like the name of the method that is called (in our example `m1@decl`), if it has a receiver (an object on which the method call was issued - in our case `receiver`) and the type of the receiver (`receiver@typ`).

7 Results

The results of our analysis are, in many cases, similar to the results of the original study, but they are not the same. For most of the inheritance usage we report fewer cases:

For down-call: we observed 27 % (median) of the class-class relations involving potential down-calls whereas the original study reports 34 % median.

For subtype: for class-class pairs, we observed 76 % of subtype usage, just like the original study. Subtype usage can also be seen in class-interface and interface-interface pairs, for class-interface pairs we report a median of 61 % (original study 69 %) and for interface-interface pairs our median is 75 %, while the original median is % 72.

For reuse, we also see that there is opportunity for replacing inheritance with composition. However, we report significantly fewer cases of external reuse, and also significantly more cases of internal reuse. For class-class pairs, our external reuse median is 4 % (original study : 22 %) and our internal reuse median is 20 % (original study 2 %).

For other inheritance cases, we also found some usage, and we also observed that these usages are not significant when compared to subtype and reuse.

Despite not being part of a research question, the perCCUsed (percentage of class-class pairs which have subtype or reuse attribute) is an important metric. For this metric, we have a median of 88 %, while the original study reports 99 %.

The results of the both studies are summarized in table 1.

Inheritance Usage	Replication median (%)	Original median (%).
Down-call	27	34
Subtype - class class pairs	76	76
External reuse - no subtype	4	22
Internal reuse only	20	2
Subtype - class interface pairs	61	69
Subtype - interface interface pairs	75	72

Table 1. Comparative Summary of Results

8 Discussion

The fact that the source code distributions are in many cases very different from the byte code distributions, has a major impact on our results. Moreover, our limitation about the analysis of the external method calls is highly likely to deliver fewer cases in subtype and external reuse. We also did our best to be able to interpret the inheritance model in a sound way, however, there may still be misunderstandings from our side about definitions of certain concepts.

Keeping these limitations in mind, we tried to find out some projects which we could manually investigate to search for reasons for our differences. One small project which had similar byte and source code contents was cobertura (v. 1.9.4.1). For this project, the original study reported five down-call cases, while we could not observe any. Here we include one case for which we suspect a false positive from the byte code analysis. Original study reported a down-call usage between classes GTToken and Token.

The GTToken class from project cobertura has the following source code:

```
1  public static class GTToken extends Token
2  {
3    int realKind = JavaParser15Constants.GT;
4  }
```

An excerpt from the byte code of GTToken is as follows:

```
1     0: aload_0
2     1: invokespecial #10 // Method
3        // net/sourceforge/cobertura/javancss/parser/java15/Token."<init>":()V
4     4: aload_0
5     5: bipush        126
6     7: putfield      #12   // Field realKind:I
7    10: return
```

The definition of down-call makes it necessary that child class GTToken overrides at least one method. In this case, however, we do not see any methods defined by GTToken. GTToken defines only one field. When we look at the byte code, however, we see the command `invokespecial`. We wondered if this was causing the down-call report in the original study. We have e-mailed the authors, and they also could not bring an explanation about this particular case.

From our manual investigation, we also found one more down-call case which is different from the case explained above, for which we also suspect a false positive. For this case, we also mailed the authors and we suspect an interpretation difference for down-call definition between two studies.

We also did a manual investigation for other cases of inheritance usage and we also suspect some false positives for the external reuse, however we did not have enough time to discuss this via e-mail with the authors.

To summarize, we suspect some false positives for down-call and external reuse cases of the original study. We also think that this may be introduced due to the analysis of byte code, in some cases byte code may be misleading. However, because of the limitations of our study, we can not give an exact percentage of false positives, but we do not expect a high percentage of false positives.

9 Threats to Validity

The major threat to validity to our replication is the input we are using. The fact that our input is very different from the original study poses a threat to validity when comparing the results of two studies.

Furthermore, we have a limitation when analysing the external method calls. We know that this causes fewer cases to be counted for subtype and external reuse, but we can not give an exact percentage. This also constitutes a threat for the validity during comparison of subtype and external reuse percentages.

We did our best to understand the inheritance model proposed by the authors. However, there can still be some interpretation differences about the inheritance usage definitions, and this may also pose a threat to our analysis results.

Our minor limitations, which were briefly discussed in section 5.3, will also affect our results and should also be mentioned as, however minor, possible threats to validity.

10 Conclusion

Our conclusions for the research questions are similar to the original study, but they are not the same.

For the first question (about down-call), original study reports about one third of the inheritance relations involving such a case, while we found about one fourth.

For the second research question, we found about 60 % of all inheritance cases involve subtype relationship. The authors also report a higher percentage (66 %) about subtype usage.

About research question three (replacement of inheritance with composition), the pairs without subtype but with reuse attribute are taken into account. Authors found a significant percentage of reuse (median 22 % for external and 2 % for internal reuse). We also report a similar percentage, but the division between internal and external uses is very different in our case (median of 4 % for external reuse and 20 % for internal reuse).

For the last research question, which is about the other uses of inheritance in Java, the authors found out that these occur in many systems, but their use is not generally significant. Although our percentages are not exactly the same with the original study for various other uses of inheritance, our results also agree with this conclusion.

When we investigate the possible reasons for the differences, we see that especially the differences in our study set up play a role here. In addition to this, for the subtype and external reuse, it is highly probable that we report fewer cases because of our limitation of external method analysis. We also conclude that the analysis of byte code can result in false positives in some particular occasions for down-call and external reuse..

As future work for the original study, one can discuss the proposed inheritance model and the metrics and consider some alternatives for detecting and counting various inheritance usages. Especially, how down-call usage is detected and the role of indirect usage in subtype and external reuse, according to us, present some opportunities for further discussion.

Acknowledgments

Author would like to Ewan Tempero for his detailed answers to her questions and to Bas Brekelmans for his help in validating results of this study.

References

[Brooks08] Brooks, A. and Roper, M. and Wood, M. and Daly, J. and Miller, J. Section: "Replication's role in software engineering." from the book : "Guide to advanced empirical software engineering." Springer London, 2008. pp: 365-379.

[Cartwright98] Cartwright, Michelle. "An empirical view of inheritance." Information and Software Technology 40.14 (1998): pp: 795-799.

[Daly96] Daly, J., Brooks, A., Miller, J., Roper, M., & Wood, M. (1996). "Evaluating inheritance depth on the maintainability of object-oriented software." Empirical Software Engineering, 1(2), pp: 109-132.

[Lammel11] Lammel, R., Linke, R., Pek, E., & Varanovich, A. (2011, October). "A framework profile of. net." In 18th Working Conference on Reverse Engineering (WCRE), 2011 (pp. 141-150). IEEE.

[Mancl90] Mancl, Dennis, and William Havanas. "A study of the impact of C++ on software maintenance." In Proceedings of International Conference on Software Maintenance (ICSM), 1990, IEEE, pp: 63 - 69

[Nasseri08] Nasseri, Emal, Steve Counsell, and M. Shepperd. "An empirical study of evolution of inheritance in Java OSS." In 19th Australian Conference on Software Engineering (ASWEC), 2008, IEEE, pp: 269 - 278

[Sjoberg05] Sjberg, D. I., Hannay, J. E., Hansen, O., Kampenes, V. B., Karahasanovic, A., Liborg, N. K., & Rekdal, A. C. (2005). "A survey of controlled experiments in software engineering." IEEE Transactions on Software Engineering, 31(9), pp: 733-753.

[Taivalsaari96] Taivalsaari, Antero. "On the notion of inheritance." ACM Computing Surveys (CSUR) 28.3 (1996): 438-479.

[Tempero08] Tempero, Ewan, James Noble, and Hayden Melton. "How do Java programs use inheritance? An empirical study of inheritance in Java software." In European Conference On Object Oriented Programming (ECOOP) 2008. Springer Berlin Heidelberg, pp: 667-691.

[Tempero08Web] Ewan D. Tempero, Hong Yul Yang, and James Noble. Inheritance Use Data, 2008. Last accessed on 1 September 2014. URL: https://www.cs.auckland.ac.nz/~ ewan/qualitas/studies/inheritance/

[Tempero10] Tempero, E., Anslow, C., Dietrich, J., Han, T., Li, J., Lumpe, M. & Noble, J. (2010, November). "The Qualitas Corpus: A curated collection of Java code for empirical studies." In 17th Asia Pacific Software Engineering Conference (APSEC), 2010, pp. 336-345, IEEE.

[Tempero13] Tempero, Ewan, Hong Yul Yang, and James Noble. "What programmers do with inheritance in Java." European Conference On Object Oriented Programming (ECOOP), 2013 Springer Berlin Heidelberg, 2013. pp: 577-601.

[Terra13] Terra, R., Miranda, L. F., Valente, M. T., & Bigonha, R. S. (2013). "Qualitas. class Corpus: A compiled version of the Qualitas Corpus." ACM SIGSOFT Software Engineering Notes, 38(5), pp: 1-4.

[ValleeRai99] Valle-Rai, R., Co, P., Gagnon, E., Hendren, L., Lam, P., & Sundaresan, V. (1999, November). "Soot-a Java bytecode optimization framework." In Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research (CASCON), 1999, (p. 13). IBM Press.

Explora: Infrastructure for Scaling Up Software Visualisation to Corpora

Leonel Merino, Mircea Lungu, Oscar Nierstrasz

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

Abstract. Visualisation provides good support for software analysis. It copes with the intangible nature of software by providing concrete representations of it. By reducing the complexity of software, visualisations are especially useful when dealing with large amounts of code. One domain that usually deals with large amounts of source code data is empirical analysis. Although there are many tools for analysis and visualisation, they do not cope well software corpora. In this paper we present Explora, an infrastructure that is specifically targeted at visualising corpora. We report on early results when conducting a sample analysis on Smalltalk and Java corpora.

1 Introduction

A software corpus is a curated catalogue of software systems intended to be used for empirical studies of code artefacts. The advantage of doing research on corpora is that it encourages repeatable analyses. Corpus analysis is especially used in the context of empirical software engineering, where results should be repeatable [9, 15, 16]. Visualisation is especially useful for dealing with large amounts of source code, since it provides software a concrete representation making complex data easier to understand. However, most visualisation tools are not designed for corpora.

Imagine Edgar, an empirical software engineering researcher, who wants to assess the prevalence of reuse of software in different languages. To begin, he chooses to study reuse through inheritance and invocation in one of the oldest object-oriented languages, Smalltalk and one of the most popular, Java. To gain an initial insight into the data, he must carry out explorative data analysis, including visualisation.

To set up the environment for the analysis, he needs to overcome several problems: *1*) visualising one system at a time (as most visualisation systems allow) prevents patterns from being recognized at the corpus level; *2*) the lack of means for real-time data manipulation (such as sorting, filtering, searching, inspecting) discourages experimentation; and *3*) two technical issues, memory consumption and performance, complicate fetching and manipulating corpus data.

In this paper we introduce our approach, Explora, which copes with these issues.

2 Analysis Example

Edgar wants to target Smalltalk and Java corpora. He has a fair experience implementing and maintaining systems in both languages. He realises that in his experience Smalltalk systems have deeper hierarchies than Java ones, so he wants to explore whether increased specialisation in Smalltalk systems correlates with less reuse of their classes. Thus, he wants to answer the following research question:

RQ: "*How different is reuse by inheritance and invocation in Smalltalk and Java systems?*"

He believes that metrics are a suitable way for tackling this research question. He chooses four metrics from a catalogue of metrics proposed by Lanza and Marinescu to characterise two types of reuse: reuse via inheritance and reuse via invocation [12].

1) Average Hierarchy Height (AHH): the average depth of the inheritance trees of a system is one of the two metrics that characterise inheritance.
2) Depth of Inheritance Tree (DIT): the maximum length of the path of each class to the root class in a hierarchy; a measure of the system can be obtained by aggregating this metric for each of the classes.
3) Fan-In: this quantifies the dependent classes (access and invocation relationships) of a class.
4) Fan-Out: the outgoing coupling, which characterises communication.

Edgar decides to start his analysis by visualising these metrics on all the systems in both corpora. He decides to use Explora for his analysis. He downloads the models of the systems in the two corpora following the installation instructions of Pangea[1] locally and places them in a dedicated folder called the *model workspace*. To query corpora, Explora uses a so-called interactive Playground implemented on top of the Moldable Inspector infrastructure [3] of Moose [8].

Step 1: Computing the Metrics Figure 1 shows a screenshot of the Playground in which:

1) As the user modifies the query in the *Mapper* pane, the right pane is updated. The Mapper pane shows a query written in plain Smalltalk for collecting AHH (line 3), and the aggregated maximum value of DIT (line 7), Fan-In (line 9) and Fan-Out (line 11) from every system in the two studied corpora. The query defines an inline array of associations, where each association is linked to a different metric. The query uses the bound variable *eachModel* to refer to the FAMIX meta-model of each system. The model enables code such as *"eachModel allModelClasses"* and *"c superclassHierarchy"*.

[1] `http://scg.unibe.ch/research/pangea`

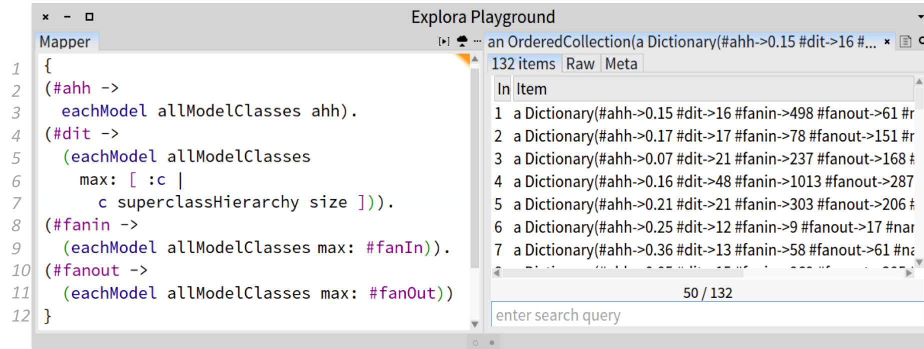


Fig. 1. Collecting metrics from Smalltalk and Java corpora.

The user can evaluate metrics defined in the system model, as is the case of AHH, Fan-In and Fan-Out, or compute custom ones such as the one specified for DIT. The query is sent to all the models of the systems in the model workspace

2) The right pane shows the result of the query, which is always a collection of objects retrieved from the queried system models by executing the query in the Mapper pane. The objects in this collection can be further manipulated (*i.e.* sorted, filtered, further queried, visualised).

Each pane of the Playground is linked to an object. By using the bound variable *self* the user can manipulate the object. The Playground supports navigation using the Miller column technique[2] on which the evaluation of a script in a pane adds a new pane to the right that is linked with the returned object. The Mapper is a special pane (since it is the first) that uses a different bound variable (eachModel) for referencing specifically system models.

Step 2: Generating an Initial Visualisation A special type of manipulation supported in right pane of the Playground uses the result object (*i.e.* the collection of dictionaries) as input for a visualisation. To use this feature, Edgar switches from the list view of the result to the Raw tab, which allows him to write a visualisation script. For instance, Figure 2 shows in the left pane an implementation of a lightweight visualisation using the Mondrian DSL. The result object is referred to in this script as “self” (see lines 4, 5, 11 and 12).

The right pane shows the generated visualisation. Each box represents a system, Smalltalk ones being grouped at the bottom while Java systems are at the top. The width of each box is mapped to AHH and the height to DIT. Edgar mapped the metrics in this way so that boxes with a larger area will indicate systems with many deep hierarchies. Furthermore, the darker the green

[2] http://en.wikipedia.org/wiki/Miller_columns

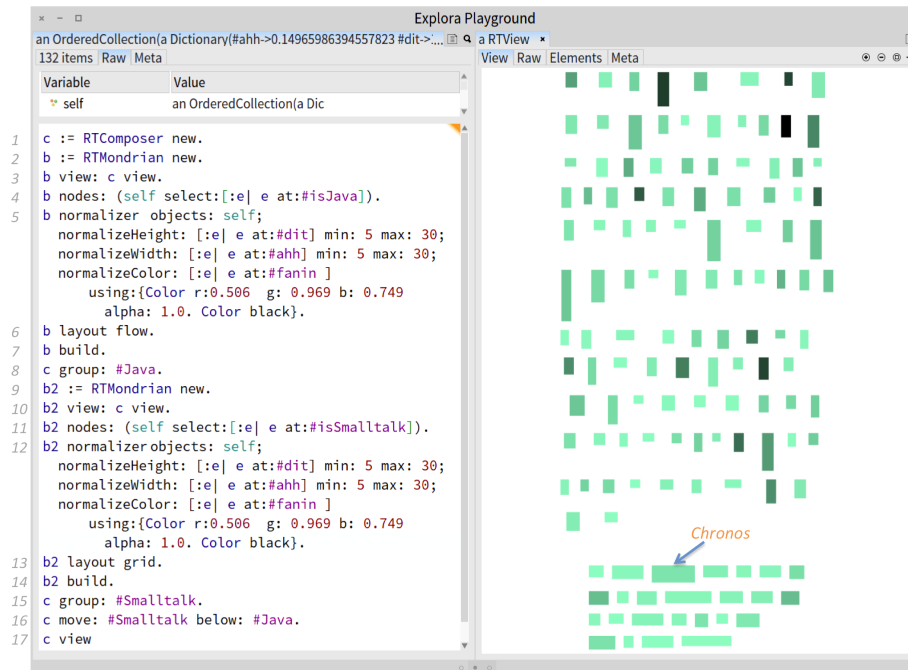


Fig. 2. Visualising AHH and *maximum* values of DIT and Fan-In among Java and Smalltalk systems.

of the box the higher the value of Fan-In. At a first glance, the analysis of the visualisation seems to reveal a pattern. Smalltalk systems are landscape oriented in lighter colour and Java ones have a portrait orientation with a darker colour. Since he took the maximum values of Fan-In and DIT, Edgar realises that in general the Java corpus contains the most invoked class (highest Fan-In), and the deepest hierarchy (highest AHH and DIT). This suggests that Java systems exhibit more reuse than Smalltalk ones. This can be a misleading result due to the decision of aggregating DIT and Fan-In using maximum values, since they do not provide Edgar insight into a general tendency.

Step 3: Exploring alternative Visualisations In consequence, Edgar decides to find out if this pattern still prevails when values are aggregated using the median. He modifies the implementation accordingly, by aggregating the values of DIT and Fan-In using the median. Without leaving the environment, he goes to the left pane shown in Figure 1 and changes lines 9 and 11 accordingly (collecting median instead of maximum values). The Playground recallsl the implementation of the previous visualisation generating a new one automatically (Figure 3). Edgar notes the difference between system *Chronos* in Figure 2 and in Figure 3. The analysis shows that even though it has neither the most

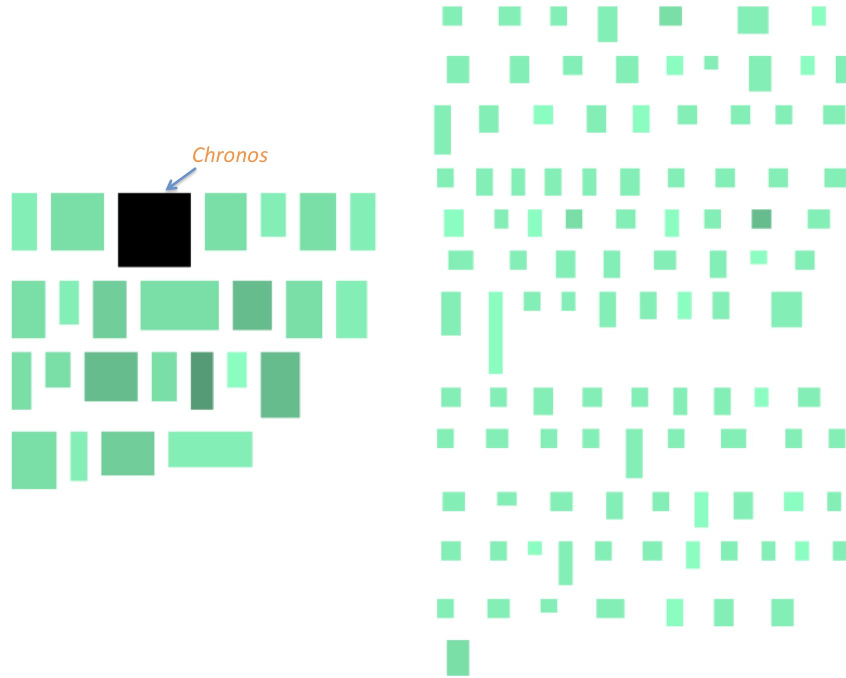


Fig. 3. Visualising AHH and *median* values of DIT and Fan-In among Java and Smalltalk systems.

invoked class in the two corpora (light green in Figure 2), nor the deepest hierarchy (lower height in Figure 2), in general its classes are the most invoked and have deep hierarchies (large and dark box in Figure 3). Figure 3 shows that most systems of the Smalltalk corpus exhibit more reuse by having larger, deeper and more invoked hierarchies.

Diving Into an Individual System Figure 4 shows a detailed visualisation of Chronos which is a library for manipulating dates and times.[3] Classes are represented by circles. The darker the circle, the more invoked the class (higher Fan-In). The size of the circle is mapped to Fan-Out allowing the user to compare classes that behave as clients and providers in invocation relationships. Blue edges between classes show inheritance relationships and grey edges represent invocations. For a better analysis, only invocations of highly invoked classes are shown (Fan-In greater than 90). From the visualisation Edgar can distinguish main provider classes that are highly invoked (dark circles), even though some of them are clients as well (darker and larger circles). Most notably, there is a hierarchy in Figure 4-A that includes many highly invoked classes (*ChronosObject*).

[3] `http://smalltalkhub.com/#!/~Chronos/Chronos`

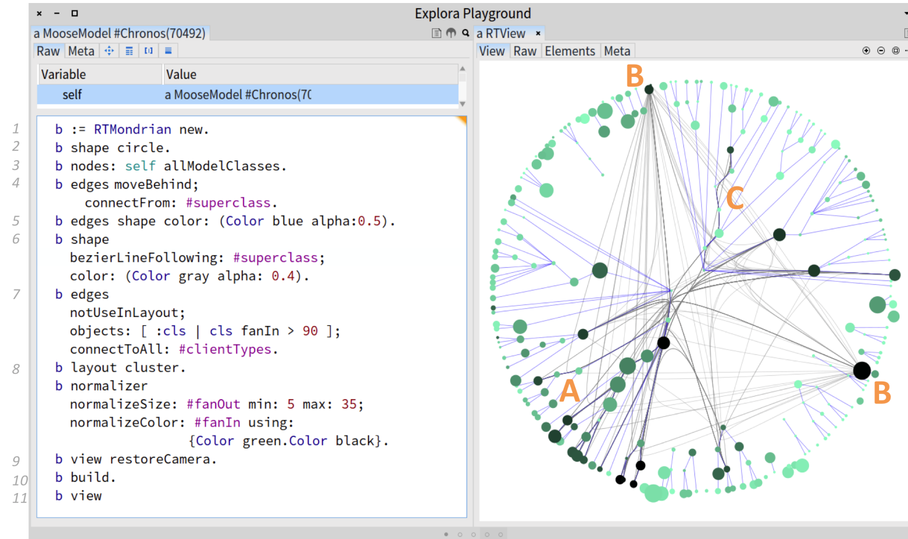


Fig. 4. Drilling-down in the Chronos system. Large and dark nodes represent classes with high fan-out and respectively high fan-in

Besides, there are two classes in Figure 4-B *TimeZoneAnnualTransitionPolicyFactory* and *DateSpec* that without being part of a hierarchy attract many invocations. Finally in Figure 4-C Edgar notes a small hierarchy of chained invocations.

The top-down exploration starting from the visualisation at the corpora level enabled Edgar to decide what systems to focus on. It gave him an overall assessment of reuse between the two corpora as well as a detailed vision of the reuse in a system, when he drilled down into Chronos. He learned that the greater specialisation of Smalltalk systems seems not to affect reuse. Indeed he found deep hierarchies in the Chronos system but he also found that those hierarchies are heavily reused.

3 Technical Infrastructure

When designing Explora we combined several tools for accomplishing the analysis task. Explora is inspired by Pangea [2], and uses the Moose [8] platform for analysing FAMIX [20] models of software systems. It reuses Object Model Snapshots (OMS) from Pangea's data model. An OMS is a custom Moose image containing a single FAMIX model of a system. The model currently includes OMSs of two corpora: 1) Qualitas Corpus [19] and 2) SqueakSource-100 [2].

Explora is written in Pharo, an open-source Smalltalk dialect. The Pharo live programming environment allows users to explore and navigate data in a

dynamic fashion. Explora uses a Playground built on top of the Moldable inspector [3] of Pharo for querying the model, and manipulating results. The Roassal visualisation engine [1] provides a comprehensive API for visualising data in an agile fashion. Roassal provides several Domain-Specific Languages including Mondrian and Grapher.

Workflow

1) The user defines and triggers a query for collecting data from the corpora.
2) A main process looks for OMSs available in a local folder called the source workspace, and evaluates the query in each of them (there are sequential and parallel modes). An OMS is used as a cache holding a live version of the system model that can be awakened, queried, and put back to sleep again.
3) The independent result returned by each OMS is serialised using Fuel [7].
4) These partial results are aggregated to be returned to the user.
5) The user decides whether to go back to 1) or continue with the following step.
6) The user can manipulate the results by filtering, sorting, inspecting or using them as input for a visualisation.

Although the architecture scales up by adding more OMSs horizontally, it is still constrained to the memory available in the Pharo 32 bit environment for materialising objects. A workaround for this issue is to collect less data by including into each OMS only essential information.

Example's Benchmark Table 1 shows a benchmark with the performance results and memory consumption at each step when we run the example analysis. The data were collected by: 1) calling the garbage collector; 2) measuring the memory used (average among 10000 times); 3) executing the step; 4) measuring the memory used (average among 10000 times); and 5) calculating the difference between 4) and 2).

Note that during the execution of the step 1 (Computing the Metrics) new processes are created (in parallel or sequentially) using more memory, however this is released after the execution.

Step	Description	Performance (Secs.)	Memory (MB)
1	Computing the Metrics	39.108	4.3
2	Generating an Initial Visualisation	0.238	2.1
3	Exploring Alternative Visualisation	0.238	2.3
4	Diving into an Individual System	0.212	6.3

Table 1. Performance and memory consumption

4 Future Work

Automatic Visualisation Although Roassal provides a number of expressive DSLs for different tasks, it requires expertise to generate useful visualisations. We envision an approach that exploits expertise of proven well-designed visualisations automatically visualise results. Users without expertise should be able to profit from such visualisations. Lately we are studying how experts visualise software. We have been classifying their visualisations into several dimensions such as goal, domain, granularity. We are working to develop an approach that makes use of this classification to provide automatic visualisation.

Automatic Visualisation Assessment Knowledgeable users who can implement data visualisations would benefit from *Automatic Visualisation Assessment* AVA. While users are implementing visualisations, AVA would give them feedback about visual design guidelines that are violated, and suggest how to resolve them. Our idea is to develop a model of visual design constraints covering the main pitfalls that developers encounter when visualising data, such as visual cluttering, layout selection, and colour conflict.

Expanding the Corpora There are a few systems in Qualitas Corpus that do not fit into an OMS. This will be solved with a 64-bit version of Pharo. We also realise that SqueakSource-100 should be expanded to more systems to provide more interesting results. Finally, we think that adding more corpora from other languages would be an advantage for experimental analysis.

5 Related Work

Explora expands related work by scaling up software visualisation to corpora. Explora's design is based on three pillars: 1) *liveness* of the Pharo environment which enables interactive exploration; 2) ready-to-use software *corpora* models which encourage repeatable analyses; and 3) agile *visualisation* to provide support for data analysis. Figure 5 shows how related work and Explora cover these concepts.

5.1 Visualisation Tools

To the best of our knowledge there is no visualisation tool that provides support for software corpora. Some of them allow users to load into memory several models of systems, but they cannot visualise systems together. Only one of these systems offers liveness.

CodeCrawler [6] is a visualisation tool based on Moose and FAMIX models. It includes many built-in views covering several common software analysis tasks. Views can be partially customised by assigning a specific mapping between the built-in metrics with the visual properties of the representation. CodeCrawler

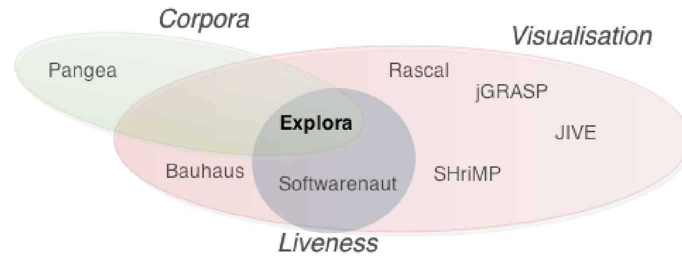


Fig. 5. Explora and related work

was superseded by *Mondrian* [14], a high-level DSL for specifying visualisations. Both of them are meant for analysis of single model systems.

SHriMP [18] visualises software using nested graph views for structural entities such as packages, classes and methods. Edges between artefacts represent dependencies such as inheritance, composition and association relationships. The tool is meant to explore software structure and to navigate source code. Hyperlinks are used ease navigation through source code. SHriMP targets developers analysing their own code or legacy one but always coping with single systems.

jGrasp [4] is a lightweight development environment implemented in Java. Traditional data structures, such as stacks, queues and linked lists, can be easily identified in the visualisation. It is intended to support Java teaching through program visualisation. It only allows the user to visualise an isolated project.

JIVE [10] stands for Java Interactive Visualisation Environment and is mainly used for debugging, maintenance and learning. It provides interactive visualisations of the runtime state and call history of a program. It is integrated in the Eclipse IDE, allowing users to visualise a single project.

Softwarenaut [13] is an analysis tool written in Smalltalk and which profits from its liveness. It visualises software using hierarchical views. It includes three complementary perspectives which allow the user to explore and navigate data. The tool includes pre-packaged metrics that can be mapped to visual properties. Although Softwarenaut allows users to load several model systems into memory, it can only visualise one at a time.

5.2 Data Analysis Tools

Rascal [11] is a Domain Specific Language for source code analysis and manipulation. It is implemented as a plug-in for Eclipse, and consequently benefits from other tools installed in the environment, and exploits Eclipse to obtain software models cheaply. It can only visualise the systems currently loaded in the Eclipse workspace.

Bauhaus [17] is a tool suite written in Ada that supports multi-language program understanding and reverse engineering for maintenance and evolution.

It provides tools to extract, analyse, query and visualise software artefacts. It provides support for analysis and visualisation of single systems.

Pangea [2] is an environment for static analysis of multi-language software corpora. Based on Moose it provides an expressive scripting language. However, since it is implemented as a bash script, it offers neither liveness, nor visualisation. Pangea's output is normally a text file while in Explora it is a live object.

Large-Scale Data Analysis *MapReduce* [5] is a programming model and implementation for processing large datasets. The model is based on the disaggregation of a large dataset into smaller pieces that can be handled by different servers in parallel. The query sent by a client is computed independently by each server. Afterwards, the result of the computation of a server is aggregated. Explora is inspired by MapReduce. In Explora, the corpora are disaggregated into system models which compute queries independently. Explora is not distributed over a network but runs locally.

6 Conclusion

In conclusion, although there are many tools for software analysis and visualisation most of them do not scale to software corpora. Data analysis tools that do scale to corpora are not live. On the other hand, visualisation tools that do offer liveness do not scale to corpora. In this paper we presented Explora, an infrastructure for scaling up software visualisation to corpora. We presented an example of analysis stressing its strengths, showing how visualisation can help one to explore and understand software. However, we acknowledge that useful visualisations are difficult to achieve. In consequence, in the future we want to tackle this issue by automatically visualising software by mapping queries to suitable, proven visualisations. We also think that users with the knowledge for visualising software can profit from automatic visualisation assessment, a dynamic evaluation of the visualisation that provides feedback concerning violations of visual design rules and guidelines.

Acknowledgements

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. Vanessa Peña Araya, Alexandre Bergel, Damien Cassou, Stéphane Ducasse, and Jannik Laval. Agile visualization with Roassal. In *Deep Into Pharo*, pages 209–239. Square Bracket Associates, September 2013.

2. Andrea Caracciolo, Andrei Chis, Boris Spasojević, and Mircea Lungu. Pangea: A workbench for statically analyzing multi-language software corpora. In *Source Code Analysis and Manipulation (SCAM), 2014 IEEE 14th International Working Conference on*, pages 71–76. IEEE, September 2014.
3. Andrei Chiş, Oscar Nierstrasz, and Tudor Gîrba. The Moldable Inspector: a framework for domain-specific object inspection. In *Proceedings of International Workshop on Smalltalk Technologies (IWST 2014)*, 2014.
4. James H Cross, Dean Hendrix, and David A Umphress. jGRASP: an integrated development environment with visualizations for teaching java in cs1, cs2, and beyond. In *Frontiers in Education, 2004. FIE 2004. 34th Annual*, pages 1466–1467. IEEE, 2004.
5. Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008.
6. Serge Demeyer, Stéphane Ducasse, and Michele Lanza. A hybrid reverse engineering platform combining metrics and program visualization. In Francoise Balmas, Mike Blaha, and Spencer Rugaber, editors, *Proceedings of 6th Working Conference on Reverse Engineering (WCRE '99)*. IEEE Computer Society, October 1999.
7. Martín Dias, Mariano Martinez Peck, Stéphane Ducasse, and Gabriela Arévalo. Fuel: a fast general purpose object graph serializer. *Software: Practice and Experience*, 44(4):433–453, 2014.
8. Stéphane Ducasse, Tudor Gîrba, and Oscar Nierstrasz. Moose: an agile reengineering environment. In *Proceedings of ESEC/FSE 2005*, pages 99–102, September 2005. Tool demo.
9. Jean-Marie Favre, Dragan Gasevic, Ralf Lämmel, and Ekaterina Pek. Empirical language analysis in software linguistics. In *Software Language Engineering*, pages 316–326. Springer, 2011.
10. Paul V Gestwicki and Bharat Jayaraman. Jive: Java interactive visualization environment. In *Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 226–228. ACM, 2004.
11. Paul Klint, Tijs van der Storm, and Jurgen Vinju. RASCAL: A domain specific language for source code analysis and manipulation. In *Source Code Analysis and Manipulation, 2009. SCAM '09. Ninth IEEE International Working Conference on*, pages 168–177, 2009.
12. Michele Lanza and Radu Marinescu. *Object-Oriented Metrics in Practice*. Springer-Verlag, 2006.
13. Mircea Lungu, Adrian Kuhn, Tudor Gîrba, and Michele Lanza. Interactive exploration of semantic clusters. In *3rd International Workshop on Visualizing Software for Understanding and Analysis (VISSOFT 2005)*, pages 95–100, 2005.
14. Michael Meyer, Tudor Gîrba, and Mircea Lungu. Mondrian: An agile visualization framework. In *ACM Symposium on Software Visualization (SoftVis'06)*, pages 135–144, New York, NY, USA, 2006. ACM Press.
15. Paloma Oliveira, Marco Tulio Valente, and Fernando Paim Lima. Extracting relative thresholds for source code metrics. In *Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), 2014 Software Evolution Week-IEEE Conference on*, pages 254–263. IEEE, 2014.
16. Ekaterina Pek. *Corpus-based empirical research in software engineering*. PhD thesis, Universitaet Koblenz-Landau, 2013.
17. Aoun Raza, Gunther Vogel, and Erhard Plödereder. Bauhaus — a tool suite for program analysis and reverse engineering. In *Reliable Software Technologies - Ada-Europe 2006*, pages 71–82. LNCS (4006), June 2006.

18. Margaret-Anne Storey, Casey Best, and Jeff Michaud. SHriMP Views: An interactive and customizable environment for software exploration. In *Proceedings of International Workshop on Program Comprehension (IWPC '2001)*, 2001.
19. 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.
20. Sander Tichelaar. *Modeling Object-Oriented Software for Reverse Engineering and Refactoring*. PhD thesis, University of Bern, December 2001.

Detecting Refactorable Clones by Slicing Program Dependence Graphs

Ammar Hamid[1] and Vadim Zaytsev[2]

[1] ammarhamid84@gmail.com
[2] vadim@grammarware.net
Institute of Informatics, University of Amsterdam, The Netherlands

Abstract. Code duplication in a program can make understanding and maintenance difficult. The problem can be reduced by detecting duplicated code, refactoring it into a separate procedure, and replacing all the clones by appropriate calls to the new procedure. In this paper, we report on a confirmatory replication of a tool that was used to detect such refactorable clones based on program dependence graphs and program slicing.

1 Motivation

With the discussion about the extent to which code clones are harmful for software readability, maintainability and ultimately quality, still ongoing, there is still significant evidence on cost increases being caused by code duplication in at least some scenarios [5,18]. For simplicity, we intend to adopt that view and look a step further. Once clones are identified, ideally we would like to provide advanced support for programmers or maintenance engineers to remove them — that is, to use refactorings [4] to "de-clone" source code by merging identical code fragments and parametrising similar ones [17].

The sheer number of code clone detection techniques and tools is immensely overwhelming [15,16,13]. In section 2, we will give a very brief overview of the field and explain terminology needed to understand the rest of the paper. One of the promising family of methods which is not too complex for a final Master's project yet also not too much of a beaten track in code clone research, is *graph-based*. Given two programs, we automatically build a graph-like structure with known properties, employ some slicing and/or matching and based on that can diagnose them with duplication.

Eventually we have converged to a relatively well-known paper of Raghavan Komondoor and Susan Horwitz [8] and dedicated ourselves to replicating it. Some details about that project can be found in section 3, but in general they propose to use program dependence graphs [11] (PDG) and program slicing [19]. The authors of the original study were able to find isomorphic subgraphs of the PDG by implementing a program slicing technique using a combination of backward slicing and forward slicing. Basically they search for sets of syntactically equivalent node pairs and perform backward slicing from each pair with a

single forward slicing pass for matching predicates nodes. The theoretical foundation behind this method mostly lies in plan calculus [14] and an advanced graph partitioning algorithm [6,7] and essentially allows to detect clones semantically, regardless of various refactorings that may have been applied to some of the copies but not to others. This leads to reporting only those clones that can indeed be refactored — as we show on Table 1.

We modified the original study in several ways: some were forced upon us by technicalities, for others we had our own reasons — all explained in section 4. We report our results, compare them to the original study and try to explain the differences in section 5 and conclude the paper with section 6.

2 Background

Several studies show that 7–23% of the source code for large programs is duplicated code [2,9]. Within a project, code duplication will increase code size, and make maintenance difficult. Fixing a bug within a project with a lot of duplicated code is becoming a challenge because it is necessary to make sure that the fix is applied to all of the duplicated instances. Lague et al [10] studied the development of a large software system over multiple releases and found that programmers often missed some copies of the duplicated code when performing modification. Similar results have been observed by Geiger et al [5] and Thummalapenta et al [18] who observe the already expected negative impact of clone co-evolution on software maintenance effort.

In code duplication studies we usually distinguish among the following types of clones [13,15]:

- *Exact clones* (type 1) — identical duplicates with some variations allowed in whitespace and comments;
- *Parametrised clones* (type 2) — variations are allowed in identifier names, literals, even variable types;
- *Near miss clones* (type 3) — statements are allowed to be changed, added or removed up to some extent;
- *Semantic clones* (type 4) — same computation with a different syntax and possibly even different algorithms;
- *Structural clones* — higher level similarities, conceptually bottom-up-detected implementation patterns;
- *Artefact clones* — function clones and file clones;
- *Model clones* — duplicates over artefacts other than code;
- *Contextual clones* — code fragments deemed duplicate due to their usage patterns.

Out of these, type 2 and type 3 are the most well-researched ones, with model clones quickly getting more and more attention every year.

Techniques and tools can be roughly classified into these groups [13,16] (in the parenthesis we show a software artefact category in the terms of parsing-in-a-broad-sense megamodel [20]):

Procedure 1

```
 int foo(void) {
++   int i = 1;
     bool z = true;
     int t = 10;
++   int j = i + 1;
++   int n;
++   for (n=0; n<10; n++) {
++      j = j + 5;
     }
++   int k = i + j - 1;
     return k;
 }
```

Rewritten Procedure 1

```
 int foo(void) {
     bool w = false;
     int t = 10;
**   return new_procedure_bar();
 }
```

Procedure 2

```
 int bar(void) {
++   int i = 1;
     bool w = false;
     int t = 10;
++   int s;
++   int b = a + 1;
++   for (s=0; s<10; s++) {
++      b = b + 5;
     }
++   int c = a + b - 1;
     return c;
 }
```

Rewritten Procedure 2

```
 int bar(void) {
     bool w = false;
     int t = 10;
**   return new_procedure_bar();
 }
```

Newly extracted procedure:

```
 int new_procedure_bar(void) {
++   int i = 1;
++   int j = i + 1;
++   int n;
++   for (n=0; n<10; n++) {
++      j = j + 5;
     }
++   int k = i + j - 1;
     return k;
 }
```

Table 1. Two functions with duplicated code and a refactoring result. In the left column, the duplicated code is marked with `++`; in the right column clones are replaced with calls to a newly extracted function. This example demonstrates that not everything that has the same structure or the same syntax is reported as clones (e.g. `int t = 10;` which has no shared predecessor).

- *Text based* (Str) such as SIMIAN — blazingly fast methods usually looking for exact clones, quite often in a language-independent, -parametric or -agnostic manner;
- *Token based* (TTk, Lex) such as CCFINDER — somewhat more sophisticated lexical tools;
- *Tree based* (Ptr, Cst, Ast) such as DECKARD — looking for clones in parse trees, suffix trees or abstract syntax trees;
- *Graph based* (Ast, Dia) such as DUPLIX — making decisions based on control flow graphs, data dependency graphs, program dependence graphs or partite sets and vertices;
- *Model based* (Dia) such as CONQAT — metamodel-specific representation, usually graph-like;
- *Metrics based* such as COVET — using metrics, fingerprinting and/or clustering to work on text or ASTs;
- *Hybrid* such as CLONEMINER — independent component analysis, some variants of semantic indexing and longest subsequence methods that require reasoning over trees, memory states, vector spaces, etc.

Following the original paper [8], we use CodeSurfer[3], a commercial tool that can be used to generate program dependence graphs (PDGs) from C programs. It provides an API that can be used from Scheme programs [1]. In general, PDG nodes represent program statements and predicates, while PDG edges represent data and control dependencies. PDG provides an abstraction that ignores arbitrary sequencing choices made by a programmer, and instead captures the important dependences among program components. Essentially, a program dependence graph is built starting from a control flow graph (CFG) with statements as nodes and possible transitions among them as edges, which is then analysed for dominance to form an acyclic post-dominator graph — the two are merged into a control dependence graph. A program dependence graph is formed from the control dependence graph by enhancing it with additional edges for all data dependencies, an example is given on Figure 1. The resulting complex structure is graph-like with nodes of several kinds (regions, statements, entry/exit points) and edges of several kinds (data/control dominance, possibly labelled) — there are algorithmic variations which are not important for understanding the current paper. Such a PDG is remarkable in a sense that it captures many structural aspects of a program and still allows to abstract from concrete details such as variable names and precise positioning of the code. For a larger/smaller scale, related methods are used such as system dependence graphs (SDGs) or execution dependence graphs (EDGs) [12].

The last bit of background needed for understanding this paper is program slicing [19,3], which is a well-known technique for obtaining a "view" of a program with only those statements that are relevant for the chosen variable. In terms of PDG we can have two query types in program slicing [7]. *Backward slicing* from node x means finding all the nodes that influence the value of node

[3] CodeSurfer, `http://www.grammatech.com/research/technologies/codesurfer`.

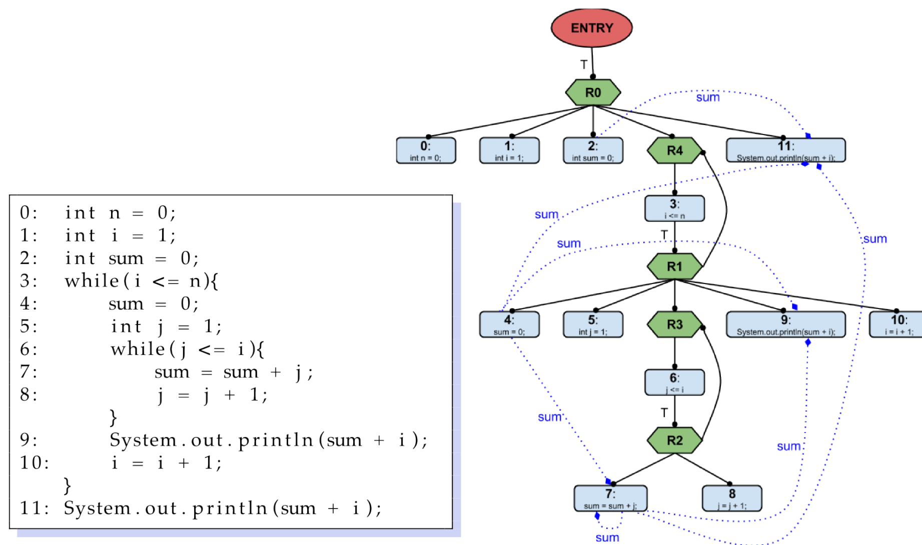


Fig. 1. A tiny code fragment demonstrating the concept of a program dependence graph: the listing on the left; the corresponding PDG fragment on the right (only data dependencies for *sum* are shown, the complete graph is much bigger and more cluttered) [21].

x. *Forward slicing* from node y means finding all the nodes that are influenced by node y. This is an important technique to filter out any statements that are irrelevant for clone detection.

3 Original study

The main research question asked by Komondoor and Horwitz is the following: can we find code clones of type 3 (non-contiguous, reordered, intertwined), which are refactorable into new procedures? [8]

3.1 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

Find relevant procedures. We are only interested in finding clones for procedures that are reachable from the main program execution. Only then we can safely remove unreachable procedures from our program and just not detect clones of them. We do this by getting a system initialisation vertex and forward-slicing 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.

Find pairs of vertices with equivalent syntactic structure. We scan all PDGs from the previous step to find vertices that have the 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*).

Find clones. From a pair of equivalent structure expressions, we back-slice 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.

Group clones. This is the step where we make sense of the found clones before displaying them. For example, when using CodeSurfer, 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 are mapped back to the actual program text representation and grouped together before displaying them. It is important that the programmer can understand and take action on the reported clones.

Experimental setup. The authors of the original paper used CodeSurfer version **1.8** to generate PDGs and wrote Scheme program of **6123** lines that access CodeSurfer API to work with the generated PDGs. They also had to have a C implementation of **4380** lines to do the processing of those PDG to actually find clones.

They were able to find isomorphic subgraphs of the PDG by implementing a program slicing technique that used a combination of backward slicing and forward slicing. They applied it to some open-source software written in C (`tail`, `sort`, `bison`) and demonstrated the capability of slicing to detect non-contagious code clones. We will show the actual numbers later when we compare the results with the replication.

4 Changes to the original study

The motivation of this replication study is to be able to validate algorithm and results of the original study. Once validated, we would like to publish our code and intermediate results into a public repository so that it is easier for any future researchers to either re-validate our results or to extend our program.

We had to use CodeSurfer 2.3 instead of 1.8 used in the original study: just to get it running was already a challenge impossible to overcome — we would eventually need to do a sandboxing of some 2001 version of OS, which would then need to be properly licensed (CodeSurfer is not open source, but we have applied for the academic license and got both 1.8 and 2.3 to experiment with).

Porting the existing code (kindly provided to use by Raghavan Komondoor) to the new version of CodeSurfer was also ruled out as a viable option: the API changed too much, and actually covered many things with standard calls that needed to be programmed in full when working with version 1.8. In the end, we reimplemented the algorithm from scratch, using both the original paper and the code behind it as guides. Our implementation has **536** LOC of Scheme, which is huge improvement against the 6123 LOC of the original study. The improvement is mostly not ours to claim, but CodeSurfer API's. For post-processing of clones, we wrote a Ruby script, which was again shorter: 161 LOC versus the original 4380 LOC, partly due to the improved API, but partly also due to the language choice (the original post-processing was done in C++). Actually, given a bit more time, it should have been possible to avoid post-processing entirely, or rather to implement in all in Scheme. The code is available online for anybody to do this — `http://github.com/ammarhamid/clone-detection` — we accept pull requests.

There are several other important changes from the original paper that we need to explain. As mentioned above, we only detect clones within the reachable procedures, excluding any unused procedures that are not reachable from main execution. This makes the result more accurate, since dead code is out of our consideration.

Furthermore, we only use backward slicing and no forward slicing to detect clones. Let us have a look at the example on Table 2. According to the original paper, only statements indicated by `++` will be reported as clones while statement marked with `**` is excluded. The main argument according to the original paper is that `fp3` is used inside a loop but the loop predicate itself is not matching (for loop and the first while loop predicate doesn't match) – or a so called cross loop [8].

However, we argue that we should still report the statement marked with `**` as a clone together with the fact that their loop predicate doesn't match. For a software developer it would mean one could still refactor this into two separate procedures, instead of a single procedure proposed by the original paper (Table 3 and Table 4). Therefore, we consider that *forward slicing is only necessary to define refactoring strategy* and not for *detecting the clone* itself.

Fragment 1

```
   ...
** fp3 = lookaheadset + tokensetsize;
   for(i = lookaheads(state);
       i < k; i++) {
++     fp1 = LA + i * tokensetsize;
++     fp2 = lookaheadset;
++     while (fp2 < fp3) {
++         *fp2++ |= *fp1++;
       }
   }
```

Fragment 2

```
 ...
** fp3 = base + tokensetsize;
   ...
   if(rp) {
       while((j = *rp++) > 0) {
           ...
++         fp1 = base;
++         fp2 = F + j * tokensetsize;
++         while(fp1 < fp3) {
++             *fp1++ |= *fp2++;
           }
   }
```

Table 2. Two clones from bison that illustrates the necessity to have a forward slicing according to the original paper [8]

The new fragment 1

```
...
fp3 = location(lookaheadset,
               tokensetsize);
...
for(i = lookaheads(state);
    i < k; i++) {
    compute(LA, lookaheadset,
       i, tokensetsize, fp3);
}
```

The new fragment 2

```
...
fp3 = location(base, tokensetsize);
...
if(rp) {
    while((j = *rp++) > 0) {
       ...
       compute(F, base,
             j, tokensetsize, fp3);
    }
}
```

Table 3. The new fragments after refactoring (without forward slicing)

The extracted procedures

```
int location(int base, int size) { return base + size; }

void compute(int cons, int base, int index, int size, int loc) {
    fp1 = cons + index * size;
    fp2 = base;
    while (fp2 < loc) { *fp2++ |= *fp1++; } }
```

Table 4. The new refactored procedures. In this case, procedure location has only one statement which probably unnecessary to create a new procedure for it. But the point is if we use forward slicing in clone detection phase, we might hide this statement prematurely from the programmers, who at least should be aware of the situation before proceeding with refactoring.

Study	Program	LOC	PDG nodes	Elapsed time, minutes:seconds		
				Scheme	C++	Ruby
Original	tail	1569	2580	00:40	00:03	—
Replication	tail	1668	3052	00:05	—	00:01
Original	sort	2445	5820	10:00	00:07	—
Replication	sort	2499	6891	00:30	—	00:01
Original	bison	11540	28548	93:00	01:05	—
Replication	bison	10550	33820	126:00	—	00:42

Table 5. Comparison on program size, number of nodes, implementation and time.

5 Results

To be as close to the original paper as possible, we used the GNU git repositories[4] to locate versions that were released around 2001: CoreUtils 4.5.2 (for `tail` and `sort`) and Bison 1.29 (for `bison`).

Table 5 shows the comparison of the sizes of the three programs (in number of LOC and in number of nodes), and the running times for the algorithm between the original and replication study. Figure 2 shows the comparison of the result in details between the original and replication study.

We do not have a solid explanation for the differences observed, but we can hypothesise on some issues:

Altered algorithm. We did use a slightly different algorithm (only reachable code; no forward slicing) to detect clones. However, we have also tried running it exactly as it was intended originally, and the differences were rather minor and could not explain some of the drastic differences.

Manual inspection was performed to ensure that the clones detected by our tool are indeed clones and are indeed refactorable. It was possible to review all clones from `tail` and `sort` and cover a random selection of clones for `bison` — no false positives were found.

Bison running time in the original study is suspiciously short, which does not reflect the explosive performance behaviour that we have observed in our implementation. This could indicate a bug in one of the implementations, or point to a drastically different (optimised, distributed) algorithm used for the actual run of the original experiment. It could also be a simple reporting mistake (e.g., "one and a half hours" reported instead of actual "one and a half days").

Size of some clones reported for `tail` and `bison` is longer than most functions (group 70+), which means either a mistake or some unreported procedure used in the original experiment to combine several subsequent full-function clones into one.

Testing a program of 10 KLOC is always harder than testing a program of 1 KLOC, especially if both programs are algorithmically heavy yet the shorter one relies on a more advanced API. More investigation is needed to see which of these factors were at play and which results are closer to the truth.

[4] http://git.savannah.gnu.org/cgit/

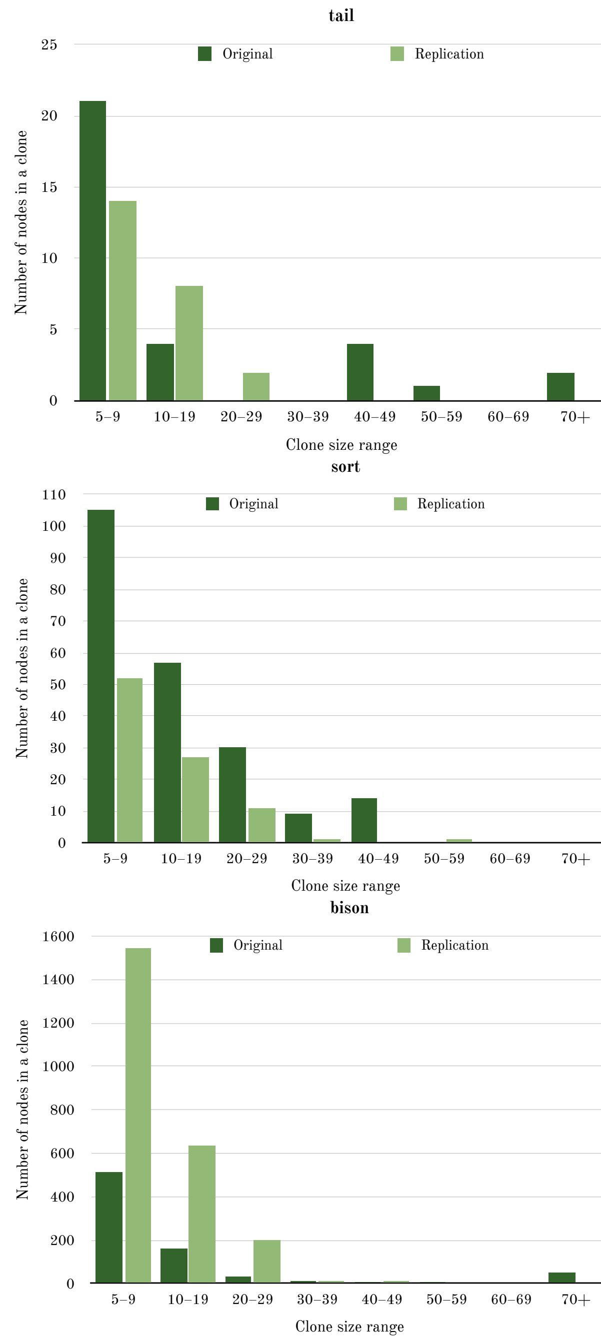


Fig. 2. Detailed comparison results between the original and replication study

6 Conclusion

We have departed on a quest to find refactorable semantic clones and have conducted a replication of a paper that did it with PDG and program slicing. Our results are statistically somewhat different from the results of the original study, but we can conclude nevertheless that the algorithm described there, works. So, *the fusion of PDG and slicing is suitable for Type 3 clone detection.*

As a side product, we have noticed how significantly CodeSurfer has improved over the years: the amount of code we needed to write to achieve the same objectives, is ten times less than what had to be done 13 years ago, with almost no postprocessing of the obtained results needed.

As for quantitative differences, unfortunately we could not compare them in detail since we lack the original data, and we failed in getting the code operational (it would require an old version of CodeSurfer operating on an old system, preferably with performance comparable to the machine used for the original experiment). However, we do present some evidence of correctness in the form of manually reviewed code clones that we reported. We can also conclude that the clones are indeed refactorable — this has been evaluated through manual inspection of the tool reports.

Both the code and the intermediate results of our experiments have been shared as open source: `http://github.com/ammarhamid/clone-detection`, to make it easier to revalidate, replicate, and extend. We hope our clone detector is a suitable tool to use for future work. Possible future extensions should include detecting interprocedural clones as well, which would allow detection of type 4 clones and refactorings such as inlining variables and extracting methods. Intuitively, it would be more useful to provide results over bigger related code fragments — however, the practical consequences remain to be seen.

Acknowledgement

We would like to thank Raghavan Komondoor for sharing his code: we ended up not using it directly, but having it at hand helped us to understand some details and make a better comparison. We would also like to thank David Vitek from GrammaTech for helping us obtaining an academic license for CodeSurfer and trying to work us through the changes from 1.8 to 2.3 — again, we ended up not opting for a migration, but quickly estimating that to be too much of an undertaking, was a part of this project's success. Last but not least, our thanks go to the participants of SATToSE 2014 for all the discussions we have had in L'Aquila.

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. B. S. Baker. On Finding Duplication and Near-Duplication in Large Software Systems. In *WCRE*, pages 86–95, 1995.
3. J. Beck and D. Eichmann. Program and Interface Slicing for Reverse Engineering. In *ICSE*, pages 509–518. IEEE, 1993.
4. M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design or Existing Code*. Addison-Wesley Professional, 1999.
5. R. Geiger, B. Fluri, H. C. Gall, and M. Pinzger. Relation of Code Clones and Change Couplings. In *FASE*, pages 411–425. Springer, 2006.
6. S. Horwitz. Identifying the Semantic and Textual Differences Between Two Versions of a Program. In B. N. Fischer, editor, *PLDI*, pages 234–245, 1990.
7. S. Horwitz, T. W. Reps, and D. Binkley. Interprocedural Slicing Using Dependence Graphs. *ACM ToPLaS*, 12(1):26–60, 1990.
8. R. Komondoor and S. Horwitz. Using Slicing to Identify Duplication in Source Code. In *SAS*, pages 40–56. Springer, 2001.
9. K. Kontogiannis, R. de Mori, E. Merlo, M. Galler, and M. Bernstein. Pattern Matching for Clone and Concept Detection. *ASE*, 3(1/2):77–108, 1996.
10. B. Laguë, D. Proulx, J. Mayrand, E. Merlo, and J. P. Hudepohl. Assessing the Benefits of Incorporating Function Clone Detection in a Development Process. In *ICSM*, pages 314–321, 1997.
11. K. J. Ottenstein and L. M. Ottenstein. The Program Dependence Graph in a Software Development Environment. In *SDE*, pages 177–184, 1984.
12. J. Qiu, X. Su, and P. Ma. Library Functions Identification in Binary Code by Using Graph Isomorphism Testings. In Y.-G. Guéhéneuc, B. Adams, and A. Serebrenik, editors, *SANER*, pages 261–270. IEEE, Mar. 2015.
13. D. Rattan, R. K. Bhatia, and M. Singh. Software Clone Detection: A Systematic Review. *Information & Software Technology*, 55(7):1165–1199, 2013.
14. C. Rich and R. C. Waters. *The Programmer's Apprentice*. ACM, 1990.
15. C. K. Roy, J. R. Cordy, and R. Koschke. Comparison and Evaluation of Code Clone Detection Techniques and Tools: A Qualitative Approach. *SCP*, 74(7):470–495, 2009.
16. C. K. Roy, M. F. Zibran, and R. Koschke. The Vision of Software Clone Management: Past, Present and Future. In S. Demeyer, D. Binkley, and F. Ricca, editors, *CSMR-WCRE*, pages 18–33. IEEE, 2014.
17. R. Tairas. Clone Detection and Refactoring. In *OOPSLA*, pages 780–781, 2006.
18. S. Thummalapenta, L. Cerulo, L. Aversano, and M. D. Penta. An Empirical Study on the Maintenance of Source Code Clones. *EMSE*, 15(1):1–34, 2010.
19. M. Weiser. Program Slicing. *IEEE TSE*, 10(4):352–357, 1984.
20. V. Zaytsev and A. H. Bagge. Parsing in a Broad Sense. In J. Dingel, W. Schulte, I. Ramos, S. Abrahão, and E. Insfran, editors, *MoDELS*, volume 8767 of *LNCS*, pages 50–67. Springer, Oct. 2014.
21. L. Zhang. Implementing a PDG Library in Rascal. Master's thesis, Universiteit van Amsterdam, The Netherlands, Sept. 2014.

A Critique on Code Critics

Angela Lozano *, Gabriela Arévalo **, and Kim Mens

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

Universidad Abierta Interamericana
Av. Montes de Oca 745
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 a recommendation facility of the Pharo Smalltalk IDE. They signal controversial implementation choices such as code smells at class and method level. They aim to promote the use of good and standard coding idioms for increased performance or a better use of object-oriented constructs. This paper studies relations among code critics by analyzing co-occurrences of code critics detected on the Moose system, a large and mature Smalltalk application. Based upon this analysis, we present a critique on code critics, as a first step towards an improved grouping of code critics that identifies issues at a higher level of abstraction, by combining lower-level critics that tend to co-occur, as well as improvements in the definition of the individual critics.

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

1 Introduction

A plethora of code recommendation tools exists to support developers when coding a software system. Whereas some of these recommendations remain at a high level of abstraction (*e.g.*, low coupling and high cohesion), others are much more specific (*e.g.*, 'classes should not have more than 6 methods').

Research on recommendation systems to detect and correct controversial implementation choices typically follows a top-down approach. Recommendations defined at a high level of abstraction are refined into the detection of more concrete symptoms until a straightforward detection strategy is reached. Different recommendation approaches exist that detect issues like design flaws [12] or antipatterns [13]. While these approaches discover similar issues, they often vary significantly in the heuristics, metrics and thresholds they use. These differences have various causes. Heuristics are incomplete by definition. The definition of many metrics remains open to interpretation resulting in different tools that may provide different results for the same metric. And thresholds used tend to be either absolute values that cannot be reused across different applications, or

* Angela Lozano is financed by the CHaQ project of the Flemish IWT funding agency.
** also DCyT - Universidad Nacional de Quilmes and CONICET - Buenos Aires, Argentina

relative values whose cut point may be arbitrary. For these reasons, it is difficult to justify that concrete detection strategies and how they are combined into higher-level recommendations accurately represent all and only those entities that a higher-level recommendation aims to capture.

As opposed to defining high-level recommendations as an ad-hoc combination of lower-level issues, this paper presents a first step towards 'discovering' higher-level recommendations from a detailed analysis of the occurrence of more specific low-level ones. More specifically, our analysis is based on a study and possible interpretation of the *co-occurrence* of low-level recommendations in several applications.

The low-level issues analyzed in this particular paper are the so-called *code critics*. Code critics are a list of detectors for harmful implementation choices in Pharo Smalltalk that signal certain defects or performance issues in Smalltalk source code, mainly in methods and classes. Each critic is defined with a short name and a rationale that explains why that implementation choice could be harmful and, in some cases, also proposes a refactoring. An example of such a code critic is the critic named *'Instance variables not read AND written'* with rationale:

> "Checks that all instance variables are both read *and* written. If an instance variable is only read, you can replace all of the reads with nil, since it couldn't have been assigned a value. If the variable is only written, then we don't need to store the result since we never use it. This check does not work for the data model classes, or other classes which use the `instVarXyz:put:` messages to set instance variables."

Although code critics sometimes report false positives (like the `instVarXyz:put:` messages mentioned in the rationale of the critic above), the Code Critics browser allows one to 'ignore' each reported result individually. Results that have been ignored are saved within the image[1], so that the system remembers that they have been ignored and does not present them again to the developer when the same code critics are checked again later.

Each code critic belongs to one of the following categories: *Unclassified* rules, *Style* issues, *Coding Idiom Violations*, suggestions for *Optimization*, *Design Flaws*, *Potential Bugs*, actual *Bugs* and likely *Spelling* errors. For instance, the code critic named *'Instance variables not read AND written'* is categorized as an *Optimization* issue.

This paper is structured as follows: Section 1 detailed the problem and context of low-level recommendation tools. Section 2 introduces the concept of code critics in more detail, and Section 3 shows how we define the distance function to calculate if code critics co-occur in the analyzed application. Section 4 presents critiques on individual critics and several patters of co-occurring critics. Section 5 concludes our work and presents some future work.

---

[1] Smalltalk systems store the entire program and its state in an image file.

2 An Introduction of code critics

Although Pharo's *Critic Browser* is designed to be launched by a developer from a menu in the IDE, the tool can also be run programmatically to analyze part of the image with a selected set of critics. In our experiment, we analyzed 120 code critics, 27 applied to classes, and 93 applied to methods. We excluded the category of *Spelling* rules, which check the spelling of comments and identifiers of classes, methods and variables. We are less interested in these rules as they do not refer to either the structure or design of the source code, and tend to generate quite some noise in the results.[2]

Id	Critic name
CC01	A metamodel class does not override a method that it should override
CC02	Class not referenced
CC03	Class variable capitalization
CC04	Defines = but not hash
CC05	Excessive inheritance depth
CC06	Excessive number of methods
CC07	Excessive number of variables
CC08	Instance variables defined in all subclasses
CC09	Instance variables not read AND written
CC10	Method defined in all subclasses, but not in superclass
CC11	No class comment
CC12	Number of addDependent: messages > removeDependent:
CC13	Overrides a 'special' message
CC14	References an abstract class
CC15	Refers to class name instead of 'self class'
CC16	Sends 'questionable' message
CC17	Subclass responsibility not defined
CC18	Variable is only assigned a single literal value
CC19	Variable referenced in only one method and always assigned first
CC20	Variables not referenced

Table 1. Some of the most frequent class-level critics and their identifiers.

Table 1 lists some of the most frequently found class-level code critics, and Table 2 lists some discovered method-level critics. We added an identifier to each of them for easy reference later. For example, CC09 refers to the code critic *'Instance variables not read AND written'*.

For our analysis we studied Moose [5], a Smalltalk platform consisting of a variety of software and data analysis tools. More specifically, we analyzed all packages contained in the downloadable image containing the latest distribution of Moose (i.e., Pharo 1.4). For each package studied (71 in total) we accumulated all critics found in methods and classes, except for those methods and classes

[2] For the same reason they do not even appear in recent versions of the *Critic Browser*.

Id	Critic name
MC01	detect:ifNone: -> anySatisfy:
MC02	Inconsistent method classification
MC03	Law of Demeter
MC04	Methods implemented but not sent
MC05	Rewrite super messages to self messages when both refer to same method
MC06	Sends different super message
MC07	Temporaries read before written
MC08	Unclassified methods
MC09	Uses detect:ifNone: instead of contains:
MC10	Utility methods

Table 2. Some common method-level critics and their identifiers.

related to tests. We excluded the tests because critics about test code often lead to false positives. Test code tends to adhere to other idioms than ordinary code. For instance, test code often contains duplicated code between test methods (due to similar calls to 'assert' or other testing methods). Moreover, test code often contains trial-and-error code to deal with all cases to be tested, which is typically not considered good practice in normal code.

3 Critiques on individual and co-occurring code critics

Our analysis generates two boolean tables per package: one for its classes and another for its methods. Each table shows which source code entities suffer from which critics. Each column represents a method or class of the package, and each row represents which entities are in the result set of a code critic. E.g., suppose we analyze the following class-level code critics in the package Compiler (which is part of the analyzed distribution): *'Instance variables not read AND written' (CC09), 'Sends 'questionable' message' (CC16), 'Excessive number of variables' (CC07), 'Excessive number of methods' (CC06)* and *'Variables not referenced' (CC20).* Table 3 presents the results[3], where the rows identify the *critiqued* entities for a corresponding critic in the analyzed package. In other words, *critiqued*(c, p) is a sequence of boolean values $< c(e_1), c(e_2), \ldots, c(e_n) >$ where $c(e_i) = true$ if and only if e_i is the ith entity in package p (by alphabetic order on its fully qualified name), and e_i is in the result set of code critic c.

Next, we calculate the distance between pairs of critics based on the entities they *critique*. The distance between two code critics c_1 and c_2, for a given package p, is calculated by counting the number of classes or methods where the critics do not match (XOR of the critiqued entities), over the number of classes

[3] To limit the size of the example, this table present only a subset of all classes that were critiqued. However, for the sake of the example, in order to illustrate how the approach works, we ask the reader to assume that the classes shown in Table 3 are all the critiqued classes in the package.

	AmbiguousSelector	BlockNode	Encoder	LiteralVariableNode	VariableNode	CommentNode	UndVariableReference	MessageNode	AssignmentNode	ParseNode	MethodNode	Decompiler	Compiler	Parser	BytecodeEncoder	TempVariableNode
CC09	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0
CC16	0	1	1	0	1	0	0	1	1	1	1	1	1	0	0	0
CC07	0	1	1	0	0	0	0	1	0	1	1	1	0	1	0	0
CC06	0	1	1	0	0	0	0	1	0	1	1	1	0	1	1	1
CC20	0	1	0	0	0	0	0	1	0	1	0	0	0	0	0	0

Table 3. Code critics (CC) per class for the Compiler package.

	CC09	CC16	CC07	CC06	CC20
CC09	0	0.81	0.77	0.81	0.83
CC16	0.81	0	0.40	0.50	0.66
CC07	0.77	0.40	0	0.22	0.57
CC06	0.81	0.50	0.22	0	0.66
CC20	0.83	0.66	0.57	0.66	0

Table 4. Distance among the code critics shown in Table 3.

or methods that violate one or both of the critics being analyzed (OR of the critiqued entities). This distance value varies between zero and one. Values close to zero mean that a pair of critics tends to affect the same source code entities.

$$D_p(c_1, c_2) = \frac{|critiqued(c_1, p)\ \oplus\ critiqued(c_2, p)|}{|critiqued(c_1, p)\ \vee\ critiqued(c_2, p)|}$$

For instance, Table 5 calculates the distance between *'Instance variables not read AND written'* and *'Variables not referenced'* based on the presented example. The resulting distance, shown as a shaded cell in Table 4, is 0.83 (i.e., 5/6) because their results differ in five classes, but coincide in one class (BlockNode). Therefore, the critics have low co-occurrence for the results of this package.

CC09: Instance variables not read & written	1111000000000000
CC20: Variables not referenced	0100000101000000
XOR	1011000101000000
OR	1111000101000000

Table 5. Calculation of the distance between a pair of critics based on their results for a given package (shown in table 3).

Using the Boolean table 3 and the distance table 4 we proceed to discard pairs of code critics that do not seem interesting for our analysis, based on three criteria. First, pairs with high distances (greater than 0.9) are discarded as they tend not to co-occur often and therefore are likely to represent accidental matches. Secondly, we discard pairs of critics that *always* occur together (distance zero) in the same source code entities, because they are likely to represent alternative implementations of a same code critic. Thirdly, we exclude all pairs of critics for which one of the code-critics covers more than 90% of all source code entities analyzed, because as a consequence of their high coverage they will show a strong correlation with nearly all other code-critics and thus generate significant noise in the results. In our example, all distances are kept in our analysis. The choice of thresholds of 0.9 and 90% was based on initial experiments where we tried to determine what values would constitute a good cut point to discard less relevant pairs of critics. However, these thresholds should be reevaluated when applying the approach to other code critics, other applications, or different programming languages.

4 Identified patterns

Based on the raw results of our initial analysis, this section presents some interesting critiques which we have observed. Since this is a preliminary research, we do not claim these critiques to be exhaustive nor complete. In the text below, we use the word *critique* to denote the identified patterns in our analysis, and *critic* to refer to Pharo's code critics. We present our critiques as patterns, consisting of a short name, a description and some concrete examples. The patterns are divided in two big categories. The first category describes the critiques discovered by analyzing individual code critics. Note that we limited our analysis of individual code critics to those that appear at least in one of the non-discarded co-occurrences. The second category describes the critiques which stem from the observed correlations between pairs of code critics (extracted from their co-occurrence as explained in Section 3).

4.1 Critiques on Individual Critics

Here we present our critiques on the individual class-level critics of Table 1.

Misleading name. Some code critics have misleading names and should be improved. For example, *'References an abstract class' (CC14)* is misleading. According to the name, a developer could assume that the code critic identifies a class B that is referencing an abstract class A. But in fact it detects the opposite, namely an abstract class A being referred to from somewhere within the analyzed application. A better name would thus be *'Abstract class being referenced'*. The name *'Instance variables not read AND written' (CC09)* is ill chosen too because, looking at how this code critic is implemented, it refers to instance variables which are EITHER read-only, write-only, OR not referenced

at all. A better name for this code critic could therefore be *'Instance variables not fully exploited'*.

Too general. Some critics are too general and could be split into several more specific critics. For example, the critic *'Instance variables not read AND written' (CC09)* mentioned above could be split into three different critics ('unreferenced instance variables', 'only written instance variables', 'only read instance variables'). The critics *'Overrides a special message' (CC13)* and *'Sends 'questionable' message' (CC16)* are about specific messages and could be split into separate critics for each of those messages. This would however lead to many individual critics, but they could be presented as a common group to the user, allowing him to inspect or ignore the details of the individual underlying critics, if he desires to do so.

Too tolerant. We also observed that, despite the fact that some critics seem meaningful and well-defined, they produce mostly false positives. This happens because there are often cases where it is acceptable not to adhere to some critics. However, when a critic produces mainly such false positives, we can wonder whether it is useful to keep the critic. Nevertheless, our results might be biased, since we analyzed only one rather well-designed framework (Moose).

An example of such a critic is *'Refers to class name instead of 'self class' ' (CC15)*, for which we discovered mostly acceptable deviations. For example, in Smalltalk it is quite common and acceptable in methods for checking equality to write `anObject isKindOf: X`, to verify that the type of `anObject` is indeed of a particular class `X` (and not some subclass). Similarly, the expression `self class == X` is often used to check if a given instance of this class is indeed of class X. Another case is when you write `X new`, because you want to be sure to create an instance of X and not of one of its subclasses. A last example is when you write an expression like `X allsubclasses` to refer to the root X of a relevant class hierarchy, and you want to manipulate the individual classes.

Many of the critics which are too tolerant could be refined further in order to avoid catching some of the false positives they produce. For example, if we consider CC15 again, we note that it often regards an expression like `isKindOf: X` used in a method implemented by class X as problematic, but in fact `isKindOf: self class` would be even more problematic, because it would get a different meaning in subclasses. This could be solved by making the critic take into account this case or any other of the above cases as known exceptions to the critic.

Too restrictive. Whereas some critics are too tolerant, others are too restrictive and could miss interesting cases. For example, *'Excessive inheritance depth' (CC5)* uses a threshold of 10 as depth level, but may miss other cases of excessive depth such as classes with inheritance depth 9. Obviously, there is no perfect threshold, but we found 20 additional classes with a depth of at least 9 (as compared to only 10 classes with a depth of at least 10) that should have been reported. We assume the threshold was set high in order to avoid producing too many results, making it harder for the user to process all reported results.

Redundant representation of results. Another source of noise in the results could be the amount of results produced by the critic, even if none of them are false positives. Sometimes, it would suffice to present the results differently to avoid such noise. For example, consider *'Excessive inheritance depth' (CC5)* again. Currently, it reports all leaf classes of hierarchies that suffer from the critic. But this generates many unnecessary results. It suffices to know the root of the hierarchy to start fixing the problem (and additionally, this could allow the user to lower the threshold so that the critic becomes less restrictive too).

Missing critics. Some important critics seem to be missing from the list of code critics. For example, there seem to be little or no critics related to inheritance issues [10], such as *local behavior* in a class with respect to its superclass or subclasses, or good *reuse of superclass behavior and state. Local behavior* identifies methods defined and used in the class that are not overridden in subclasses, often representing internal class behavior, and *Reuse of superclass behavior and state* identifies concrete methods that invoke superclass methods by self or super sends, not redefining behavior of the class. Code critics regarding inheritance could identify bad practices when implementing hierarchies.

Good critics. Whereas in this paper we focused mainly on negative critiques on code critics, we can remark that there are useful and well-designed code critics too. Our ultimate goal is to keep the good critics while identifying those that can be improved, in order to come up with a new and better-structured set of code critics. For example, *'Defines = but not hash' (CC04)* shows all classes that override `=` but not `hash`. If method `hash` is not overridden, then the instances of such classes cannot be used in sets. The implementation of `Set` assumes that equal elements have the same hash code. Another example is *'Method defined in all subclasses, but not in superclass' (CC10)* which detects classes defining a same method in all subclasses, but not as an abstract or default method in the superclass. This critic helps us find similar code that might be occurring in all the subclasses and that should be pulled up into the superclass.

4.2 Patterns of Co-occurring Critics

Now that we have described some critiques based on an analysis of individual code critics, we discuss some critiques derived from our analysis of the co-occurrence of pairs of code critics.

Redundant Critics. Critics are redundant when they detect the same problem. This happens for critics that come in two versions: one which just detects the problem and another one which detects it and at the same time proposes an automated refactoring to the problem. An example of this is *'detect:ifNone: `->` anySatisfy:' (MC01)* versus *'Uses detect:ifNone: instead of contains:' (MC09)*. Whereas MC01 offers an automated restructuring, in spite of its name MC09 only detects the problem. Although we did discover such cases in our experiment where we ran the critics directly, Pharo's Critic Browser would only use

one of these critics in order to avoid the user to get repeated results. Observe that the solution suggested by critic MC09 differs from the solution proposed by MC01, which can be confusing. Given that a same critic could have several possible refactorings, it would therefore be better to keep refactoring and detection strategies separated, and to have only one detection strategy per critic.

Indirect Correlation. This occurs when the results of two critics overlap significantly, without them having a common root cause. For instance, the following two correlations seem to occur essentially because one of the critics (*CC06*) generates so many results. They are *'Excessive number of methods' (CC06)* vs. *'Excessive number of variables' (CC07)*, and *'Sends 'questionable' message' (CC16)* vs. *'Excessive number of methods' (CC06)*.

Overlap Requires Splitting. A third pattern occurs when two critics produce overlapping results because they have a common root cause. It would be good to split such critics such that the common part becomes one separate critic and the non-overlapping parts become other critics. For instance, *'Instance variables not read AND written' (CC09)* is overlapping with *'Variables not referenced' (CC20)* because both critics detect unreferenced instance variables. While CC09 should be split as explained in section 4.1 (too general), CC20 could be split in a critic for class variables and one for instance variables. The critic for 'unreferenced instance variables' would then become a common subcritic for both CC09 and CC20.

Overlap Requires Merging. This pattern occurs when two code critics that regularly occur together could be combined into a single more specific critic. For instance, in the Smalltalk language, methods are grouped in method protocols representing the purpose of the method. Instance creation methods like `new`, for example, are put in the `'instance-creation'` protocol. The method-level critic *'Inconsistent method classification' (MC02)* is triggered when methods are wrongly classified and *'Unclassified methods' (MC08)* are reported when no protocol was assigned to a method. These critics coincide when an overridden method is unclassified whereas the method it overrides was classified. From the point of view of critic MC02, it is considered as an inconsistent classification since the classification of the parent and child method are different, whereas from the point of view of critic MC08 the child method is unclassified. Combining them in a new dedicated critic *'Inconsistently unclassified methods'* makes sense, because there is an easy refactoring that could be associated to this particular combination of critics, namely to classify the child method in the same protocol as the parent one. For cases where the critics do not overlap, the original critics MC02 and MC08 should still be reported.

Same niche. Sometimes, code critics seem to correlate just because they both refer to a specific kind of source entity. For example, the two independent critics on abstract classes *'References an abstract class' (CC14)* and *'Subclass responsibility not defined' (CC17)* often appear together, simply because they are the

only ones that both apply to abstract classes. (This pattern could be considered as a specific case of Indirect Correlation.)

Almost subset. This pattern occurs when most of the result set for one critic in practice always seems to be a subset of that for another critic. For example, the results for code critic *'Variable referenced in only one method and always assigned first' (CC19)* refers to the same variables reported by *'Instance variables not read AND written' (CC09)*. Indeed, if a variable is used only in one method and always assigned first *(CC19)*, it is likely that this variable will not be read in that same method (or any other method) and thus is reported by *CC09* too.

Ill-defined critic. Correlations between two critics may arise because one of them is ill-defined. If the ill-defined critic were fixed, the correlation would probably disappear. For example, *'Refers to classname instead of self class' (CC15)* correlates with *'Sends 'questionable' message' (CC16)*, because CC15 often gives false positives related to the use of `isKindOf:`, which is also one of the questionable messages. If we would fix CC15 to avoid those false positives, this correlation would likely disappear.

Noisy correlation. This pattern describes critics that seem to be correlated to many other critics and therefore produce too much noise. They could better be removed if the overlap with another critic is not strong (likely to be only accidental matches). For example, *'Excessive number of Methods' (CC6)* has this problem, because the more methods a class has, the higher the chance that the class suffers from other critics as well.

High-level critics. Whereas in this section we analyzed the co-occurrence of critics mainly by focusing on their shortcomings, in forthcoming research we will analyze the results more in-depth and will also identify good, desired or expected correlations between critics. For example, the correlation between *'Utility methods' (MC10)* and *'Law of Demeter' (MC03)* is not unexpected as it may indicate an imperative (non object-oriented) programming style.

5 Discussion, Conclusion and Future Work

This paper presented our initial results of an analysis of low-level code critics detected on the Moose system, a large and mature Smalltalk application. The results of this analysis can help us identify which low-level critics could benefit from redefinition or refactoring so that they would provide more accurate or meaningful results, as well as how to combine them into more *high-level critics* to improve the recommendations they provide.

As future work, we plan to provide a more in-depth analysis, including a deeper analysis of the method-level critics, and propose concrete improvements, combinations and refactorings of the existing code critics. This analysis could then be repeated iteratively, to further improve the improved critics, again by

analyzing their correlations, until we eventually reach a stable group of proposed critics.

Finally, although in this paper we focused on Pharo Smalltalk's code critics only, we believe the ideas and approach presented in this paper to be easily generalizable to other code checking tools and programming languages. To confirm this, we have started to analyze other code checking tools for similar correlations and improvements: CheckStyle [2], PMD [7] and FindBugs [4, 11] for Java, Splint [9] or Cppcheck [3] for C, Pylint [8] for Python, FxCop for .NET, PHP Mess Detector [6] for PHP and Android Lint [1] for Android programming. For each of these tools, we performed an initial analysis on a single application. We observed that, in spite of the fact that some of these tools focus on checks that are quite different from Pharo's code critics, our approach could still be used to analyze those tools. Whereas for most tools we indeed found many examples similar to the critique patterns mentioned in this paper, for some tools we discovered only very few correlations. This could be due to the particular applications that were analyzed (indeed, in our analysis of the 51 packages of Moose too, there were a few packages that did not have many critics). Or it could suggest that, while the approach remains applicable, it may be less relevant for some of the tools we analyzed. This may for example be the case for tools that are already quite mature and offer a stable and orthogonal set of checks. More experiments are needed to confirm this. This may be the topic of a forthcoming paper.

References

1. Androidlint. `http://tools.android.com/tips/lint`. Accessed: 2015-03-30.
2. Checkstyle. `http://checkstyle.sourceforge.net`. Accessed: 2015-03-30.
3. Cppcheck. `http://cppcheck.sourceforge.net/`. Accessed: 2015-03-30.
4. Findbugs. `http://findbugs.sourceforge.net`. Accessed: 2015-03-30.
5. MOOSE. `http://www.moosetechnology.org/`. Accessed: 2015-03-30.
6. Phpmd. `http://phpmd.org/`. Accessed: 2015-03-30.
7. PMD. `http://pmd.sourceforge.net/`. Accessed: 2015-03-30.
8. Pylint. `http://www.pylint.org/`. Accessed: 2015-03-30.
9. Splint. `http://www.splint.org/`. Accessed: 2015-03-30.
10. G. Arévalo, S. Ducasse, S. Gordillo, and O. Nierstrasz. Generating a catalog of unanticipated schemas in class hierarchies using formal concept analysis. *Inf. Softw. Technol.*, 52(11):1167–1187, Nov. 2010.
11. D. Hovemeyer and W. Pugh. Finding bugs is easy. In *Companion to the 19th Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications, OOPSLA 2004*, pages 132–136. ACM, 2004.
12. 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.
13. 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.

User interface level testing with TESTAR; what about more sophisticated action specification and selection?

Sebastian Bauersfeld and Tanja E. J. Vos

Research Center on Software Production Methods (PROS)
Universitat Politècnica de València
Valencia, Spain

Abstract. Testing software applications at the Graphical User Interface (GUI) level is a very important testing phase to ensure realistic tests because the GUI represents a central juncture in the application under test from where all the functionality is accessed. In earlier works we presented the TESTAR tool, a Model-Based approach to automate testing of applications at the GUI level whose objective is to generate test cases based on a model that is automatically derived from the GUI through the accessibility API. Once the model has been created, TESTAR derives the sets of visible and unblocked actions that are possible for each state that the GUI is in and randomly selects and executes actions in order to drive the tests. This paper, instead of random selection, we propose a more advanced action specification and selection mechanism developed on top of our test framework *TESTAR*. Instead of selecting random clicks and keystrokes that are visible and unblocked in a certain state, the tool uses a Prolog specification to derive sensible and sophisticated actions. In addition, it employs a well-known machine learning algorithm, called Q-Learning, in order to systematically explore even large and complex GUIs. This paper explains how it operates and present the results of experiments with a set of popular desktop applications.

1 Introduction

Graphical User Interfaces (GUIs) represent the main connection point between a software's components and its end users and can be found in almost all modern applications. This makes them attractive for testers, since testing at the GUI level means testing from the user's perspective and is thus the ultimate way of verifying a program's correct behaviour. Current GUIs can account for 45-60% of the entire source code [14] and are often large and complex. Consequently, it is difficult to test applications thoroughly through their GUI, especially because GUIs are designed to be operated by humans, not machines. Moreover, they are inherently non-static interfaces, subject to constant change caused by functionality updates, usability enhancements, changing requirements or altered contexts. This makes it very hard to develop and maintain test cases without resorting to time-consuming and expensive manual testing.

In previous work, we have presented TESTAR [5,6,3], a Model-Based approach to automate testing at the GUI level. TESTAR uses the operating system's Accessibility API to recognize GUI controls and their properties and enables programmatic interaction with them. It derives sets of possible actions for each state that the GUI is in (i.e. the visible widgets, their size, location and other properties such as whether they are enabled or blocked by other windows etc.) and randomly selects and executes appropriate ones in order to drive the tests. In completely autonomous and unattended mode, the oracles can detect faulty behaviour when a system crashes or freezes. Besides these free oracles, the tester can easily specify some regular expressions that can detect patterns of suspicious titles in widgets that might pop up during the executed tests sequences. For more sophisticated and powerful oracles, the tester can program the Java protocol that is used to evaluate the outcomes of the tests.

The strength of the approach is that the technique does not modify nor require the SUT's source code, which makes it applicable to a wide range of programs. With a proper setup and a powerful oracle, TESTAR can operate completely unattended, which saves human effort and consequently testing costs. We believe that TESTAR is a straightforward and effective technique of provoking crashes and reported on its success describing experiments done with MS Word in [5]. We were able to find 14 crash sequences while running TESTAR during 48 hours[1] applying a strategy of random selection of visible/unblocked actions.

In this paper we will investigate more sophisticated ways of action specification and selection. Instead of clicking randomly on visible and unblocked locations within the GUI, we enable the tester to define the set of visible actions that he or she wants to execute. The definitions are written in Prolog syntax and allow the specification of thousands of actions – even complex mouse gestures – with only a few lines of code. Moreover, we will use a machine learning algorithm called Q-Learning [4] to explore the GUI in a more systematic manner than random testing. It learns about the interface and strives to find previously unexecuted actions in order to operate even deeply nested dialogs, which a random algorithm is unlikely to discover. In the next section we will explain how TESTAR obtains the GUI state and executes actions.

This paper is structured as follows. Section 2 presents the TESTAR approach for testing at the GUI level and describes the extensions for action specification and selection. Section 3 presents the results of a first experiment in which we applied TESTAR to a set of popular applications to test its ability in finding reproducible faults. Section 4 lists related work, section 5 reviews the approach and section 6 describes future work.

2 The TESTAR Approach

Figure 1 shows how TESTAR would operate on MS Word. At each step of a sequence, TESTAR (A) determines the state of the GUI, i.e. the visible widgets,

[1] Videos of these crashes are available at `http://www.youtube.com/watch?v=PBs9jF_pLCs`

their size, location and other properties (such as whether they are enabled or blocked by other windows etc.). From that state it (B) derives a *set of feasible actions* (the green dots, letters and arrows, which represent clicks, text input and drag and drop operations, respectively) from which it then (C) selects one (marked red) and finally executes it. By repeating these steps, TESTAR will be able to generate arbitrary input sequences to drive the GUI. The following subsections will explain this in more details, together with the new more sophisticated ways of deriving actions and selection them for execution.

2.1 Determine the GUI State

All of our experiments described in this paper are conducted on MacOSX. For this platform the Accessibility API – which simplifies computer usage for people with disabilities – is used to obtain the SUT's GUI state. It allows to gather information about the visible widgets of an application and gives TESTAR the means to query their property values. Since it is a native API written in *ObjectiveC*, we make use of the *Java Native Interface* (JNI) to invoke its methods. After querying the application's GUI state, we save the obtained information in a so-called *widget tree* which captures the structure of the GUI. Figure 2 displays an example of such a tree. Each node corresponds to a visible widget and contains information about its type, position, size, title and indicates whether it is enabled, etc. The Accessibility API gives access to over 160 attributes which allows us to retrieve detailed information such as:

- The **type** of a widget.
- The **position** and **size** which describe a widget's rectangle (necessary for clicks and other mouse gestures).
- It tells us whether a widget is **enabled** (It does not make sense to click disabled widgets).
- Whether a widget is **blocked**. This property is not provided by the API but we calculate it. For example, if a message box blocks all other widgets behind it, then TESTAR can detect those and other modal dialogs (like menus) and sets the blocked attribute accordingly.
- Whether a widget is **focused** (has keyboard focus) so that TESTAR knows when it can type into text fields.
- Attributes such as **title, help** and other descriptive attributes are very important to distinguish widgets from each other and give them an identity. We will make use of this in the next subsection when we describe our algorithm for action selection.

This gives us access to almost all widgets of an application, if they are not custom-coded or drawn onto the window. We found that the Accessibility API works very well with the majority of native applications (since the API works for all standard widgets, the developers do not have to explicitly code for it to work).

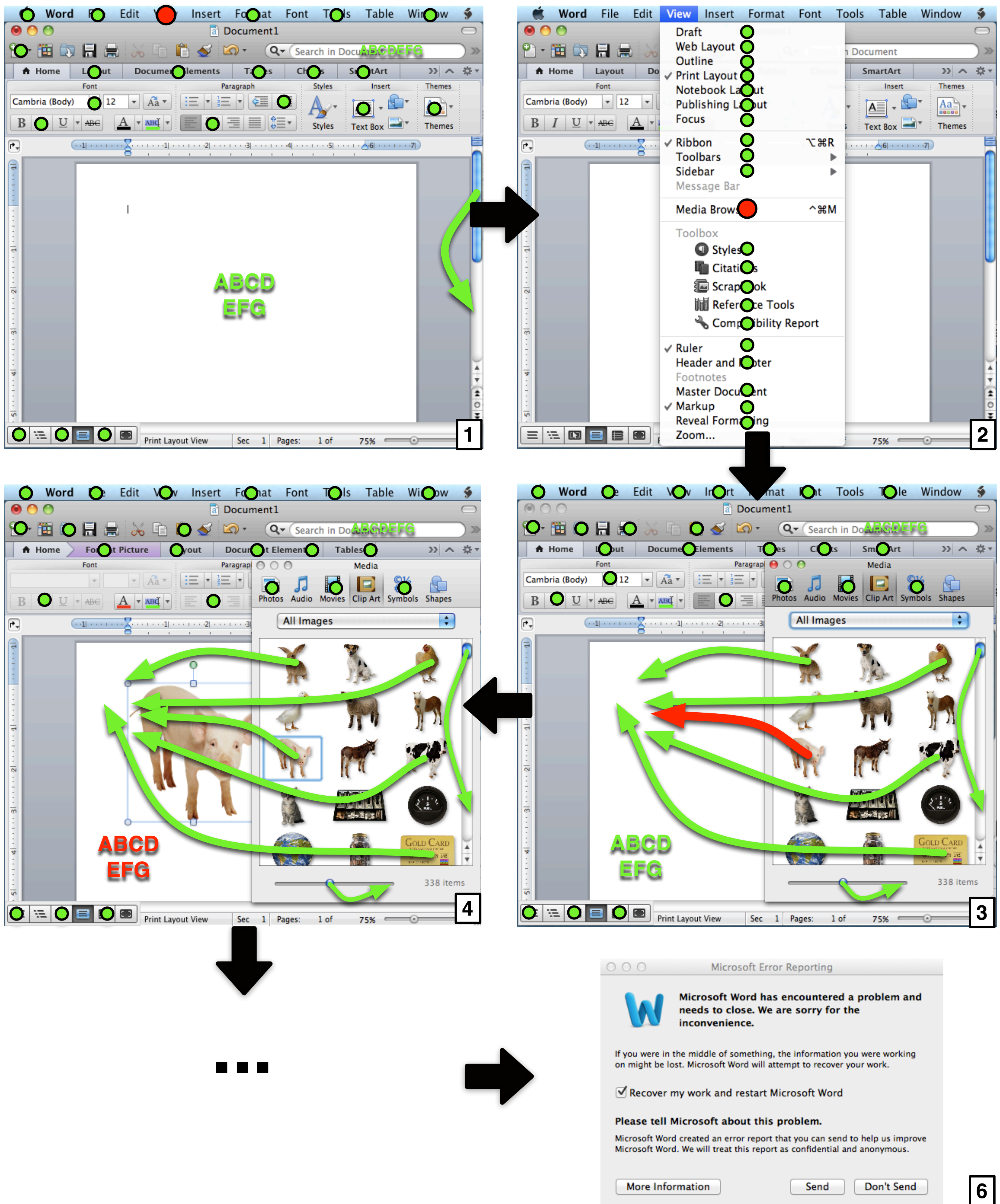


Fig. 1. Sequence generation by iteratively selecting from the set of currently available actions. The ultimate goal is to crash the SUT. (In order to preserve clarity the graphic does not display all possible actions.)

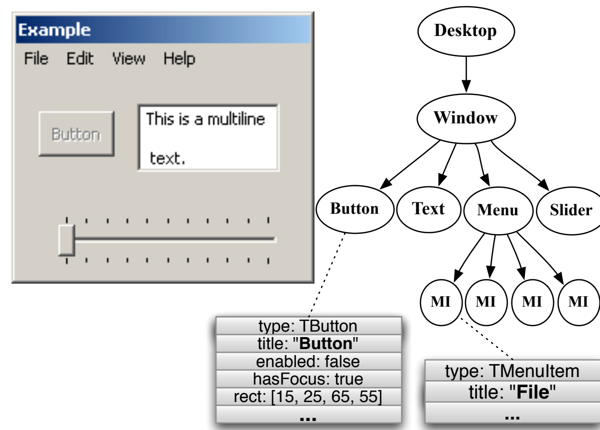


Fig. 2. The state of a GUI can be described as a widget tree which captures property values for each control.

2.2 Derive Actions

Having obtained the GUI's current state, we can go on to derive a set of actions. One thing to keep in mind is that the more actions TESTAR can choose from, the bigger the sequence space will be and the more time the search for faults will consume. Ideally, TESTAR should only select from a small set of actions which are likely to expose faults. Thus, our intention is to keep the search space as small as possible and as large as necessary for finding faults. For each state we strive to generate a set of *sensible* actions which should be appropriate to the widgets that they are executed on: Buttons should be clicked, scrollbars should be dragged and text boxes should be filled with text. Furthermore, we would like to exercise only those widgets which are *enabled* and not *blocked*. For example: it would not make sense to click on any widget that is blocked by a message box. Since the box blocks the input, it is unlikely that any event handling code (with potential faults) will be invoked.

One way to tell TESTAR how to use specific widgets would be to provide a set of rules so that it can generate actions according to a widget's type, position and other attribute values. This would work reasonably well for many GUIs, since most widgets are standard controls. However, it is not very flexible. There might be non-standard widgets which TESTAR does not know how to use or which are used in a different way than they were designed for (Microsoft Word uses static text labels as table items and has lots of custom coded widgets). Moreover, on screen 3 of Figure 1 how would TESTAR know that it can drag images from the media browser into the document or that it can draw onto the canvas in Figure 4? Finally, a tester might want to execute only specific parts of the application, e.g. click only buttons and menu items or leave out certain actions which could trigger "hazardous" operations that delete or move files.

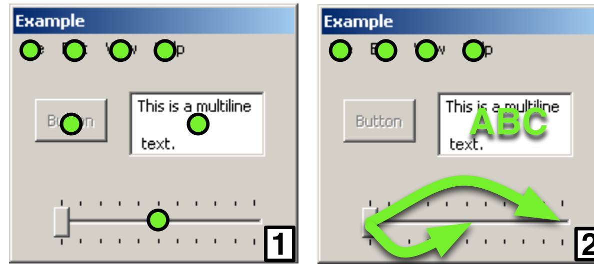
Due to these reasons, we let the tester specify which actions TESTAR should execute. Since modern GUIs can be very large and complex, with potentially ten thousands of widgets, the tester needs a comfortable and efficient way of defining actions. We found that action specification in Java is often verbose and not very concise. Definitions such as "Drag every image within the window titled 'Images' onto the canvas with help text 'Document'." often require many lines of code. Therefore, we were looking for a more declarative language, which allows the tester to specify actions in a more natural way. Eventually, we decided to integrate a *Prolog* engine into our framework. Prolog is a programming language that has its roots in first-order-logic and is often associated with the artificial intelligence and computational linguistics communities [9]. In Prolog the programmer writes a database with facts and rules such as

```
parent(bruce, sarah).
parent(sarah, tom).
parent(gerard, tom).
ancestor(X,Y):- parent(X,Y);
               (parent(X,Z), ancestor(Z,Y)).
```

where the first three lines state that Bruce is Sarah's parent and that Sarah and Gerard are Tom's parents. The third line is a recursive rule and reads: "X is Y's ancestor, if X is the parent of Y or there exists a Z so that X is a parent of Z and Z is an ancestor of Y" (the semicolon and comma represent disjunction and conjunction, respectively). The programmer can now issue queries such as `?- ancestor(X,tom)` ("Who are Tom's ancestors?") against this database. Prolog will then apply the facts and rules to find the answers to this question, i.e. all possible substitutions for `X` (Sarah, Gerard, Bruce). This can be applied to much more complex facts and hierarchies and Prolog is known to be an efficient and concise language for those kind of relationship problems [9].

In our case we reason over widgets that also form a hierarchy. However, we do not have an explicitly written fact database. Instead, the widget tree acts as this database, as it describes the relationships among the widgets and contains their property values. The tester can then use the Prolog engine to define actions for the current GUI state. Figure 3.1 shows an example of how this is done. Let us assume we want to generate clicks for all widgets within our example dialog. The corresponding Prolog query is listed under the image and reads: "For all widgets W, which have an ancestor A, whose title is 'Example', calculate the center coordinates X and Y and issue a left click". Since this also generates clicks for the disabled button widget, the text box and the slider, we might want to improve this code, in order to obtain a more appropriate action set: Figure 3.2 shows the adapted Prolog code and its effects. We implemented predicates such as `widget(W)`, `enabled(W)`, `type(W, R)` and many others to allow the tester to reason over the widget tree.

Besides traditional clicks, text typing and drag and drop operations, the Prolog engine also facilitates the definition of more complex mouse gestures such as the ones depicted in Figure 4. The tester simply defines a set of points that the cursor is supposed to pass through and TESTAR then uses cubic splines to




```
1) widget(W),ancestor(A,W),title(A,"Example"),center(W,X,Y),lclick(X,Y).

2) widget(W),ancestor(A,W),title(A,"Example"),enabled(W),type(W,T),(
     ((T='Button'; T='MenuItem'), center(W, X, Y), lclick(X, Y));
     (T='Text', center(W, X, Y), click(X, Y), type('ABC'));
     (T='Slider', center(W, 'Thumb', X1, Y1),
        rect(W, Left, Top, Width, Height), Y2 is Top + Height / 2,
        (X2 is Left + Width / 2; X2 is Left + Width), drag(X1,Y1,X2,Y2))
   ).
```

Fig. 3. Action specification with Prolog queries.

calculate the cursor's trajectory. This allows for complex drawing gestures and handwriting.

The ability to define the set of actions makes it possible to customize TESTAR test and to restrict the search space to the actions that the tester thinks are likely to trigger faults. As our experiments will show, a reasonably sized set of "interesting" actions, can be more effective in finding faults than unconstrained mouse and keyboard inputs. The Prolog engine and the fact that GUIs usually consist of a limited set of widget types with similar functionality, allow the tester to specify thousands of these actions with only a few lines of code. Thus, even large and complex applications can be configured quickly.

For our Prolog engine we used a modified version of the *PrologCafe*[2] implementation, which compiles and transforms Prolog into Java code.

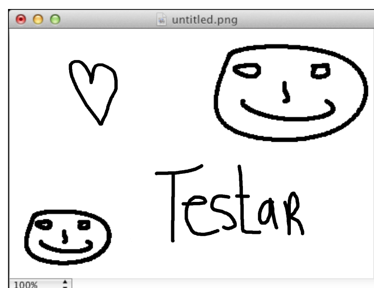


Fig. 4. TESTAR is capable of executing complex mouse gestures.

[2] http://code.google.com/p/prolog-cafe

2.3 Action Selection

The action selection strategy is a crucial feature within TESTAR. The right actions can improve the likelihood and decrease the time necessary for triggering crashes. Consequently, we want it to learn about the GUI and be able to explore it thoroughly. In large SUTs with many – potentially deeply nested – dialogs and actions, it is unlikely that a random algorithm will sufficiently exercise most parts of the GUI within a reasonable amount of time. Certain actions are easier to access and will therefore be executed more often, while others might not be executed at all.

Ideally, TESTAR should exercise all parts of the GUI equally and execute each possible action at least once. Instead of pursuing a systematic exploration like Memon et al. [13], which use a depth first search approach, we would like to have a mixture between a random and systematic approach that still allows each possible sequence to be generated. Thus, our idea is to change the probability distribution over the sequence space, so that seldom executed actions will be selected with a higher likelihood than others, in order to favor exploration of the GUI. To achieve this, a straightforward greedy approach would be to select at each state the action which has been executed the least number of times. Unfortunately, this might not yield the expected results: In a GUI it is often necessary to first execute certain actions in order to reach others. Hence, these need to be executed more often, which requires an algorithm that can "look ahead". This brought us to consider a reinforcement learning technique called *Q-Learning*. We will now explain how it works, define the environment that it operates in and specify the reward that it tries to maximize.

We assume that our SUT can be modelled as a finite *Markov Decision Process* (MDP). A finite MDP is a discrete time stochastic control process in an environment with a finite set of states S and a finite set of actions A [16]. During each time step t the environment remains in a state $s = s_t$ and a decision maker, called *agent*, executes an action $a = a_t \in A_s \subseteq A$ from the set of available actions, which causes a transition to state $s' = s_{t+1}$. In our case, the agent is TESTAR, the set A will refer to the possible GUI actions and S will be the set of observable GUI states.

An MDP's state transition probabilities are governed by

$$P(s, a, s') = Pr\{s_{t+1} = s' | s_t = s, a_t = a\}$$

meaning, that the likelihood of arriving in state s' exclusively depends on a and s and not on any previous actions or states. This is called *Markov Property* which we assume holds approximately for the SUT[3]. In an MDP, the agent receives rewards $R(s, a, s')$ after each transition, so that it can learn to distinguish good from bad decisions. Since we want to favor exploration, we set the rewards as

[3] Since we can only observe the GUI states and not the SUT's true internal states (*Hidden Markov Model*), one might argue whether the Markov Property holds sufficiently. However, we assume that the algorithm will still perform reasonably well.

follows:

$$R(s,a,s') := \begin{cases} r_{init} & \text{, if } x_a = 0 \\ \frac{1}{x_a} & \text{, else} \end{cases}$$

Where x_a is the amount of times that action a has been executed and r_{init} is a large positive number (we want actions that have never been executed before, to be extraordinarily attractive). Hence, the more often an action has been executed, the less desirable it will be for the agent. The ultimate goal is to learn a policy π which maximizes the agent's expected reward. The policy determines for each state $s \in S$ which action $a \in A_s$ should be executed. We will apply the Q-Learning algorithm in order to find π. Instead of computing it directly, Q-Learning first calculates a *value function* $V(s,a)$ which assigns a numeric quality value – the expected future reward – to each state action pair $(s,a) \in S \times A$. This function is essential, since it allows the agent to look ahead when making decisions: The agent then simply selects $a^* = argmax_a\{V(s,a)|a \in A_s\}$ within each state $s \in S$.

Algorithm 1 shows the pseudo-code for our approach. Since it does not have any prior knowledge about the GUI, the agent starts off completely uninformed. Step by step it discovers states and actions and learns the value function through the rewards it obtains. The quality value for each new state action pair is initialized to r_{init}. The heart of the algorithm is the value update in line 9: The updated quality value of the executed state action pair is the sum of the received reward plus the maximum value of all subsequent state action pairs multiplied by the discount factor γ. The more γ approaches zero, the more opportunistic and greedy the agent becomes (it considers only immediate rewards). When γ approaches 1 it will opt for long term reward. It is worth mentioning that the value function and consequently the policy will constantly vary, due to the fact that the rewards change. This is what we want, since the agent should always put emphasis on the regions of the SUT which it has visited the least amount of times[4].

Instead of always selecting the best action (line 6), one might also consider a random selection proportional to the values of the available state action pairs. This would introduce more randomness in the sequence generation process. For our experiments, however, we sticked to the presented version of the algorithm.

Representation of States and Actions In order to be able to apply the above algorithm, we have to assign a unique and stable identifier to each state and action, so that we are able to recognize them. For this, we can use the structure and the property values within the widget tree. For example: To create a unique identifier for a click on a button we can use a combination of the button's property values, such as its title, help text or its position in the widget hierarchy (parents / children / siblings). The properties selected for the identifier should be

[4] Again, this is contrary to the Markov Property where rewards are supposed to be stationary. We counter the problem of constantly changing rewards with frequent value updates, i.e. we sweep more often over the state actions pairs within the generated sequence (i.e. line 9 of the algorithm).

```
    Input: r_init                              /* reward for unexecuted actions */
    Input: 0 < γ < 1                                          /* discount factor */
 1  begin
 2  |  start SUT
 3  |  V(s,a) ← r_init     ∀(s,a) ∈ S × A
 4  |  repeat
 5  |  |  obtain current state s and available actions A_s
 6  |  |  a* ← argmax_a{V(s,a)|a ∈ A_s}
 7  |  |  execute a*
 8  |  |  obtain state s' and available actions A_s'
 9  |  |  V(s,a*) ← R(s,a*,s') + γ · max_{a∈A_s'} V(s',a)
10  |  until stopping criteria met
11  |  stop SUT
12  end
```

Algorithm 1: Sequence generation with Q-Learning [4]

relatively "stable": The title of a window, for example, is quite often not a stable value (opening new documents in a text editor will change the title of the main window) whereas its help text is less likely to change. The same approach can be applied to represent GUI states: One simply combines the values of all stable properties of all widgets on the screen. Since this might be a lot of information, we will only save a hash value generated from these values[5]. This way we can assign a unique and stable number to each state and action.

3 Evaluation

We performed two different experiments. In the first one, we compared three different approaches with TESTAR summarized in Table 1 to find out: ***RQ1*** *- which of these testing approaches needs the least amount of time to crash applications?*.

In the second experiment we used only approach C. At first we had it generate a set of short sequences – with at most 200 actions – with the goal to trigger crashes for each SUT. We then tried to replay these crashing sequences to determine the reproducibility of the revealed faults and find out: ***RQ2****: What fraction of the generated crash sequences are reproducible, i.e. trigger the crash again upon replay?*

3.1 The variables: What is being measured?

In the first experiment we measured the average time it took each approach to crash an application, which is equivalent to what Liu et al. [12] do in their experiments. We considered an SUT to have crashed if a) it unexpectedly terminated or b) it did not respond to inputs during more than 60 seconds.

[5] Of course this could lead to collisions. However, for the sake of simplicity we assume that this is unlikely and does not significantly affect the optimization process.

Table 1. The approaches of TESTAR that were compared.

Approach	Description
A	TESTAR's default configuration which randomly selects actions, but is informed in the sense that it uses the Accessibility API to recognize the visible/unblocked actions.
B	TESTAR with random selection of actions specified by the tester using Prolog as described in Section 2.2. This approach also randomly selects actions, but it is "informed", in the sense that it makes use of Prolog action specifications tailored to each SUT.
C	This approach is equivalent to B, but uses the Q-learning algorithm instead of random selection.

In the second experiment we measured the absolute number of crashes that we found and the percentage of the crashing sequences that we were able to reproduce.

3.2 The Systems Under Test (SUTs)

We applied the three approaches to a set of popular MacOSX applications as listed in the first column of Table 2. We tried to include different types of SUTs such as office applications (Word, Excel, Mail, iCal ...) an instant messenger (Skype), a music player (iTunes) and a drawing application (Paintbrush), to find out whether our approach is generally applicable.

3.3 The protocol

We carried out all of our experiments on a 2.7 GHz Intel Core i7 MacBook with 8GB RAM and MacOSX 10.7.4. To verify and analyze the found crashes, we used a *frame grabber* which recorded the screen content of our test machine during the entire process. Thus, many of the found crashing sequences can be viewed on our web page[6]. Before running any tests, we prepared each application by performing the following tasks:

– Write scripts which restore configuration files and application data before each run: This is a crucial step, since we compared different approaches against each other and wanted to make sure that each one starts the application from the same configuration. Most applications save their settings in configuration files, which needed to be restored to their defaults after each run. In addition, applications like *Mail* or *Skype* employ user profile

[6] http://www.pros.upv.es/testar

Table 2. Competition between approaches A, B and C..

Application	**Avg. time T (minutes) to crash (3 runs per SUT and approach)**		
	A	**B**	**C**
Excel 2011 v14.1.4	20.95	9.24	15.06
iTunes v10.6.1	1072.55*	53.86	49.23
PowerPoint 2011 v14.1.4	21.18	9.77	8.61
Skype v5.8.0.945	1440*	144.26	130.79
iCal v5.0.3	1099.81*	103.82	146.53
Calculator v10.7.1	62.93	18.75	19.93
Word 2011 v14.1.4	50.33	14.41	12.56
Mail v5.2	1276.71*	134.02	122.31
Paintbrush v2.0.0	1440*	239.82	234.04
Overall Average	720.5	80.89	82.12

data such as mailboxes or contact lists. During a run, mails and contacts might be deleted, added, renamed or moved. Thus, each approach had a list of directories and files which they automatically restored upon application start.

– Setup a secure execution environment: GUI testing should be done with caution. Unless they are told otherwise, the test tools will execute every possible action. Thus, they might print out documents, add / rename / move / delete files, printers and fonts, install new software or even shut down the entire machine. We experienced all of these situations and first tried to counter them by disallowing actions for items such as “Print”, “Open”, “Save As”, “Restart”, etc. However, applications such as Microsoft Word are large and allow many potentially hazardous operations which are difficult to anticipate. Therefore, we decided to run our tests in a sandbox (a MacOSX program called *sandbox-exec*) with a fine-grained access control model. This allowed us to restrict read and write access to certain directories, block specific system calls and restrict internet access. The latter was necessary when we were testing *Skype* and *Mail*. Since we employed some of our own contact lists and mail accounts, we could have contacted people and might have transmitted private data.
– For approaches B and C, we moreover defined a set of *sensible* actions: As described in Section 2.A and 2.B, we took care to specify actions appropriate to the specific widgets: We generated clicks on buttons and menu items as well as right-clicks, input for text boxes, drag operations for scrollbars, sliders and other draggable items. For one of the tested applications (*Paintbrush*, a Microsoft Paint clone) we also generated mouse gestures to draw figures as shown in Figure 4. For each application we strived to define a rich set of actions, comprising the ones that a human user would apply when working

with the program. The LOCs of the Prolog specifications can be found in Table 3.

Table 3. Size of Prolog action specifications in Lines Of Code (LOC).

Application	LOC
Excel 2011 v14.1.4	29
iTunes v10.6.1	25
PowerPoint 2011 v14.1.4	29
Skype v5.8.0.945	30
iCal v5.0.3	28
Calculator v10.7.1	11
Word 2011 v14.1.4	29
Mail v5.2	30
Paintbrush v2.0.0	39

For each approach we set the delay between two consecutive actions to 50 ms, to give the GUI some time to respond. During the entire testing process, the SUTs were forced into the foreground, so that even if they started external processes, these did not block access to the GUI.

In the first experiment we applied all three approaches to the applications in Table 2, let each one run until it caused a crash and measured the elapsed time. This process was repeated three times for each application, yielding 81 runs in total and 27 for each approach. Since we had only limited time for the evaluation, we stopped runs that failed to trigger crashes within 24 hours and took 24 hours as the time. This only happened with approach A and we marked the corresponding cells in Table 2 with "*". We used our video records to examine each crash and to determine whether an SUT was really frozen or did only perform some heavy processing, such as importing music or mailboxes in the case of iTunes and Mail.

In the second experiment we had approach C exercise each of the object SUTs again. This time we limited the amount of actions that were generated to 200, since we strived to generate short and comprehensible crashing sequences. If the SUT did not crash after 200 actions, it was restarted. We run C for 10 hours on each application in order to generate several short crashing sequences. After that, we replayed each of the crashing sequences 5 times. We considered the crash to be reproducible if during one of these playbacks the application crashed after the same action as during the initial crash.

3.4 The results

Table 2 lists our findings for the first experiment. It shows that approach B and C were able to trigger crashes for all applications and needed significantly less

time than the default TESTAR (we performed a t-test with significance level $\alpha = 0.05$). These findings are consistent with [8], where the authors carried out their experiments on Motorola cell phones. Consequently, during the experiment, with advanced TESTAR action specification in Prolog, approaches B and C were found to be faster in finding crashes than the default setting (A). The additional effort of writing the Prolog specifications was relatively small for us. As can be found in Table 3, most of the specifications consisted of less than 30 LOCs. Only the specification for Paintbrush is slightly more complex, since we added a few mouse gestures to draw figures into the drawing area. A future additional study should be done with real testers to see if the learning curve is not to steep.

Unfortunately, approach C is not significantly faster in crashing the application than approach B, so we do not have a clear outcome about our Q-learning approach for action selection. Each of the two algorithms seems to perform better for specific applications. We will need to investigate on the cause of this outcome in future work since a quick analysis already found that approach C on the average executes about 2.5 times as many different actions as B, as we expected.

Table 4 shows the results of the second experiment. We were able to reproduce 21 out of 33 triggered crashes, which is more than 60% and yields the answer to RQ2. Some of the reproducible crashes we found were indeterministic so that during some playbacks the application did not crash, whereas during others it did. This might be caused by the fact that the execution environment during sequence recording and sequence replaying is not always identical. Certain factors, which might have been crucial for the application to crash, are not entirely under our control. Such factors are the CPU load, memory consumption, thread scheduling, etc. For future research we consider the use of a virtual machine which might further improve the reproducibility, because it would allow to guarantee the same environmental conditions during recording and replaying.

Table 4. Reproducibility of crashes.

Application	**Crashes**	**Reproducible**
Skype v5.8.0.945	2	1
Word 2011 v14.1.4	8	5
Calculator v10.7.1	4	4
iTunes v10.6.1	2	1
iCal v5.0.3	2	2
PowerPoint 2011 v14.1.4	4	2
Mail v5.2	1	0
Excel 2011 v14.1.4	9	5
Paintbrush v2.0.0	1	1
Total	33	21
Percentage	-	**63.64%**

3.5 Threats to Validity

Some of the crashes that were generated might have been caused by the same faults. Unfortunately, we could not verify this, since we did not have access to the source code. To determine the general reproducibility in RQ2, one would need a set of sequences which trigger crashes which are known to be caused by different faults of different types.

The capabilities of approaches B and C depend on the Prolog specifications and thus on the tester's skills in defining a set of fault sensitive actions. In addition, the experiments in this paper were executed by the researcher that developed the specification functionality and so it could be expected that specifications of a person less familiar with these facilities, and hence the additional effort, could be larger.

Finally, we executed our experiments on a single platform. This might not be representative for other desktop platforms such as Windows or even mobile operating systems like Android and iOS.

4 Related Work

Amalfitano et al. [1] perform crash-testing on Android mobile apps. Before testing the application, they generate a model in the form of a *GUI tree*, whose nodes represent the app's different screens and whose transitions refer to event handlers fired on widgets within these screens. The model is obtained by a so-called GUI-Crawler which walks through the application by invoking the available event-handlers with random argument values. From the GUI-tree they obtain test cases, by selecting paths starting from the root to one of the leaves. They automatically instrument the application to detect uncaught exceptions which would crash the app. They test their approach on a small calculator application.

Liu et al. [12] employ Adaptive Random Testing (ART) to crash Android mobile applications or render them unresponsive. Their algorithm tries to generate very diverse test cases, which are different from the already executed ones. They define distance measures for input sequences and strive to generate new test cases which have a high distance to the ones which are already in the test pool. They test their approach on six small applications among which an address book and an SMS client. In addition to normal user input, like keystrokes, clicks and scrolling, they also simulate environmental events like the change of GPS coordinates, network input, humidity or phone usage.

Hofer et al. [10] apply a smart testing monkey to the Microsoft Windows calculator application. Before the testing process, they manually build an abstract model of the GUI relevant behavior using a language that is based on finite state machines and borrows elements from UML and state charts. The resulting *Decision Based State Machine* acts as both, an orientation for walking through the GUI by randomly selecting event transitions and as test oracle which checks certain properties for each state. They use the Ranorex Automation Framework to execute their test cases.

Artzi et al. [2] perform feedback-directed random test case generation for JavaScript web applications. Their objectives are to find test suites with high code coverage as well as sequences that exhibit programming errors, such as invalid-html or runtime exceptions. They developed a framework called *Artemis*, which triggers events by calling the appropriate handler methods and supplying them with the necessary arguments. To direct their search, they use prioritization functions: They select event handlers at random, but prefer the ones for which they have achieved only low branch coverage during previous sequences.

Huang et al. [11] concentrate on functional GUI testing. Their idea is to walk through the GUI (by systematically clicking on widgets) and to automatically generate a model (in the form of an *Event Flow Graph*) from which they derive test cases by applying several coverage criteria. In their experiments they test Java applications (some of them are synthetic, some are part of an office suite developed by students) which they execute by performing clicks. Sometimes they have problems with the execution of their sequences, since the GUI model they are derived from is an approximation. Thus, they repair their sequences by applying a genetic algorithm which strives to restore the coverage of the initial test suite. Their techniques are available as part of the *GUITAR* framework[7].

Miller et al. [15] conducted an interesting empirical study on the efficiency of dumb monkey testing for 30 MacOS GUI applications. They developed a monkey that issues random (double) clicks, keystrokes and drag operations on the frontmost application. They even report on the causes of the crashes for the few applications that they have the source code for. Unfortunately, they do not mention how long they were running the tests for in order for the programs to hang or crash.

Among the tools for dumb monkey testing there are: *anteater*[8] which performs crash testing on the iPhone by issuing random clicks for any view (screen) that it finds. *UI/Application Exerciser Monkey*[9] is a similar program for Android and generates random streams of clicks, touches or gestures, as well as a number of system-level events. Other monkey tools are *Powerfuzzer* (an HTTP based web fuzzer written in Python), *GUI Tester*[10] (a desktop monkey for Windows which uses a taboo-list to avoid cycling on the same events) and *MonkeyFuzz*[11] (Windows desktop monkey developed in C#).

5 Conclusion

In this paper we presented TESTAR, a tool for automated testing at the GUI level, together with some new approaches for action selection and specification. Moreover we presented the results of evaluating these new approaches for crash

[7] http://sourceforge.net/projects/guitar
[8] http://www.redant.com/anteater
[9] http://developer.android.com/tools/help/monkey.html
[10] http://www.poderico.it/guitester/index.html
[11] http://monkeyfuzz.codeplex.com

testing of a set of real-world applications with complex GUIs. The strengths of our tool are:

- Easy specification of even complex actions: This enables the tester to restrict the search space to the interesting operations. It also allows to go beyond simple clicks and keystrokes to generate mouse gestures which drive even complex GUIs and exercise the majority of their functionalities.
- Revealed faults are reproducible: Since crashing sequences are often relatively short and their actions are parameterized with the widgets they are executed on, they can be reliably replayed, which makes the majority of the crashes reproducible. This allows the developer to better understand the fault and to collect additional information during playback. He may even replay the sequence in slow motion to observe the course of events.
- Since we employ the operating system's Accessibility API the SUT does not require any instrumentation which makes the approach feasible for many technologies and operating systems: Many applications are not designed with testing in mind and it can be difficult to add testing hooks later on [7]. Our framework still allows to test those applications, without any instrumentation effort. We deliberately do not make use of any coverage techniques based on source code or bytecode instrumentation. For many large applications it is impractical or even impossible to measure code coverage, especially if the source code is not available. And by far not all applications run in virtual machines such as the JVM or Microsoft's CLR. Techniques that rely on bytecode instrumentation can certainly make use of additional information to guide the test case generation more effectively, but they are restricted to certain kinds of technologies. We strive for general applicability.

The results that we obtained from our experiments are encouraging and show the suitability of the approach for nontrivial SUTs. We proved that complex popular applications can be crashed and that those crashes are reproducible to a high degree. Once setup, the tests run completely automatic and report crashes without any additional labour.

6 Future Work

Although first results are encouraging, more experimentation needs to be done to find out why the Q-learning approach did not work as expected and how difficult writing Prolog specifications would turn out for real test practitioners.

Our current implementation runs on MacOSX, TESTAR is not restricted to any particular platform and we are in the process of developing support for Microsoft Windows. In addition, we plan implementations for other operating systems such as Linux and Android. Our approach benefits from the fact that many platforms provide an Accessibility API or one to access window manager information (such as the WinAPI).

The approach presented in this paper currently only targets critical faults such as crashes. However, we plan to extend our framework to develop techniques

for automated regression testing. Our ambitious goal is to completely replace the fragile and inherently labor intense capture and replay method and to develop a more effective and automated approach to regression testing. Therefore, we will have to use a powerful oracle which allows us to detect not only crashes but also functional faults such as incorrect output values in text fields, layout problems, etc. One way to achieve this is to perform back to back testing with two consecutive versions of an application. In this scenario, the old version serves as the oracle for the new one. The difficulty lies in detecting the intended (new features) and unintended (faults due to modifications) differences between the widget trees in each state in order to reduce the amount of false positives.

Acknowledgment

This work is supported by EU grant ICT-257574 (FITTEST) and the SHIP project (SMEs and HEIs in Innovation Partnerships) (reference: EACEA / A2 / UHB / CL 554187).

References

1. D. Amalfitano, A.R. Fasolino, and P. Tramontana. A gui crawling-based technique for android mobile application testing. In *Software Testing, Verification and Validation Workshops (ICSTW), 2011 IEEE Fourth International Conference on*, pages 252–261, March 2011.
2. Shay Artzi, Julian Dolby, Simon Holm Jensen, Anders Møller, and Frank Tip. A framework for automated testing of javascript web applications. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 571–580, New York, NY, USA, 2011. ACM.
3. S. Bauersfeld, A de Rojas, and T.E.J. Vos. Evaluating rogue user testing in industry: An experience report. In *Research Challenges in Information Science (RCIS), 2014 IEEE Eighth International Conference on*, pages 1–10, May 2014.
4. Sebastian Bauersfeld and Tanja Vos. A reinforcement learning approach to automated gui robustness testing. In *In Fast Abstracts of the 4th Symposium on Search-Based Software Engineering (SSBSE 2012)*, pages 7–12. IEEE, 2012.
5. Sebastian Bauersfeld and Tanja E. J. Vos. Guitest: a java library for fully automated gui robustness testing. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ASE 2012, pages 330–333, New York, NY, USA, 2012. ACM.
6. Sebastian Bauersfeld, Tanja E. J. Vos, Nelly Condori-Fernández, Alessandra Bagnato, and Etienne Brosse. Evaluating the TESTAR tool in an industrial case study. In *2014 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM '14, Torino, Italy, September 18-19, 2014*, page 4, 2014.
7. S. Berner, R. Weber, and R.K. Keller. Observations and lessons learned from automated testing. In *Software Engineering, 2005. ICSE 2005. Proceedings. 27th International Conference on*, pages 571–579, May 2005.
8. C. Bertolini, G. Peres, M. d' Amorim, and A. Mota. An empirical evaluation of automated black box testing techniques for crashing guis. In *Software Testing Verification and Validation, 2009. ICST '09. International Conference on*, pages 21–30, April 2009.

9. Ivan Bratko. *Prolog Programming for Artificial Intelligence*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.
10. B. Hofer, B. Peischl, and F. Wotawa. Gui savvy end-to-end testing with smart monkeys. In *Automation of Software Test, 2009. AST '09. ICSE Workshop on*, pages 130–137, May 2009.
11. Si Huang, Myra Cohen, and Atif M. Memon. Repairing gui test suites using a genetic algorithm. In *ICST 2010: Proceedings of the 3rd IEEE International Conference on Software Testing, Verification and Validation*, Washington, DC, USA, 2010. IEEE Computer Society.
12. Zhifang Liu, Xiaopeng Gao, and Xiang Long. Adaptive random testing of mobile application. In *Computer Engineering and Technology (ICCET), 2010 2nd International Conference on*, volume 2, pages V2–297–V2–301, April 2010.
13. Atif Memon, Ishan Banerjee, Bao Nguyen, and Bryan Robbins. The first decade of gui ripping: Extensions, applications, and broader impacts. In *Proceedings of the 20th Working Conference on Reverse Engineering (WCRE)*. IEEE Press, 2013.
14. Atif M. Memon. *A comprehensive framework for testing graphical user interfaces*. 2001. Advisors: Mary Lou Soffa and Martha Pollack; Committee members: Prof. Rajiv Gupta (University of Arizona), Prof. Adele E. Howe (Colorado State University), Prof. Lori Pollock (University of Delaware).
15. Barton P. Miller, Gregory Cooksey, and Fredrick Moore. An empirical study of the robustness of macos applications using random testing. In *Proceedings of the 1st International Workshop on Random Testing*, RT '06, pages 46–54, New York, NY, USA, 2006. ACM.
16. Richard S. Sutton and Andrew G. Barto. *Introduction to Reinforcement Learning*. MIT Press, Cambridge, MA, USA, 1st edition, 1998.

Qualifying chains of transformation with coverage based evaluation criteria

Francesco Basciani, Juri Di Rocco, Davide Di Ruscio, Ludovico Iovino, Alfonso Pierantonio

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. Based on our approach, when we propose one or more transformation chains to the user, it is difficult for him to choose one path instead of another without considering the semantic properties of a transformation.
In this paper when multiple chains are proposed to the user, according to his requirements, we propose an approach to classify these suitable chains with respect to the coverage of the metamodels involved in the transformation. Based on coverage value, we are able to qualify the transformation chains with an evaluation criteria which gives as an indication of how much information a transformation chain covers over another.

1 Introduction

Model-driven engineering (MDE) is a software discipline that employs models for describing problems in an application domain by means of metamodels. Different abstraction levels are bridged together by automated transformations which permit source models to be mapped to target models. 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. This process can be supported by an infrastructure based on a graph representation able to calculate the possible transformation compositions going from one model to another. Technically the entire process is supported by a repository of models, metamodels and model transformations previously presented in [6]. This automatic process can be called *transformation chaining* and has been treated in [7]. Moreover the possible chains outcome of the discovery in the model transformations stored in the repository, could be more than one and the user is responsible for choosing the better one for its purpose. Beyond the metamodel compatibility property, selecting and chaining model transformations can involve also other

properties like information loss, frequency of use, user rating or metamodel coverage. All those possible properties could be considered in a model transformation process in order to facilitate the user in the selection phase, when more than one suitable chains are proposed to the user, and one of those must be selected.

Our work extends the approach defined in [7], to support the automatic discovery and composition of model transformations with respect to user requirements. In particular 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 and proposed to the user. The extension offered here comprehends a mechanism to provide a chain transformation evaluation index in order to guide the user in the chain transformation selection. To this end we define two properties (that in Section 3 we call *Coverage Input* and *Coverage Output*) on the transformations which allow us to understand how much information is preserved within a transformation: with this information we provide a selection criterion when we are dealing with multiple chains. In this paper, we highlight the process able to calculate the metamodel *coverage*, that will be proposed to the user when more than one possible transformation chains are are pulled out.

This paper starts with a background section where the chaining mechanism is explained and in Section 3 we propose a chaining classification based on coverage while providing its formal definition. In Section 4 we explore the problem of having multiple chains in response to the user requests showing a scenario in which we use the coverage criteria to qualify them. Related works in Section 5 and Section 6 concludes the paper.

2 Model transformation chaining: background

Composing model transformations is a difficult problem that can be approached in two different ways [5]: by chaining separate model transformations and passing models from one transformation to another (*external composition*), or by composing two model transformation definitions into a new model transformation (*internal composition*). Even though both methods for composing transformations are important and complement each other, in this paper we focus on external composition[1]. Figure 1

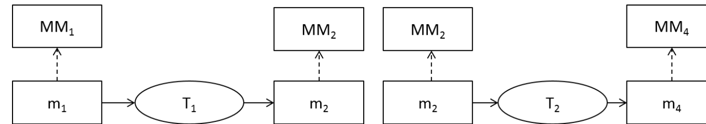


Fig. 1. Model transformation chain example

shows an explanatory model transformation chain. In particular, T_1 is a model transformation that generates models conforming to the target metamodel $\mathtt{MM}_2$ from models conforming to $\mathtt{MM}_1$. Additionally, T_2 is a model transformation that generates models conforming to $\mathtt{MM}_4$ from models conforming to the source metamodel $\mathtt{MM}_2$. Since the input metamodel of T_2 is also the output metamodel of T_1, then these two transformations

[1] For readability reasons, hereafter with the term *composition* we refer to *external composition*. Moreover, the terms *composition* and *chaining* are used interchangeably.

can be chained. Over the last years, different approaches have been studied to support the composition of model transformations (e.g., see [2,3,4,5]). The main activities that are typically performed when chaining model transformations are summarized in the following:

- *Specification of model transformation chains:* in this activity by considering the locally available model transformations, chains are specified by means of dedicated languages. For instance, in [8] the authors propose Wires*, a domain-specific language for the specification and orchestration of ATL transformations only. Another common way to chain model transformations is to use ANT scripts [2,3,4]. In [9] the authors propose the adoption of feature models to support the design of model transformation chains. In such a work, transformations are considered as features that are properly composed as specified in the considered feature models.
- *Execution of the specified model transformations chains:* in this phase the chains previously specified are executed on the source models given by the user. The execution environments of the adopted transformation languages are employed.

The first activity is the most complex one and over the last years a number of works have been presented to support it. The focus was mainly on the following aspects:

- *pre-* and *post-conditions* of transformations: when chaining transformations the conditions of applicability of a transformation (pre-conditions) and the conditions of validity of the resulting transformation (post-conditions) have to be satisfied. In [10] the authors propose an approach which adopting Higher-Order Transformations (HOT), is able to discover hidden chaining constraints between endogenous transformations by statically analysing the transformation rules.
- *commutativity/transformation order*: two model transformations are commutative (or parallel independent) if they can be chained in either order and produce the same results. In [1] the authors focus on this problem by providing an approach that permits to statically analyse two transformations and check if they are commutative or not.

In all the works mentioned above, 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 the sequential application $T_1; T_2$ if $MM_2 \subseteq MM_3$.*

[2] Apache Ant: http://ant.apache.org/
[3] Epsilon Workflow: http://www.eclipse.org/epsilon/doc/workflow/
[4] ATL-specific launch configurations and ANT tasks: http://wiki.eclipse.org/ATL/Howtos

The main focus of such works is to improve the solution found in like [1], in which authors check if two given transformations can be chained or not with respect to the metamodel compatibility property defined in [7]. Then, compatibility can be exploited to manually defining chains and singularly selecting the required transformations [9,8,3]. In [7] the authors, in response to user requests, define an automatic process in order to determine the transformation chains. Specifically, they define two activities *discovery of required model transformation (Discovery)* and *derivation of model transformation chains (Derivation)*, that by relying on a graph-based representation of a repository of artifacts [6] are able to determine all the paths between the source and target metamodels, i.e. all the possible chains that meet user requests.

3 Proposing chaining classification based on coverage

Figure 2 shows an extended version of the process seen in [7], by introducing the new activity ❷. In the previous process there were the activity ❶ (that provides a set of chains that transform a given model into a new one conform to the given target metamodel) and activity ❸ that executes the chosen chain.

The purpose of the new activity ❷ is to evaluate a specific transformation chain in order to facilitate the user chain selection. This evaluation is based on the amount of preserved information from the transformation. Therefore, with this new evaluation criteria we enrich the basic selection criteria for a chain with a new one: the coverage of the transformation with respect to the source and target metamodel.

In [11] Vignaga states that the coverage of a transformation, with respect to the source metamodel can be defined as the quotient between the total number of distinct source metaclasses whose instances are used in a transformation for producing the target models, and the total number of metaclasses in the source metamodels.

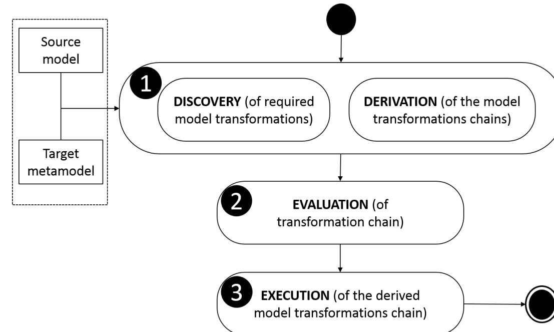


Fig. 2. Model transformations chaining process

Analogously we define the coverage of a model transformation with respect to its input metamodel (respectively output metamodel) as the ratio between the number of elements of the transformation that refer to the input metamodel (respectively output metamodel) with the total number of elements of the input metamodel itself (respectively metamodel output).

Metamodels / Transformation coverage checks which parts of a source (or target) metamodel are referenced by a given model transformation [12]. In this work, by analyzing the elements that compose the metamodels and transformations, we propose to use the value of coverage between the input and output metamodels and the transformation to classify the accuracy of the transformation we are choosing in the chaining

process. This is one of the possible criteria that can be used to evaluate transformations during the chaining process.

In the following we focus on how the coverage of an ATL model transformation is calculated with respect to its input and output metamodels. To obtain the *coverage values* (one for the input, *Coverage Input* , and one for the output, *Coverage Output*) the relationship between the number of elements of the input metamodel of the model transformation ($nElemInT(inputMM, T)$) (respectively output metamodel, $nElemInT(outputMM, T)$) on the number of elements of the input metamodel itself ($nElemInMM(inputMM)$) (respectively output metamodel, $nElemInMM(outputMM)$) is evaluated. More formally:

Definition 3 (Elements in a class). *Let $c \in C$ be a metaclass, let $nAttribute : C \rightarrow \mathbb{N}$, $nReference : C \rightarrow \mathbb{N}$, $nInheritsAttribute : C \rightarrow \mathbb{N}$, $nInheritsReference : C \rightarrow \mathbb{N}$ be functions that given a metaclass return the number of its attributes, the number of its references, the number of its inherits attributes and the number of its inherits references, respectively. Then $nElemInClass : C \rightarrow \mathbb{N}$ is defined as*

$$nElemInClass(c) = nAttribute(c) + nReference(c) + nInheritsAttribute(c) + nInheritsReference(c)$$

We are counting the number of elements in a class in terms of attributes and references (both inherited and non-inherited).

Definition 4 (Elements in a package). *Let P be a set of all packages and C a set of classes in metamodel. Let $p \in P$ be a package, let $classNotAbstract : C \rightarrow C', C' \subseteq C$, a function that given a package returns a set of its non-abstract class, let $subPackage : P \rightarrow P', P' \subseteq P$ be a function that given a package returns a set of its sub packages then $nElemInPackage : P \rightarrow \mathbb{N}$ is defined as*

$$nElemInPackage(p) = \sum_{s \in subPackage(p)} nElemInPackage(s) + \sum_{c \in classNotAbstract(p)} nElemInClass(c)$$

We are counting the number of elements in a package (and its sub-packages) and its non-abstract classes.

Definition 5 (Elements in a metamodel). *Let MM be a set of metamodels and let P be a set of packages. Let $mm \in MM$ be a metamodel, let $packageSet : MM \rightarrow P$ be a function that given a metamodel returns a set of its packages, then $nElemInMM : MM \rightarrow \mathbb{N}$ is defined as*

$$nElemInMM(mm) = \sum_{p \in packageSet(mm)} nElemInPackage(p)$$

We are counting the number of elements contained in a metamodel looking among its packages.

Definition 6 (Number of covered elements). *Let C be a set of metaclasses , let A be a set of attributes, let R a set of references, let T a set of transformations, let O a set of output pattern, let I be a set of input pattern, let B be a set of bindings, let $outPattern$:*

$T \rightarrow O$ be a function that given an transformation returns a set of all output pattern, let $inPattern : T \rightarrow I$ be a function that given an transformation returns a set of all input pattern, let $bindings : T \rightarrow B$ be a function that given a transformation returns a set of all bindings, let $referencedElement : B \rightarrow A \cup R$ be a function that given a binding returns a attribute or reference referenced by binding, let $t \in T$ be a transformation, let $mm_1 \in MM$ be a metamodels, let $isInMMClass : C, MM \rightarrow \{0, 1\}$ be a function that given a mataclass and a metamodel returns 1 if i are contained in MM, 0 otherwise, let $isInMMAttribute : A, C \rightarrow \{0, 1\}$ be a function that given an attribute and a metaclass returns 1 if i are contained in MM, 0 otherwise, let $isInMMRef : R, C \rightarrow \{0, 1\}$ be a function that given a reference and a metaclass returns 1 if i are contained in MM, 0 otherwise, let $distinctMetaclassInOp : T \rightarrow \{c \in C | \exists op \in outPattern(T) \wedge c = op.referredElements\}$ be a function that given a transformation returns a set of distinct metaclasses referenced by an out pattern, let $distinctMetaclassInIp : T \rightarrow \{c \in C | ip \in inPattern(T) \wedge c = op.referredElements\}$ be a function that given a transformation returns a set of distinct metaclasses referenced by an in pattern, let $distinctAttBindings : T \rightarrow \{a \in A \mid \exists b \in bindings(T) \ \wedge \ a = referencedElement(b)\}$ be a function that given a transformation returns a set of distinct attribute referenced by a binding, let $distinctRefBindings : T \rightarrow \{r \in R | \exists b \in bindings(T) \ \wedge \ r = referencedElement(b)\}$ be a function that given a transformation returns a set of distinct reference referenced by an binding then $nElemInT : T, MM \rightarrow \mathbb{N}$ is defined as

$nElemInT : (mm_1, t) = \sum_{i \in distinctMetaclassInOp(T)} isInMMClass(i, MM) +$

$\sum_{i \in distinctMetaclassInIp(T)} isInMMClass(i, MM) +$

$\sum_{i \in distinctAttrInBindings(T)} isInMMAttribute(i, MM) +$

$\sum_{i \in distinctRefInBindings(T)} isInMMReference(i, MM)$

We are counting how many elements of the metamodel, provided as input to the function (which can be either the source metamodel that the target metamodel), are covered by the transformation.

Definition 7 (Transformation coverage). *Let $mm_1 \in MM$ be a metamodels, let $t \in T$ be a transformation, then $tCoverage : (MM, T) \rightarrow [0, 1]$ defined as*

$$tCoverage(mm_1, t) = \frac{nElemInT(mm_1, t)}{nElemInMM(mm_1)}$$

With this formula we calculate the coverage of a transformation. Depending on whether we provide as input of the function the source or the target metamodel, we will have as a result, respectively, the input or output coverage.

How the coverage of a chain is calculated Starting from the graph in Fig. 3, representing our repository, and the inputs provided by the user (see Fig. 2) through a *breadth-first search (BFS)*, the system retrieves all the paths. Each retrieved path starting from the node that represents the metamodel provided as input, arrives at target metamodel still supplied by the user. It is important to remark that at this stage the coverage values on the edges are not yet considered.

Once the list of paths between source and target is obtained, the *chain coverage* is calculated and this is done through the formula derived from the following definition:

Definition 8 (Chain Coverage). *Let TC be a set of transformation chains, let MM be a set of metamodels, let T be a set of transformations, let $t \in T$ be a transformation, let $ct \in TC$ be a chain transformation, let $sourceMM : T \rightarrow MM$ be a function that given a transformation return the source metamodel, let $targetMM : T \rightarrow MM$ be a function that given a transformation return the target metamodel then $chainCoverage : TC \rightarrow [0, 1]$ is defined as*

$$chainCoverage(ct) = \prod_{t \in transformationChain(ct)} tCoverage(t, sourceMM(t)) * tCoverage(t, targetMM(t))$$

With this formula we take into account on the one hand the values of coverage in a chain and on the other hand we take into account the length of the same (having values between 0 and 1). Once we determined these values (for each insertion and/or deletion of a model transformation) they are retained as weights on edges in the graph structure (you can see an example in Fig. 3).

4 Dealing with multiple chains

The sub-activities *Discovery* and *Derivation* of activity ❶ in Fig. 2 might give place to different possible chains among which the user must choose.

and before going ahead with executing activity, users have to select one of them.

As said before, possible path of chaining can be distinguished based on different parameters and with this work we focus on the coverage of the transformation respect to the source and target metamodel. Based on the framework proposed in [6] and taking advantage of all its services in support of the chaining mechanism, what we do is to enhance the representation graph for the maintenance of the artifacts, adding values at the beginning (*Coverage Input*) and the end (*Coverage Output*) of an arc which represents a transformation.

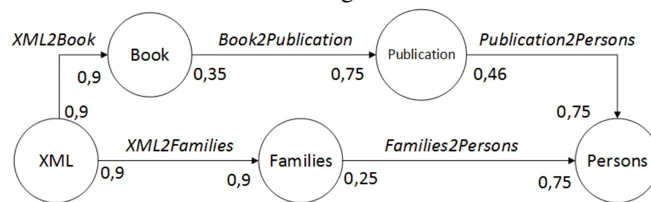


Fig. 3. Graph structure of metamodels and model transformations with *Coverage Input* and *Coverage Output* on edges

In other words an edge represents a transformation from source metamodel to the target in the repository, and the *coverage in input/output* represents the ratio between elements in the source/target metamodels and the elements consumed/produced by the transformation respectively.

Let us suppose that the user gives as input an *XML* model and requests to generate a target *Persons* model. According to the available transformations in the repository, the result of activity ➊ is the list of the chains, which in the example are these two:

(1) $XML2Families \rightarrow Families2Persons$
(2) $XML2Book \rightarrow Book2Publication \rightarrow Publication2Persons$

The model transformation *Families2Persons* is shown in Fig. 4.a, it is the second transformation composing the chain (1) that has been retrieved. This transformation has *Families* metamodel as source, represented in Fig. 4.b and *Persons* metamodel as target, depicted in Fig. 4.c.

```
1  -- @path Families= /metamodel/Families.ecore
2  -- @path Persons=metamodel/Persons.ecore
3
4  module Families2Persons;
5  create OUT : Persons from IN : Families;
6
7⊕ helper context Families!Member def: familyName : String =
21
22⊕ helper context Families!Member def: isFemale() : Boolean =
32
33⊖ rule Member2Male {
34     from
35         s : Families!Member (not s.isFemale())
36     to
37         t : Persons!Male (
38             fullName <- s.firstName + ' ' + s.familyName
39         )
40 }
41
42⊖ rule Member2Female {
43     from
44         s : Families!Member (s.isFemale())
45     to
46         t : Persons!Female (
47             fullName <- s.firstName + ' ' + s.familyName
48         )
49 }
50
```

a) Families2Persons.atl

```
Families
  Family
    lastName : String
    father : Member
    mother : Member
    sons : Member
    daughters : Member
  Member
    firstName : String
    familyFather : Family
    familyMother : Family
    familySon : Family
    familyDaughter : Family
PrimitiveTypes
```

b) Families.ecore

```
Persons
  Person
    fullName : String
  Male -> Person
  Female -> Person
PrimitiveTypes
```

c) Persons.ecore

Fig. 4. Families2Persons example

This list of chains is processed by the activity ➋ (*Evaluation*). We calculate how much the model transformation actually "covers" the elements of the source and target metamodel and we obtain Table. 1, summarizing the values that the system extrapolates by analyzing the source and target metamodels, and the model transformation.

The process to assign the coverage values to the existing transformations in the repository can be summarized as:

Metamodels static analysis → Transformation static analysis → Coverage Calculation

The system firstly extract from the static analysis of metamodels and transformation, the *Matched Rules* in the transformation and its related source and target patterns:

nElemInMM(Families)					
Classes	Abstracts	StructuralFeatures	InheritedStructuralFeatures	Attributes	References
Family	false	5*	0	1	4
Member	false	5**	0	1	4
Total			12		

nElemInMM(Persons)					
Classes	Abstracts	StructuralFeatures	InheritedStructuralFeatures	Attributes	References
Person	true	1***	0	1	0
Male	false	0	1	0	0
Female	false	0	1	0	0
Total			4		

nElemInT(Families, Families2Persons) / nElemInT(Persons, Families2Persons)				
	Classes (Not Abstract)	Attributes	References	Total Elements
Families	2	2	8	12
Persons	2	1	0	4

* *lastName, father, mather, sons, daughter*
** *firstName, familyFather, familyMother, familySon, familyDaughter*
*** *fullName*

Table 1. Report extrapolated by the system concerning the source and target metamodels and the model transformation.

– Source Classes: *Member*;
– Source Attributes covered by Matched Rules: *firstName*, *familyName*;
– Target Classes: *Male*, *Female*;
– Target Attributes: *fullName*;

than the system, according to the Definition 7 seen previously, outputs two *coverages values*, one for the source and one for the target:

$$TCoverage_{input} = \frac{3}{12} = 0,25 \qquad TCoverage_{output} = \frac{3}{4} = 0,75$$

Referring to the structural graph, representing our repository in Fig. 3, assuming that user requests as input metamodel XML and requires the target metamodel as $Persons$, the system derives all the possible chains with the following *coverage values*:

Transformation chain	Coverage Value
$XML2Families \rightarrow Families2Persons$	$0,15$
$XML2Book \rightarrow Book2Publication \rightarrow Publication2Persons$	0.07

The result that comes out from this activity of "evaluation" suggest that could be convenient to invoke a chain like *XML2Families* → *Families2Persons*, that has a coverage value higher than the other one. After the user's selection, activity ❸ can start, i.e. the execution of the chain.

5 Related work

Increasingly, model transformation chaining has been a current topic of research and it has been treated from different perspectives.

In [2] the authors present a convenient approach to design highly flexible chains from existing independent model transformations. The main difference with the presented approach is the *design* part that in our case is automatically calculated by the engine in discovery mode. They propose to artificially change the input and output of transformations in order to recover the compatibility of the involved metamodels. The work in [2] is an extension of what is presented in [1] where the authors address the problem of identifying conflicts between transformations, and checking if two transformations are commutative or not.

A language for defining composition of transformations is given in [3]. To support the concrete realization of transformation chains they propose a language to allow the concatenation of transformation components. A recent work [9] uses feature models to classify model transformations. Based on these feature models, automated techniques help the designer to generate executable chain of transformations. Another interesting work has been exposed in [13] where transformation chaining is called *orchestration*. This paper introduces a graphical executable language for the orchestration of ATL transformations, which provides appropriate mechanisms to enable the modular and compositional specification and execution of complex model transformations chains. The work presented in [4] describes an approach to designing large model transformations for large languages, based on the principle of separation of concerns. Chains are built by linking output parameters to input ones through connectors. Differently from such works we do not require the specification of transformation chains that in our approach are automatically derived with respect to the request of the user and to the transformations, which are stored in a dedicated repository.

In [14] the authors propose a mechanism of module superimposition to compose small and reusable transformations. The idea is to overlay several transformation definitions on top of each other and then execute them as one transformation. Differently from our work, the approach in [14] is specific for ATL and it is an internal composition approach. The work proposed in this paper is an external composition technique and it is independent from the used model transformation language. Vignaga [11] describes a number of metrics for ATL model transformations, described according to the ATL metaclass to which they apply. In this paper and also in [12] the coverage is treated and the formula exposed has been reused also in our work.

6 Discussion and conclusions

In this paper we present an improvement of the approach seen in [7] that support model transformations chaining. 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 our approach are related to the possibility of qualifying the chains with the coverage that helps the user to choose

a chain among all the others that the system is able to retrieve. This is done by analyzing all the elements that compose both metamodels (input and target) of a model transformation and the model transformation itself. With this technique, in some way, we are going to evaluate how much information is preserved or lost when you make a single transformation and consequently how much it preserves or how much is lost in the whole chain. We are aware that in order to better characterize the concept of *information loss* we should consider in a different way both the models in input and output of a transformation. This, however, will be investigated in the next works.

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., Van Baelen, S., Hovsepyan, A., Joosen, W., Berbers, Y.: Towards a Transformation Chain Modeling Language. In Vassiliadis, S., Wong, S., Hmlinen, T., eds.: Embedded Computer Systems: Architectures, Modeling, and Simulation. Volume 4017 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2006) 39–48
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. Basciani, F., Di Rocco, J., Di Ruscio, D., Di Salle, A., Iovino, L., Pierantonio, A.: MDEForge: an extensible Web-based modeling platform. CloudMDE 2014 (2014) 66
7. Basciani, F., Di Ruscio, D., Iovino, L., Pierantonio, A.: Automated chaining of model transformations with incompatible metamodels. In: Model-Driven Engineering Languages and Systems. Springer (2014) 602–618
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. 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
10. Chenouard, R., Jouault, F.: Automatically Discovering Hidden Transformation Chaining Constraints. In Schrr, A., Selic, B., eds.: Model Driven Engineering Languages and Systems. Volume 5795 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2009) 92–106
11. Vignaga, A.: Metrics for Measuring ATL Model Transformations. Technical report (2009)
12. Wang, J., Kim, S.K., Carrington, D.: Verifying metamodel coverage of model transformations. In: Software Engineering Conference, 2006. Australian. (2006) 10 pp.–
13. 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
14. Wagelaar, D., Van Der Straeten, R., Deridder, D.: Module superimposition: a composition technique for rule-based model transformation languages. Software & Systems Modeling **9** (2010) 285–309

Describing the correlations between metamodels and transformations aspects

Juri Di Rocco, Davide Di Ruscio, Ludovico Iovino, Alfonso Pierantonio

Department of Information Engineering Computer Science and Mathematics
University of LAquila, Italy,
name.lastname{@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. One of the most important artifact is model transformation that are considered to be the heart and soul of MDE and as such advanced techniques and tools are needed for supporting the development, quality assurance, maintenance, and evolution of model transformations. Several works propose the adoption of metrics to measure quality attributes of transformation without considering any metamodel aspects. In this paper, we present an approach to understand structural characteristics of metamodels and how the model transformations depend on corresponding input and target metamodels.

Keywords: Model Driven Engineering, metamodeling, metamodel metrics, transformation metrics

1 Introduction

Metamodels are a key concept in Model-Driven Engineering [22]. Almost any artifact in a modeling ecosystem [13] has to be defined in accordance to a metamodel, which represents an ontological description of application domains [10]. Metamodels are important because they formally define the modeling primitives used in modeling activities and represent the *trait-d'union* among all constituent components. One of this components are model transformations (MT), in fact MT play a key role since they permit to bridge different abstraction levels by automatically mapping source models to target ones. In [23] model transformations are considered to be the "heart" and "soul" of MDE and as such they require to be treated in a similar way as traditional software artifacts [2]. 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 [20] and transformations [3,28] with the aim of assessing quality attributes, such as understandability, reusability, and extendibility [7]. 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 lifecycle especially in case of metamodel evolutions. To this end, some works propose the adoption of metrics for analysing metamodels [17,19] and transformation [28]

as typically done in software development by means of object-oriented measurements [16]. Starting from our previous work [11], we are interested in better understanding metamodel characteristics and how metamodels and transformations are correlated by investigating the correlations of different metrics applied on a corpus of more than 450 metamodels and 90 transformations. On one hand we propose an approach for *a)* measuring certain metamodeling aspects (e.g., abstraction, inheritance, and composition) that modelers typically use; and *b)* for revealing what are the common characteristics in metamodeling that can increase the complexity of metamodels hampering their adoption and evolution in modeling ecosystems [13]. On other hand we propose an approach for identifying how the transformations are correlated to metamodels. The identified correlations permit to draw interesting considerations e.g. how a model transformation is typically structured depending on the considered metamodels, and how does the complexity of metamodels has an impact on the overall model transformations development. Such considerations can be preparatory to further analysis that are very common in software development [9], e.g., estimating the effort required to develop model transformations by considering the structural characteristics of the source and target metamodels.

The paper is structured as follows: Section 2 describes the process we have conceived and applied to analyze metamodels. Interesting correlations are discussed in Section 3. Section 4 discusses related work and Section 5 concludes the paper and draws some research perspectives.

2 The correlation among metamodels and transformations

Software metrics have been proposed to assess and predict software effort and quality [15] and recent research has proposed the adoption of metrics to measure transformations. In particular, metrics on transformations have been investigated [28,3] to support the measurements of model transformations with the aim of understanding transformations via quantitative evaluations. For instance, in [28] specific metrics have been conceived to measure ATL transformations, and in [4] authors define the meaning of several quality attributes in the context of model transformations and align them to a set of metrics.

The adoption of metrics to measure metamodels has been recently proposed in [17,19,12]. In particular, in [17] authors apply object-oriented measurements to understand common structural characteristics of metamodels, whereas [19] proposes a measuring mechanism for assessing the quality of metamodels. To the best of our knowledge, none of the existing approaches calculate transformation metrics with the aim of correlating them.

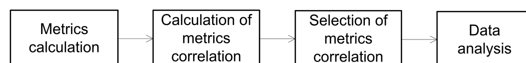


Fig. 1. Overview of the process for metamodel analysis

Since it is reasonable to claim that the complexity of model transformations is somehow related to that of the source and target metamodels, in our opinion in order to have a complete measurement of model transformations, it is necessary to identify also possible correlations between transformation and metamodel metrics e.g., to figure out at what extent the number of matched rules of given ATL transformation depends on the number of metaclasses in the source and/or target metamodels.

To this end, in this section the measurement process shown in Fig. 1 is presented. In particular, the first step of the process consists in applying a number of metrics on a representative corpus of transformations and corresponding metamodels. Afterwards the calculated metamodel and transformation metrics are correlated among them by using statistical tools. Finally, the collected data are analysed in order to cross/link structural characteristics of transformations and metamodels, e.g., how the different kinds of ATL rules (i.e., matched, lazy, and called) are typically used. It is important to remark that in the analysis step, metamodel metrics are also considered in order to identify possible correlations among transformation and metamodel metrics (e.g., how the number of metaclasses in the target metamodel impacts the structural characteristics of transformations in terms of number of matched rules, helpers, etc.). In [12], we describe the process, shown in 1, we have applied to identify linked structural characteristics and to understand how they might change depending on the nature of metamodels. In this work we have extended this process in order to calculate different set of metrics from different artifacts (metamodels and transformations) and to understand how the model transformations are dependent from corresponding input and target metamodels.

2.1 The proposed measurement process

The first step of the proposed process consists of the application of metrics on a data set of metamodels and transformations. Concerning the applied metrics on metamodels we borrowed those in [17] and added new ones by leading to a set of 28 metrics. Due to space limitations, in the rest of the paper we consider only the metrics shown in Tab. 1 for metamodels and shown in Tab. 2 form transformation. The corpus of the analyzed metamodels and transformations has been obtained by retrieving artifacts from different repositories, i.e., EMFText Zoo [6], ATLZoo [5], Github, and GoogleCode. To perform such analysis we have automatize the process for metrics calculation using a eterogenous repository called MDEForge presented in [8]. The calculated data are exported in CSV files encoding the values of all the calculated metrics. Generating CSV files enables the adoption of statistical tools like IBM SPSS, Microsoft Excel, R and Libreoffice Calc for subsequent analysis of the generated data.

2.2 Calculation and selection of metrics correlations

Correlation is probably the most widely used statistical method to detect cross-links and assess potential relationships among observed data. There are different

techniques and indexes to discover and measure correlations. In the following we overview the Pearson's and Spearman's coefficients that we have considered in this paper to measure the correlations among calculated metamamodel metrics.

The *Pearson's correlation coefficient* [18] was developed by Karl Pearson from a related idea introduced by Francis Galton in the 1880s. It is widely used in the sciences as a measure of the degree of linear dependence between two variables. In particular, the Pearson correlation coefficient is appropriate when it is possible to draw a regression line between the points of the available data (e.g., see the diagrams A and B in Fig. 2).

The *Spearman's correlation coefficient* [24] was used by Charles Spearman in the 1900s in the psychology domain. This coefficient is better than Pearson to manage situations when there is a monotonic relationship between the considered variables. For instance, in the cases shown in the diagrams C and D in Fig. 2, the Pearson coefficient would wrongly identify a very low correlations among the considered data. This is due to the fact that the assumption of linear relationships required by Pearson is not satisfied. Contrariwise, Spearman's correlation index would perform better in cases of monotonic relationships as in the diagrams C and D in Fig. 2

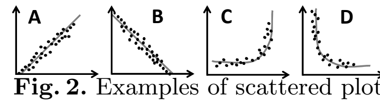


Fig. 2. Examples of scattered plots

It is also important to note that the assumption of a monotonic relationship is less restrictive than a linear relationship (an assumption that has to be met by the Pearson correlation). For this reason, we use Spearman only for highlighting curvilinear correlations. Finally, both Pearson's and Spearman's correlation indexes assume values in the range of -1.00 (perfect negative correlation) and +1.00 (perfect positive correlation). A correlation with value 0 indicates that there is no correlation between two variables. In order to assess the strength of correlations it is possible to consider the guide that Evans [14] suggests for the *absolute value* of the correlation indexes, i.e., [*0.0*,*0.19*] very weak, [*0.20*,*0.39*] weak, [*0.40*,*0.59*] moderate, [*0.60*,*0.79*] strong, and [*0.80*,*1.0*] very strong.

Metamodel metrics correlations Once the metamodel metrics have been calculated, the most correlated ones are identified and selected. In particular, we have calculated the Pearson's correlation indexes for all the values of the metamodel metrics. The outcome of this operation is a correlation matrix as the one shown in Fig 3. The discussion is based on the correlation matrix shown in Fig 3 and by considering the most interesting correlations having value greater than 0.60 (thus strong or even very strong). Because of lack of space it is not possible to discuss all the identified correlations that include the metrics shown in Table 1 and 2. However, interested readers can refer to the spreadsheet available online[1] containing all the obtained results. For instance, the number of MC[2] (number of metaclasses) is strongly correlated with the number of CMC

[1] http://www.di.univaq.it/ludovico.iovino/data-mise2015.html
[2] For the complete list of acronyms in the table we refer to [11]

(number of concrete metaclasses) as testified by their Pearson's correlation index having value 0.997.

	#MC	#AMC	#CMC	#IFLMC	#SF	#ASF	#TCWS	#MGHL	#MHS	LNS
#MC										
#AMC	0.451									
#CMC	0.997	0.377								
#IFLMC	0.874	0.139	0.894							
#SF	0.831	0.574	0.810	0.488						
#ASF	-0.102	-0.064	-0.100	-0.176	0.155					
#TCWS	0.993	0.451	0.990	0.890	0.797	-0.131				
#MGHL	0.666	0.637	0.633	0.534	0.558	-0.216	0.678			
#MHS	0.704	0.463	0.688	0.562	0.620	-0.164	0.704	0.561		
LNS	-0.082	-0.055	-0.080	-0.030	-0.108	-0,181	-0.072	-0.100	-0.094	

Fig. 3. Pearson Correlation values related to metamodel metrics

Model transformation and metamodel metric correlations The interesting part of our analysis relies on correlating model transformation and metamodel metrics. To this end a correlation matrix based on the Spearman's index has been calculated and a fragment is shown in Fig 4. The matrix relates model transformation metrics with metrics calculated on the corresponding source and target metamodels. For instance, according to the calculated matrix, the number of output patterns (OP) of a model transformation is strongly related with the number of metaclasses (MC) contained in the output metamodel.

	B	IP	OP	TR	MR	LR	CR	RWF	RWD	H	HWC	HNC	CRT	
MC	0.450	**0.690**	0.467	0.452	0.402	0.295	0.248	0.267	0.329	-0.002	-0.082	0.168	0.088	INPUT
AMC	0.340	0.463	0.339	0.412	0.374	0.264	0.228	0.390	0.306	0.083	-0.019	0.229	-0.003	
CMC	0.478	0.504	0.496	0.468	0.412	0.290	0.289	0.260	0.360	0.036	-0.040	0.178	0.098	
SF	0.503	0.394	0.467	0.363	0.334	0.208	0.282	0.126	0.315	-0.037	-0.138	0.139	0.051	
MC	0.520	0.542	**0.783**	**0.746**	0.500	0.223	0.369	0.480	0.399	0.180	0.168	0.204	0.131	OUTPUT
AMC	0.478	0.504	0.496	0.468	0.412	0.290	0.289	0.260	0.360	0.036	-0.040	0.178	0.098	
CMC	0.503	0.394	0.467	0.363	0.334	0.208	0.282	0.126	0.315	-0.037	-0.138	0.139	0.051	
SF	**0.808**	0.506	0.505	0.481	0.451	0.202	0.266	0.375	0.284	-0.008	-0.075	0.100	-0.014	

Fig. 4. Spearman Correlation values related to transformation and metamodel metrics

3 Data analysis

In this section we discuss some relevant correlations we have identified as described in the previous section. In particular, by considering some of the identified transformation metrics, it is possible to draw interesting considerations about how the constructs of the ATL language are typically used by developers.

Moreover, by considering the correlations of both transformation and metamodel metrics (see Section 3.2), further considerations can be drawn about how structural characteristics of metamodels affect the structure of the corresponding model transformations.

3.1 Metamodels correlation analysis

In this section we briefly present the most representative metrics and correlations we have discovered in this process. We present the metrics correlation discussing the meaning and highlighting the results in the graphical representation.

How the number of metaclasses is related to the adoption of abstraction constructs In this section we discuss how the size of metamodels expressed in terms of number of metaclasses is related to the adoption of abstraction constructs, i.e., abstract metaclasses, and supertypes.

In particular, as shown in Fig. 5 the number of metaclasses (MC) and the number of those with super types (MCWS) are strongly correlated (with Pearson index 0.99). More specifically, when the number of metaclasses grows, typically also the number of classes with supertypes increases. In other words, as expected, the adoption of inheritance is proportional to the size of metamodels expressed in terms of number of metaclasses. Interestingly, metamodel designers prefer to add siblings in hierarchies instead of adding new hierarchy levels. This is testified by Fig. 5 that shows the values of the *MHS* (Max Hierarchy Sibling) and *MGHL* (Max generalization hierarchical level) metrics. Such conclusions are confirmed by the Pearson correlation indexes between *MC* and *MHS* (0.70) and the one between *MC* and *MGHL* (0.66). Finally, Fig. 5 reveals that in metamodels with at most 50 metaclasses, *i)* the number of supertypes in hierarchy is in between 0 and 20, *ii)* the number of siblings in a hierarchy is in between 0 and 10, and *iii)* the maximum height of a hierarchy is in between 0 and 5. These data represent a pattern charactering the typical typical metamodel definition.

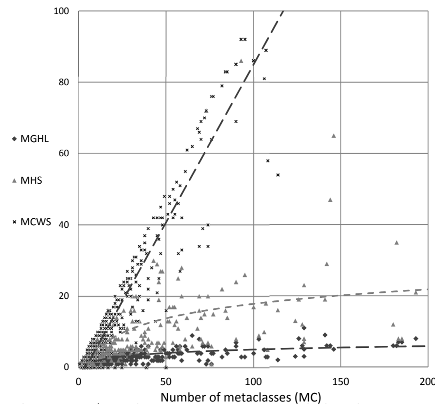


Fig. 5. Analyzing metamodel abstraction level

How structural features are used with hierarchies This section aims at comprehend how structural features are used in presence of class hierarchies. To this end, we can consider the average number of features (ASF) and the total number of metaclasses with supertypes (MCWS) metrics. Even though

the correlation index of these two metrics is low, according to the matrix in Fig 3, the Spearman approach permits to identify a greater correlation index. As shown in Fig. 6 it is evident that increasing the number of metaclasses with supertypes, the average number of structural features in a metaclass decreases. Moreover, an interesting statistical result obtained by considering the correlation between the MC and ASF metrics is that by considering metamodels having the number of metaclasses in the range between 1 and 50 , the average number of features (excluding the inherited ones) of a metaclass ranges between 1 and 5.

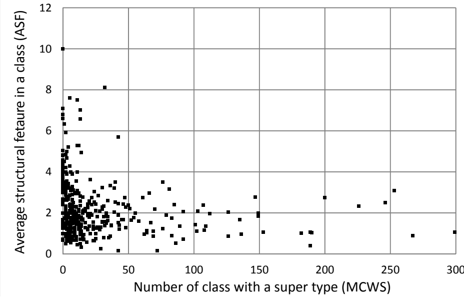


Fig. 6. Analyzing structural features introduction in hierarchies

How the number of featureless metaclasses is related to hierarchies height The correlation between the number of metaclasses with supertypes (MCWS) and the number of concrete metaclasses without features (IFLMC) is interesting for understanding how specializations of metaclasses can introduce or reduce structural features in metamodels. To this end, MCWS and IFLMC are strongly correlated as supported by the Pearson's index having value 0.890. The effect of such correlation is shown in Fig. 7[3]. In particular, by increasing the number of metaclasses with super types, the number of metaclasses without attributes or references increases too. This means that when hierarchies are introduced, usually existing features are subject to refactoring operations. Usually, what is done is to move them to super classes and to create leaves in the hierarchies inheriting features from the super types. This is in line with the typical usage of hierarchies for factorizing common aspects in superclasses.

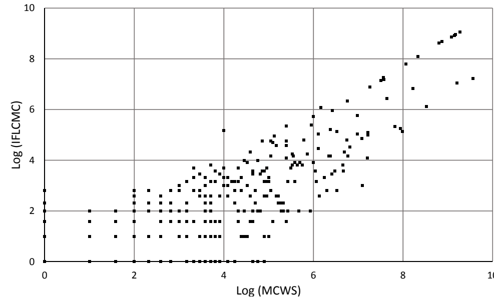


Fig. 7. Analyzing hierarchical height and featureless metaclasses

3.2 How metamodel characteristics affect model transformations

By exploiting the matrix obtained by correlating transformation and metamodel metrics, in this section we discuss how metamodels affect the development of

[3] This scattered plot diagram use date logarithmic scale for empathize the correlation

model transformations. The discussion is based on the correlation matrix shown in Fig. 4 and by considering the most interesting correlations having value greater than 0.65.

How transformation rules are influenced by target metamodels This aspect can be investigated by considering the correlation between the number of metaclasses in the target metamodel (OUT MC) and the number of TR (Transformation Rules). Such two values are correlated because of the Spearman's index having value 0.746. The graph in Fig 8[4] represents how these two values are influenced by each other in our corpus. According to the graph it is evident that increasing the number of the MC in the target metamodel the number of TR increases too. This is generally true, since the transformation writing is output-driven when the developer tries to cover all the metaclasses of the target metamodel. We can also state that the common concentration in the corpus is in the range between 1 and 20 metaclasses and 1 and 15 transformation rules, again confirming the declarative style of transformation as common choice of the developers.

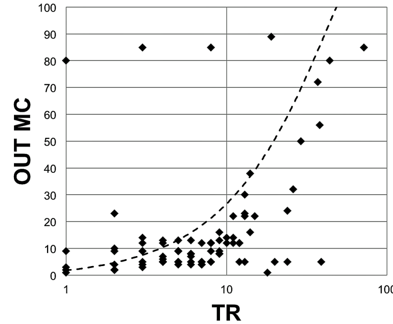


Fig. 8. How TR are influence by number of MC in target metamodel

How the structural features in the target metamodel influence the number of bindings According to the calculated Spearman correlation, the structural features (SF) of the target metamodel can influence the number of bindings (B) written in the rules of the transformations. The plot in Fig 9[4] shows that increasing the value of SF in the output metamodel (OUT SF), the number of binding grows too. The distribution is common for the number of SF between 0 and 20 distributed for the value of B that goes from 1 to about 75.

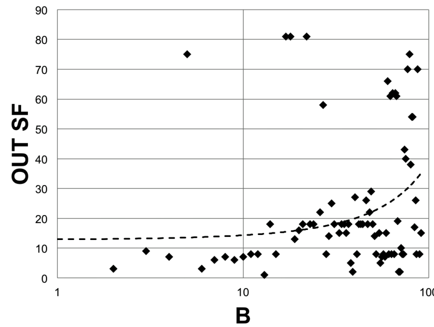


Fig. 9. How the SF in the target metamodel influence the number of B

[4] The scattered plot diagram use date logarithmic scale for empathize the correlation

How the total number of output patterns are influenced by the target metamodels According to the calculated matrix the Spearman's correlationindex between the value of OP (Output Patterns) in the rules and the number of metaclasses in the target metamodels has value 0.783. This correlation occurring in our corpus is depicted in Fig 10[4] where the value of OP in the rules of our transformations increases at the raising of the value of MC in the target metamodels. The most dense concentration is in the range of 1-10 output patterns and 1-10 metaclasses in output.

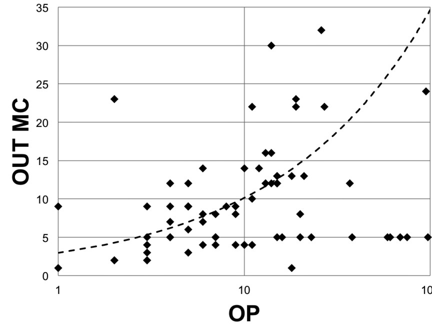


Fig. 10. How OP are influenced by the target metamodels

How the total number of input pattern are influenced by the source metamodels As anticipated in the previous sections the IP (Input Pattern) of the transformations are related to the value of MC in the source metamodel (IN MC). This is confirmed by the Spearman's correlation that results 0.692. In the graph in Fig 11[4] the distribution is less clear than the previous case but the trend is similar: increasing the value of MC in input, the value of IP increases too. This again confirms the use of declarative style as the preferred one in our corpus.

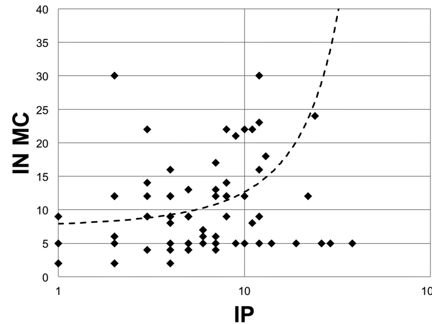


Fig. 11. How IP are influenced by the source metamodels

4 Related works

In [28] the authors introduces metrics to measure ATL transformations and the adoption of metrics to measure quality attributes of transformation without considering any metamodel aspects. In other approaches the main topic is the quality attribute driven by the metric [4], for example making the quality of model transformations measurable. In [25] the authors have focused on transformation model measurements in order to better understand transformations via a quantitative evaluation, like the declarative factor of modules and rules. In [27] an analogous approach for measuring model repositories is shown, simply considering models in the evaluation. The authors in [26] investigate factors

having impact on the execution performance of model transformations and they extracted metrics for the analysis. Van Amstel et al. propose a set of six quality attributes to evaluate the quality of model transformations [1]. All cited works propose the adoption of metrics to measure quality attributes of transformation without considering any metamodel aspects. The authors of [21] worked on how model transformations can improve the quality of models using metrics. A similar approach for understanding structural characteristics of metamodels and their relationships has been presented in [11]. Williams et al. in [17] is the first one to discuss metrics related to a large metamodel collection exposing how metamodels are commonly structured, and how they evolve over time.

5 Conclusions and future work

In this paper, we proposed a number of metrics which can be used to acquire objective, transparent, and reproducible measurements of metamodels and transformations. The first goal is to better understand the main characteristic of metamodels, how they are coupled, and how they change depending on the metamodel structure. We have also proposed an approach to analyze model transformations by considering also the corresponding metamodels. The approach relies on the correlation of different metrics and has been applied on a corpus of 450 metamodels and 90 transformations and permitted to draw interesting considerations that we intend to extend in the future.

References

1. van Amstel, M.F., Lange, C.F., van den Brand, M.G.: Using metrics for assessing the quality of asf+ sdf model transformations. In: Theory and Practice of Model Transformations, pp. 239–248. Springer (2009)
2. van Amstel, M., van den Brand, M.: Model transformation analysis: Staying ahead of the maintenance nightmare. In: ICMT 2011, LNCS, vol. 6707, pp. 108–122. Springer (2011)
3. 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)
4. van Amstel, M., Lange, C., van den Brand, M.: Metrics for analyzing the quality of model transformations. Proceedings of the 12th ECOOP Workshop on Quantitative Approaches on Object Oriented Software Engineering pp. 41–51 (2008)
5. ATLAS Group: ATL Transformations Zoo. http://www.eclipse.org/m2m /atl/atlTransformations/
6. ATLAS Group: EMFTEXT Concrete Syntaxes Zoo. http://emftext.org/index.php/EMFText_Concrete_Syntax_Zoo$
7. Bansiya, J., Davis, C.G.: A hierarchical model for object-oriented design quality assessment. Software Engineering, IEEE Transactions on 28(1), 4–17 (2002)
8. Basciani, F., Di Rocco, J., Di Ruscio, D., Di Salle, A., Iovino, L., Pierantonio, A.: Mdeforge: an extensible web-based modeling platform. pp. 66–75 (2014), http://ceur-ws.org/Vol-1242/paper10.pdf
9. Boehm, B., Abts, C., Chulani, S.: Software development cost estimation approaches a survey. Annals of Software Engineering 10(1-4), 177–205 (2000), http://dx.doi.org/10.1023/A%3A1018991717352

10. 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)
11. Di Rocco, J., Di Ruscio, D., Iovino, L., Pierantonio, A.: Mining metrics for understanding metamodel characteristics. In: Proceedings of the 6th International Workshop on Modeling in Software Engineering. pp. 55–60. MiSE 2014, ACM, New York, NY, USA (2014), http://doi.acm.org/10.1145/2593770.2593774
12. Di Rocco, J., Di Ruscio, D., Iovino, L., Pierantonio, A.: Mining metrics for understanding metamodel characteristics. In: MiSE. pp. 55–60 (2014)
13. 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)
14. Evans, J.S.B., Over, D.E.: Rationality and reasoning. Psychology Press (2013)
15. Fenton, N.E., Pfleeger, S.L.: Software Metrics: A Rigorous and Practical Approach. PWS Publishing Co., Boston, MA, USA, 2nd edn. (1998)
16. 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)
17. 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)
18. Lee Rodgers, J., Nicewander, W.A.: Thirteen ways to look at the correlation coefficient. The American Statistician 42(1), 59–66 (1988)
19. 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
20. Monperrus, M., Jezequel, J.M., Champeau, J., Hoeltzener, B.: Model-Driven Software Development. IGI Global (Aug 2008), http://www.igi-global.com/chapter/measuring-models/26829/
21. Saeki, M., Kaiya, H.: Measuring model transformation in model driven development. In: CAiSE Forum. vol. 247 (2007)
22. Schmidt, D.: Guest Editor's Introduction: Model-Driven Engineering. Computer 39(2), 25–31 (2006)
23. Sendall, S., Kozaczynski, W.: Model transformation: The heart and soul of model-driven software development. IEEE Softw. 20(5), 42–45 (Sep 2003), http://dx.doi.org/10.1109/MS.2003.1231150
24. Spearman, C.: The proof and measurement of association between two things. The American journal of psychology 15(1), 72–101 (1904)
25. Tolosa, J.B., Sanjuán-Martínez, O., García-Díaz, V., Lovelle, J.M.C., et al.: Towards the systematic measurement of atl transformation models. Software: Practice and Experience 41(7), 789–815 (2011)
26. Van Amstel, M., Bosems, S., Kurtev, I., Pires, L.F.: Performance in model transformations: experiments with atl and qvt. In: Procs. ICMT2011, pp. 198–212. Springer (2011)
27. Vépa, E., Bézivin, J., Brunelière, H., Jouault, F.: Measuring model repositories. In: Proceedings of MSM06 (2006)
28. Vignaga, A.: Metrics for measuring atl model transformations. MaTE, Department of Computer Science, Universidad de Chile, Tech. Rep (2009)

6 Appendix

Acronym	Name	Description
AMC	Number of abstract MetaClass	Number of metaclasses that cannot be instantiated in models
ASF	Average Structural Features	Average number of attributes and references in a metaclass
CMC	Number of concrete MetaClass	Number of metaclasses that can be directly instantiated
IFLMC	Number of concrete Immediately Featureless MetaClass	The number of concrete metaclasses that have no attributes or references, but may inherit features from a superclass
LNS	Isolated metaclasses	It is the percentage of metaclasses that are not connected with any other one
MC	Number of total MetaClass	Number of metaclasses in the metamodel (MC = AMC + CMC)
MCWS	Number of class with a super type	Number of metaclasses having at least one super type
MGHL	Maximum generalization hierarchical level	Maximum hierarchical depth in the metamodel
MHS	Max Hierarchy Sibling	Maximun number of classes inheriting from a generic superclass
SF	Number of structural features	Number of attributes and references in the metamodel

Table 1. Some of the used metrics for measuring metamodels

Acronym	Name	Description
B	Number of bindings	Number of bindings in all output pattern
IP	Number of Input Pattern	The metric number of input pattern elements measure the size of the input pattern of rules. Note that since called rules do not have an input pattern, the metric number of input model elements does not include called rules.
OP	Number of Output Pattern	The metric number of output pattern elements measure the size of the output pattern of rules.
TR	Number of Transformation Rules	A measure for the size of a model transformation is the number of transformation rules it encompasses. In ATL, there are different types of rules, viz., matched rules, lazy matched rules, unique lazy matched rules, and called rules.
MR	Number of Matched Rules (Excluding Lazy Matched Rules)	Number of matchad rule exzcluding lazy matched rule. If this matrics are equals to number of transformation rule the transformation are defined *completely declarative*
LR	Number of Lazy Matched Rules (Including Unique)	Number of lazy rule including unique
CR	Number of Called Rules	Number of Called Rules
RWF	Number of Rules with a Filter Condition on the Input Pattern	Number of rules with a filter condition on the input pattern. The input pattern has a condition. This implies that not all model elements in the source model may be transformed.
RWD	Number of Rules with a do Section	ATL allows the definition of imperative code in rules in a do block. This can be used to perform calculations that do not fit the preferred declarative style of programming. To measure the use of imperative code in a transformation, we defined number of rules with a do section
RWU	Number of Rules with a using clause	ATL allows the definition of local variable in a rule. This can be used to perform calculations that do not fit the preferred declarative style of programming. To measure the use of imperative code in a transformation, we defined number of rules with a using clause
H	Number of Helper	Number of total helper in the transformation
HWC	Number of Helpers with Context	Number of helper with context in the transformation
HNC	Number of Helpers without Context	Number of helper without context in the transformation
CRT	Number of Calls to resolveTemp()	The resolveTemp() function is used to look-up references to non-default output elements of other rules. Therefore, it is to be expected that model transformations with a large number of calls to the resolveTemp() function are harder to understand.

Table 2. Some of the used metrics for measuring transformations

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
‡ Laboratoire MIS, Université de Picardie Jules Verne, Amiens, France
{Abderrahman.Mokni, Christelle.Urtado, Sylvain.Vauttier}@mines-ales.fr, huchard@lirmm.fr, yulin.zhang@u-picardie.fr

Abstract. This papers gives an overview of our formal approach to address the architecture-centric evolution at the three main steps of component-based software development: specification, implementation and deployment. We illustrate our proposal with an example of software evolution that leads to erosion and we demonstrate how our evolution process can resolve this problem.

Keywords: Software architecture, architecture levels, reuse, software evolution, B formal models

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 component-based software systems is non trivial since an ill-mastered change may lead to architecture inconsistencies and incoherence between design and implementation. Many ADLs (Architecture Description Languages) were proposed to support architecture modeling and analysis. Examples include C2SADL [2], Wright [3] and Darwin [4]. Although, some ADLs integrate architecture modification languages, handling and controlling architecture evolution in the overall software lifecycle is still an important issue. In this paper, we attempt to provide a solution to the architecture-centric evolution that preserves consistency and coherence between architecture levels. We propose an architecture evolution process based on the formal foundations of our three-level ADL Dedal [5]. The process is then illustrated by an evolution scenario on a simplified Home Automation Software architecture. The remainder of this paper is organized as follows: Section 2 briefly discusses examples of existing ADLs. Section 3 gives an overview of Dedal and its formal foundations. Section 4 describes the evolution process based on Dedal. Section 5 presents an evolution scenario that illustrates the proposed evolution process before Section 6 concludes and discusses future work.

2 Existing ADLs

Over the two last decades, a number of ADLs were proposed [6]. Most of them provide textual notations to describe architectural entities (*i.e.* components, interfaces and connections). Initially, ADLs were domain-specific. Examples include C2SADL [2] for the design of concurrent systems and Wright [3] and Darwin [4] for the design and analysis of distributed architectures. Later on, attempts to unify ADLs and make them general-purpose were made. For example, ACME [7] was designed for such purpose. It consists in a common interchange description language that offers annotation facilities to support architecture descriptions in other languages. Another relevant example is xADL 2.0 [8]. It was designed to support various types of systems. The strength of xADL 2.0 resides in its extensibility since it is XML-based. It offers then an easy way for architects to adapt its use to any kind of architectures.

Although a lot of effort was dedicated to improve the expressiveness of ADLs and promote their use to model software architectures, several important issues are not taken into account. First, existing ADLs hardly support all the development steps of component-based software architectures (*i.e.* specification, implementation and deployment). Most of them cover only one or at most two levels which harden their integration in development processes. Second, they hardly support architecture evolution and handle architecture inconsistencies such as drift or erosion [9]. C2SADL and Darwin are exceptions. They include language to describe changes. However, they do not support reverse evolution and they do not cover all steps of component-based development either.

In this work, we address architecture evolution in the whole component-based development process. We show how architectural erosion could be avoided thanks to reverse evolution.

3 Overview of Dedal

3.1 The three architecture levels

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 [10].

To illustrate the three architecture levels of Dedal, we propose an example of a Home Automaton Software (HAS). Figure 1 presents the HAS architecture at three abstraction levels:

The abstract architecture specification (*cf.* Figure 1-a) 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.

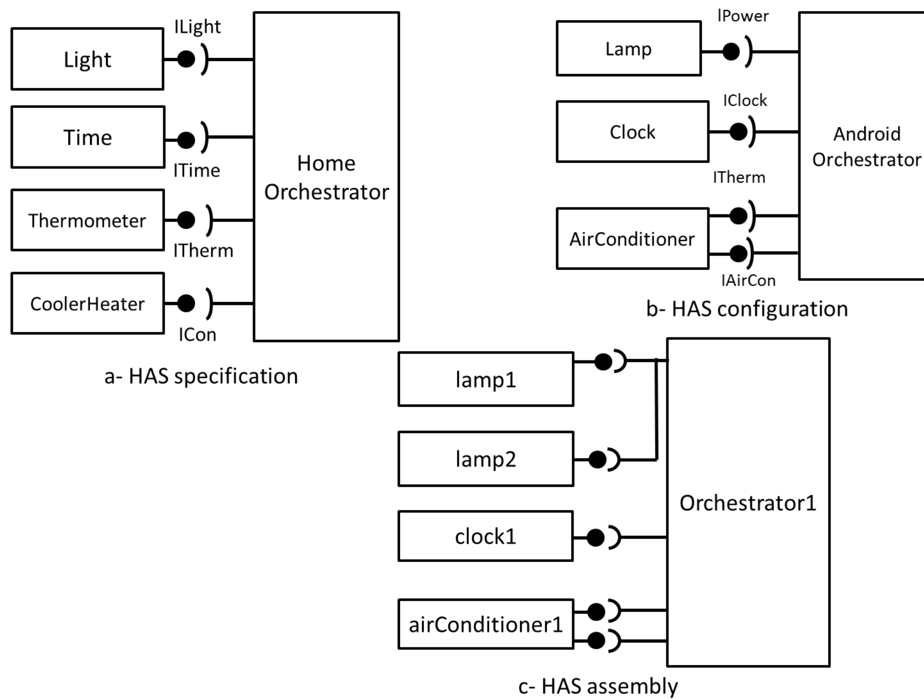


Fig. 1. Illustrative Example

The concrete architecture configuration (*cf.* Figure 1-b) 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 (*cf.* Figure 1-c) 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).

3.2 The formal foundations of Dedal

Dedal is formalized using B [11], a set-theoretic and first order logic formalism. The formalization [12] covers all the concepts of Dedal and includes a set of rules that defines the relations between the different artifacts into and over each architecture level of Dedal (*cf.* Figure 2). These rules are classified into two categories: the intra-level rules and the inter-level rules.

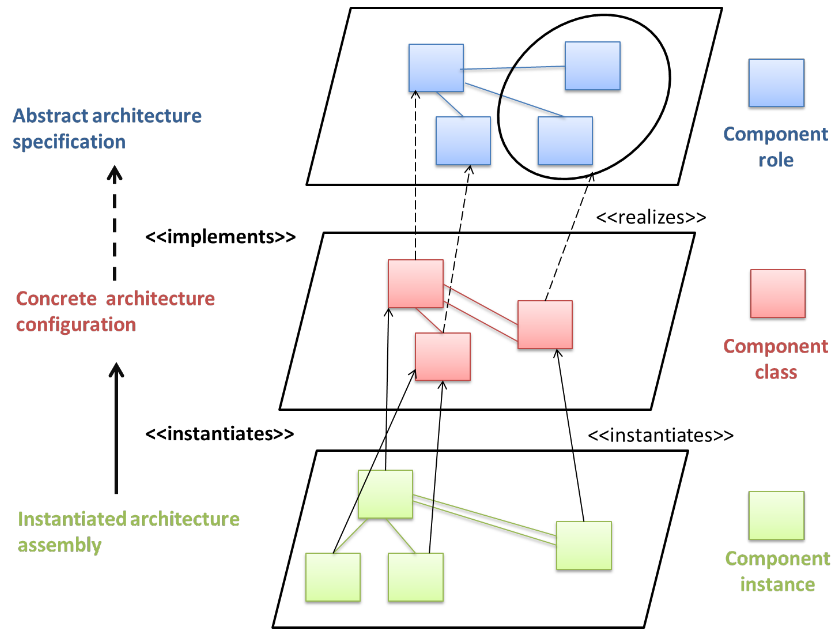


Fig. 2. Inter-level relations in Dedal

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 check the architecture consistency. For instance, the components must be correctly connected to each other (*i.e.* each required interface is connected to a compatible provided one).

Inter-level rules are specific to Dedal and consist in relations between components at different abstraction levels as shown in Figure 2. Defining inter-level rules is mandatory to decide about coherence between two architecture descriptions at different abstraction levels. For instance, the realization rule is used to check that a given configuration is a valid implementation of a given specification and the instantiation rule is used to check if an assembly correctly instantiates a given configuration.

4 Software architecture evolution in Dedal

Handling software evolution in Dedal is quite advantageous. Indeed, Dedal covers the whole life-cycle of software systems and hence all descriptions can be kept up-to-date for further reuse, reimplementation and deployment in different contexts. To keep architecture descriptions coherent, change must be propagated from where it is initiated to the other abstraction levels descriptions. As a solution, we propose an evolution process based on Dedal to enable change in software systems in a manner that preserves architecture consistency and coherence.

The evolution process in Dedal lies on two kinds of rules: (1) static rules to check architecture consistency and coherence between the different descriptions and (2) evolution rules to trigger the change at each abstraction level and propagate it to the other levels descriptions. Figure 3 presents the condition diagram of the evolution process.

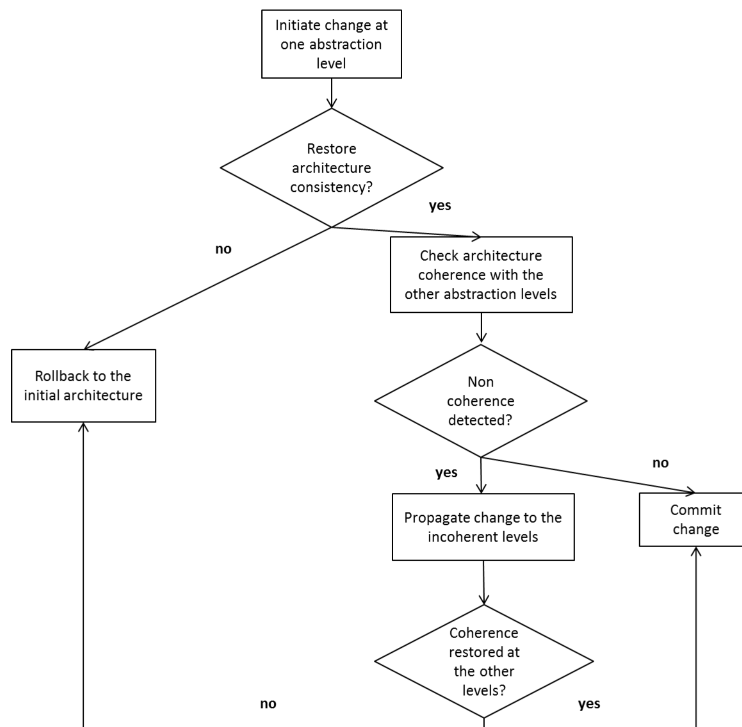


Fig. 3. Condition diagram of the evolution process

4.1 Static rules

Static rules in Dedal are classified into consistency rules and coherence rules. Consistency rules are name uniqueness, completeness, connection correctness and graph consistency. These properties are verified at the same abstraction level by the evolution manager to check weather the architecture is structurally consistent or a change must be triggered to restore consistency. Coherence rules are used to check if software descriptions at the three abstraction levels of Dedal are coherent. If an incoherence is detected, the evolution manager propagates change to the other levels to restore coherence. Coherence rules include the verification of the following properties:

- A configuration *Conf* is an implementation of a given specification *Spec*.
- A specification *Spec* is a documentation of a given configuration *Conf*.
- An assembly *Asm* is an instantiation of a given configuration *Conf*.
- A configuration *Conf* is instantiated by a given assembly *Asm*.

4.2 Evolution rules

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. The following role addition rule is an example of evolution rules at specification level:

```
/* Operation signature takes as arguments an instance of the architecture
    specification(spec) and the instance of the new role(newRole)*/
addRole(spec, newRole) =
/* preconditions */
PRE
spec ∈ arch_spec ∧ newRole ∈ compRole ∧ newRole ∉ spec_components(spec) ∧
/* spec does not contain a role with the same name*/
∀ cr.(cr ∈ compRole ∧ cr ∈ spec_components(spec)
    ⇒ comp_name(cr) ≠ comp_name(newRole))
THEN
/* actions */
/* update the set of clients (required interfaces), the set of
    servers (provided interfaces) and the set of component roles */
        spec_servers(spec) := spec_servers(spec) ∪ servers(newRole) ||
        spec_clients(spec) := spec_clients(spec) ∪ clients(newRole) ||
        spec_components(spec) := spec_components(spec) ∪ {newRole}
END;
```

5 Evolution scenario

5.1 Motivation

To illustrate the evolution approach, we propose an example of evolving the Has architecture. The objective is to enable the control of the house through a mobile

device running under Android OS. The change is initiated at the configuration level and attempts to adapt the current HAS implementation to an android device.

Figure 4-a shows the initial implementation of HAS while Figure 4-b shows the evolved one. Two main changes are noticed in the new configuration: the orchestrator is substituted for a new one compatible with android. Since a new service to control the intensity of lamp is required, the component *Lamp* is replaced by the component *AdjustableLamp* with additional provided interface (*IIntensity*) to adjust the luminosity.

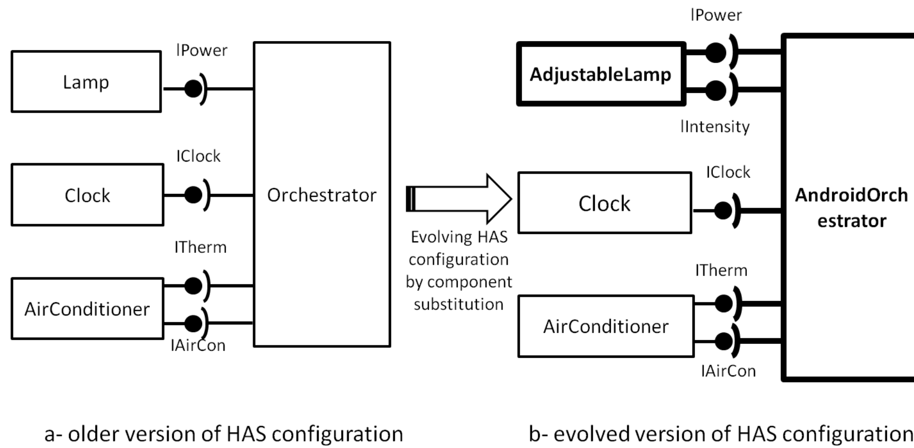


Fig. 4. Evolving the HAS configuration

5.2 Tool support overview

At this stage of work, the evolution process is assisted using ProB [13], an animation tool of B models. We manually instantiate the B formal models corresponding to the Has architecture and execute evolution rules at each abstraction level. We control the evolution process by checking consistency and coherence properties thanks to the evaluation console of ProB. This first step provides a proof feasability of our work. Ongoing work is to automate the generation of Dedal formal models and use the ProB solver to automate the evolution process.

5.3 Evolving the HAS configuration

The change is initiated by disconnecting and deleting the old orchestrator. Then, the one compatible with android is added and connected. The old Lamp is replaced by the adjustable one and connected to the android orchestrator. The evolution manager performs the following operations:

$disconnect(HAS_config, (cl4, rintIPower), (cl1, pintIPower))$
$disconnect(HAS_config, (cl4, rintIClock), (cl2, pintIClock))$
$disconnect(HAS_config, (cl4, rintITherm), (cl3, pintITherm))$
$disconnect(HAS_config, (cl4, rintIAirCon), (cl3, pintIAirCon))$
$deleteClass(HAS_config, cl4)$
$addClass(HAS_config, cl4a)$
$replaceClass(HAS_config, cl1, cl1a)$
$connect(HAS_config, (cl4a, rintIPower2), (cl1a, pintIPower2))$
$connect(HAS_config, (cl4a, rintIIntensity), (cl1a, pintIIntensity))$
$connect(HAS_config, (cl4a, rintIClock2), (cl2, pintIClock))$
$connect(HAS_config, (cl4a, rintITherm2), (cl3, pintITherm))$
$connect(HAS_config, (cl4a, rintIAirCon2), (cl3, pintIAirCon2))$

We note that *cl1*, *cl2*, *cl3* and *cl4* refer respectively to the component classes *Lamp*, *Clock*, *AirConditioner* and *Orchestrator*. *cl1a* and *cl4a* refer respectively to the new component classes *AdjustableLamp* and *AndroidOrchestrator*. Their Interface names are prefixed with *rint*(for a required interface) and *pint*(for provided interface) followed by the interface type name and eventually a number when there is several instances of the same interface

5.4 Propagating change to the HAS specification

The current HAS specification is no longer a good documentation of the new version of the HAS configuration and thus, we have a problem of erosion. Indeed, the control of the light intensity is not included in the current specification. Hence, a new documentation version is required to keep both descriptions coherent. Figure 5 shows the initial and evolved version of the HAS specification after the change propagation.

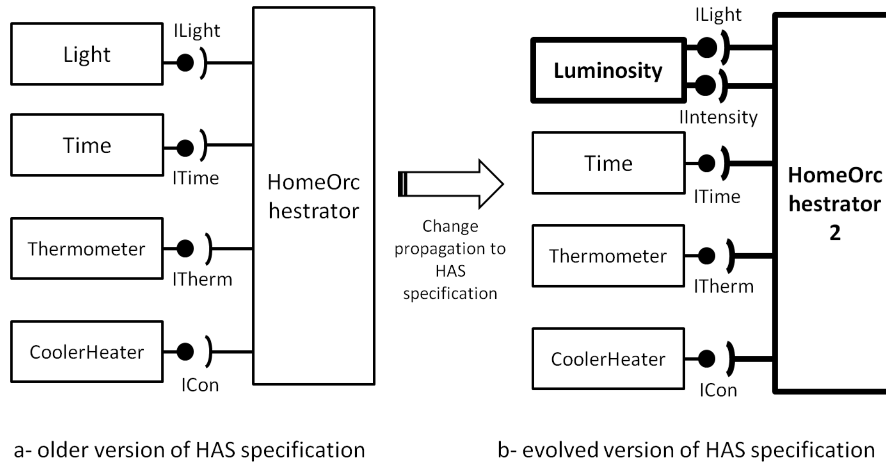


Fig. 5. Evolving the HAS specification by change propagation

The change is propagated to the HAS specification by replacing the role *HomeOrchestrator*(cr4) with *HomeOrchestrator2*(cr4a) and the role *Light*(cr1)

by *Lunminosity*(cr1a). The following operations are performed by the evolution manager:

$disconnect(HAS_spec, (cr4, rintILight), (cr1, pintILight))$
$disconnect(HAS_spec, (cr4, rintITime), (cr2, pintITime))$
$disconnect(HAS_spec, (cr4, rintITherm1), (cr3, pintITherm))$
$disconnect(HAS_spec, (cr4, rintICon), (cr5, pintICon))$
$deleteRole(HAS_spec, cr4)$
$addRole(HAS_spec, cr4a)$
$replaceRole(HAS_spec, cr1, cr1a)$
$connect(HAS_spec, (cr4a, rintILight2), (cr1a, pintILight2))$
$connect(HAS_spec, (cr4a, rintIIntensity), (cr1a, pintIIntensity))$
$connect(HAS_spec, (cr4a, rintITime), (cr2, pintITime))$
$connect(HAS_spec, (cr4a, rintITherm1), (cr3, pintITherm))$
$connect(HAS_spec, (cr4a, rintICon), (cr5, pintICon))$

5.5 Propagating change to the HAS assembly

The current HAS assembly violates the instantiation rule according to the new version of the HAS configuration. This violation is detected by the evolution manager and change is triggered at the assembly level to restore coherence. Figure 6 illustrates the changes applied on the assembly architecture.

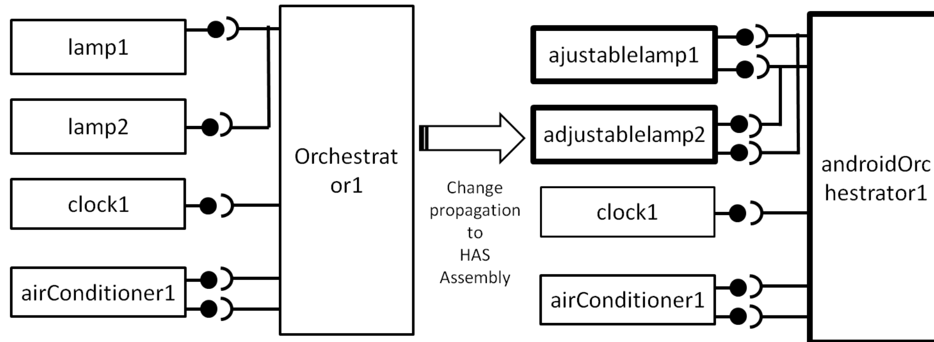


Fig. 6. Evolving the HAS assembly

6 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.

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. Magee, J., Kramer, J.: Dynamic structure in software architectures. In: Proceedings of the 4th ACM SIGSOFT Symposium on Foundations of Software Engineering. SIGSOFT '96, New York, NY, USA, ACM (1996) 3–14
5. Zhang, H.Y., Urtado, C., Vauttier, S.: Architecture-centric component-based development needs a three-level ADL. In: Proceedings of the 4th European Conference on Software Architecture. Volume 6285 of LNCS., Copenhagen, Denmark, Springer (August 2010) 295–310
6. Medvidovic, N., Taylor, R.N.: A classification and comparison framework for software architecture description languages. IEEE Transactions on Software Engineering **26**(1) (January 2000) 70–93
7. Garlan, D., Monroe, R., Wile, D.: Acme: An architecture description interchange language. In: Proceedings of the 1997 Conference of the Centre for Advanced Studies on Collaborative Research. CASCON '97, IBM Press (1997) 7–
8. Dashofy, E., van der Hoek, A., Taylor, R.: A highly-extensible, xml-based architecture description language. In: Software Architecture, 2001. Proceedings. Working IEEE/IFIP Conference on. (2001) 103–112
9. Perry, D.E., Wolf, A.L.: Foundations for the study of software architecture. SIGSOFT Software Engineering Notes **17**(4) (October 1992) 40–52
10. Zhang, H.Y., Zhang, L., Urtado, C., Vauttier, S., Huchard, M.: A three-level component model in component-based software development. In: Proceedings of the 11th GPCE, Dresden, Germany, ACM (September 2012) 70–79
11. Abrial, J.R.: The B-book: Assigning Programs to Meanings. Cambridge University Press, New York, USA (1996)
12. Mokni, A., Huchard, M., Urtado, C., Vauttier, S., Zhang, H.Y.: Towards automating the coherence verification of multi-level architecture descriptions. In: Proceedings of the 9th International Conference on Software Engineering Advances, Nice, France (October 2014)
13. Leuschel, M., Butler, M.: Prob: An automated analysis toolset for the b method. International Journal on Software Tools for Technology Transfer **10**(2) (February 2008) 185–203

Representing Uncertainty in Bidirectional Transformations

Romina Eramo, Alfonso Pierantonio, and Gianni Rosa

DISIM University of L'Aquila, Via Vetoio, 67100, Italy,
name.surname@univaq.it

Abstract. In Model-Driven Engineering, the potential advantages of using bidirectional transformations are largely recognized. The non-deterministic nature of bidirectionality represents a key aspect: i.e, consistently propagating changes from one side to the other is typically non univocal and more than one correct solutions are admitted. In this paper, the problem of uncertainty in bidirectional transformations is discussed. In particular, we illustrate how represent a family of cohesive models, generated as output of a bidirectional transformation, by means of models with uncertainty.

1 Introduction

In Model-Driven Engineering (MDE) [20], 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 [21]. 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 that give place to *non-determinism*. For instance, while MDE requirements demand enough expressiveness to write non-bijective transformations [24], the QVT standard is somewhat uncertain in asserting whether the language permits such transformations [23]. In particular, when reversing a non-injective bidirectional mapping more than one admissible solution might be found. Thus, rather than having a single model, we actually have a set of possible models and we are not sure which is the desired one. On the other hand, while a transformation can always be disambiguated at design-time by fixing those details that leave the solution open to multiple alternatives, in many cases this is impractical because the designer does not detain enough information beforehand for establishing a general solution. Recently, few declarative approaches [4, 17, 3] to bidirectionality have been proposed. They are able to cope with the non-bijectivity by generating all the admissible solutions of a transformation at once. Among them, the Janus Transformation Language [4] (JTL) is a model transformation language specifically tailored to support bidirectionality and change propagation. Unfortunately, managing a set of models explicitly is challenging and poses severe issues as its size might be quite large. The main problem of these approaches is related to the difficulty to manage a number of models.

A similar problem is present in the management of *uncertainty* [12]. Typically it occurs when the designer has not complete, consistent and accurate information required to make a decision during software development. In particular, the designer may add ambiguity during the writing of model transformation. For instance, she may be unsure

about some correspondences among source and target elements and, as a consequence, the transformation may generate multiple solutions each one representing a different design decision. Furthermore, often she can understand it only at execution time, when a multitude of models are obtained.

Providing a representation of the uncertainty generated as outcome of a model transformation process represents a first step to support designers. We propose a metamodel-independent approach to represent a family of cohesive models deriving from a bidirectional transformation by means of models with uncertainty. This paper is related to previous work [11] which introduced the uncertainty due to non-deterministic bidirectional transformations and outlined the challenges aiming to give a support to the problem.

The paper is organized as follows. Section 2 introduces the problem by means of a running example based on JTL. Section 3 discusses uncertainty and the need of a tool support for manage it. Section 4 presents the proposed approach to represent uncertainty. Finally, Sect. 5 describes related work and Sect. 6 draws some conclusion and future work.

2 A motivating example

It is more the rule than the exception that uncertainty is part of almost any aspects of software development [22]. This is also valid for model transformation design and implementation. In particular, in this section we describe how lack of information at design-time can lead to non-deterministic transformations which generates uncertainty in the solution because some mapping between elements in models may be ambiguous.

To better understand the problem and the difficulties it poses, we firstly introduce a well-known application scenario and secondly provide an implementation with JTL highlighting its intrinsic non-determinism.

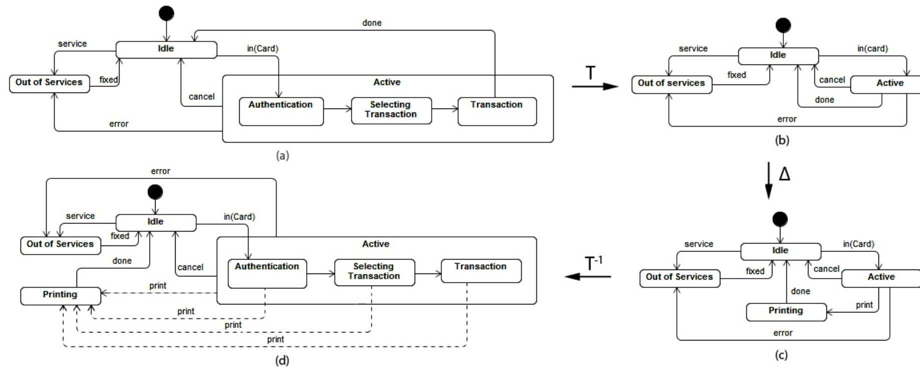


Fig. 1. Collapse/expand state diagrams in a round-trip process

Scenario. Let us consider a typical round-trip problem based on the *Collapse/Expand State Diagrams* benchmark [5]. In particular, starting from a hierarchical state diagram (involving some nesting) as the one reported in Fig. 1(a), the bidirectional transformation yields a flat state machine as provided in Fig. 1(b). A fundamental requirement of the transformation prescribes that manual modifications on the target model must be back propagated to the source model. For instance, suppose that the designer modifies the flattened machine in Fig. 1(b) to produce the model in Fig. 1(c) by:

– adding the new state `Printing`,

– adding the transition `print` that associates state `Active` to the latter, and finally
– modifying the source of the transition `done` from the state `Active` to the state `Printing`.

The expected transformation is clearly non-injective (as different hierarchical machines can be flattened to the same model). In addition, such a model refinement gives place to an interesting situation, i.e., more than one model is admissible (see dotted edges in Fig. 1(d)).

Implementation. The *HSM2SM* bidirectional transformation, which relates hierarchical and flat state machines, has been implemented by means of JTL: a constraint-based model transformation language specifically tailored to support bidirectionality. It adopts a QVT-R[1] like syntax and allows a declarative specification of relationships between MOF models. The semantics is given in terms of Answer Set Programming (ASP) [15], which is a form of declarative programming oriented towards difficult (primarily NP-hard) search problems and based on the stable model (answer set) semantics of logic programming. Then, the ASP solver[2] finds and generates, in a single execution, all the possible models which are consistent with the transformation rules by a deductive process. The JTL environment has been implemented as a set of plug-ins for the Eclipse framework and mainly exploits EMF[3].

```
1 transformation hsm2sm(source : HSM, target : SM) {
2  ...
3  top relation Transition2Transition {
4   enforce domain source sourceTrans: HSM::Transition{
5    owningStateMachine = sourceSM: HSM::StateMachine { },
6   };
7   enforce domain target targetTrans: SM::Transition{
8    owningStateMachine = targetSM: SM::StateMachine { },
9   };
10  when {...}
11  where {...}
12 }
13 relation TransitionSource2TransitionSource {
14  enforce domain source sourceTrans: HSM::Transition {
15   source = sourceState : HSM::State { }
16  };
17  enforce domain target targetTrans: SM::Transition {
18   source = targetState : SM::State { }
19  };
20  when {
21   State2State(sourceState, targetState) and
22   sourceState.owningCompositeState.oclIsUndefined();
23  }
24 }
25 relation TransitionSourceComposite2TransitionSource {
26  enforce domain source sourceTrans: HSM::Transition {
27   source = sourceState : HSM::CompositeState { }
28  };
29  enforce domain target targetTrans: SM::Transition {
30   source = targetState : SM::State { }
31  };
32  when { CompositeState2State(sourceState, targetState); }
33 }  ...
```

Listing 1.1. A fragment of the HSM2SM transformation in JTL

[1] http://www.omg.org/spec/QVT/1.1/
[2] http://www.dlvsystem.com/
[3] http://www.eclipse.org/modeling/emf/

A fragment of the *HSM2SM* transformation is illustrated in List. 1.1. It consists of a number of *relations* defined by the two involved *domains*. In particular, the following relations are reported:

- *Transition2Transition* which relates transitions of the hierarchical metamodel and transitions of the flat metamodel,
- *TransitionSource2TransitionSource* which relates source states of transitions of the hierarchical metamodel and the corresponding source states of transitions of the flat metamodel, and finally
- *TransitionSourceComposite2TransitionSource* which relates source composite states of transitions of the hierarchical metamodel and correspondent source states of transitions of the flat metamodel[4].

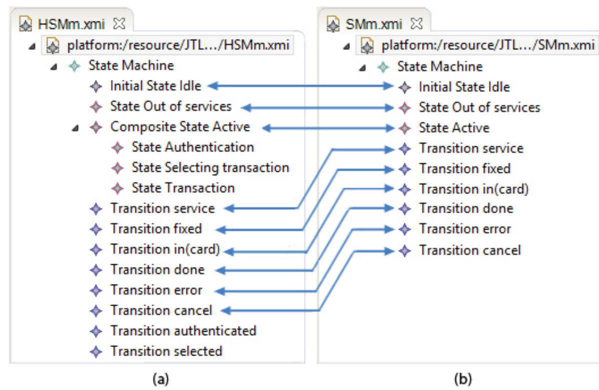


Fig. 2. The HSM model and the correspondent SM model

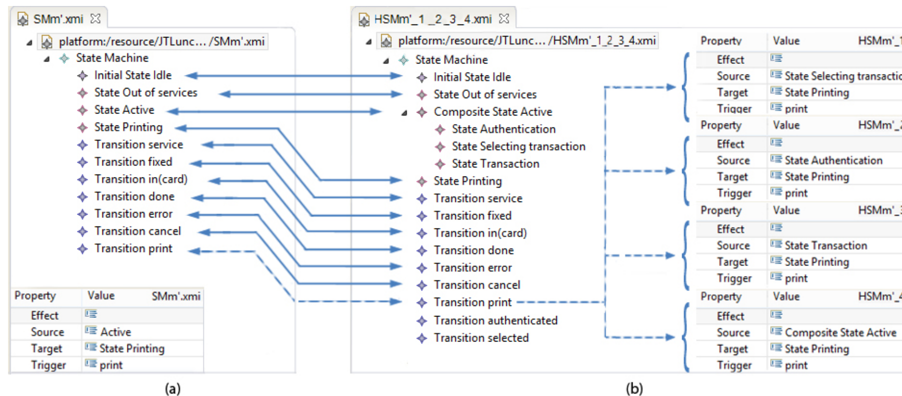


Fig. 3. The modified SM model and the correspondent HSM models

The forward application of the transformation is illustrated in Fig. 2, where the model *HSMm* on the left-hand side is mapped to *SMm* in the right-hand side. As aforementioned the transformation is non-injective. The back propagation of the changes

---

[4] The interested reader can access the full implementation at http://jtl.di.univaq.it/

showed in Fig. 1(c) therefore gives place to the following situation: the newly added transition `print` can be equally mapped to each of the nested states within `Active` as well as to the container state itself, as in Fig. 1(d). In particular, the modified target model *SMm'* in Fig. 3(a) is mapped back to the source models $HSMm'_1, ..., HSMm'_4$ in Fig. 3(b). For example, as visible in the property of the `print` transition, $HSMm'_4$ represents the case in which the transition target goes to the composite state `Active`.

Such non-determinism can still be resolved by accommodating in the transformation the prescription that any new edges in a target model should be mapped to an edge in corresponding source model, such that it *always* refers to the same kind of state, e.g., the container state. This would definitely make the transformation deterministic. However, when the solution cannot be singled out in such a general way, the decision must be left to the modeler in later stages. The potential information erosion have a negative impact on software cost and quality [19]. Thus, designers dealing with model uncertainty need to be supported with suitable mechanism and tools in order to avoid effect of having multiple design alternatives.

3 Uncertainty in modeling

In software engineering decisions have to be made at different stages. Despite the nature of software means making these decisions with absolute confidence, uncertainty appears everywhere [22]. Typically, it occurs when the designer does not have complete, consistent and accurate information required to make a decision during software development. Introducing uncertainty in modeling processes means that, rather than having a single model, we actually have a set of possible models and we are not sure which is the correct one [12]. Thus, handling uncertainty requires the modeler to use this set whenever an individual model would be used. In addition, managing a set of models explicitly is impractical as its size might be quite large. On the other hand, if uncertainty is ignored and one particular possible model is prematurely chosen, we risk having incorrect information in the model. Recently, an approach [12, 14] has been proposed to cope with different aspects of this problem. In particular, the concept of *partial model* has been given in terms of graph theory to capture uncertainty in models. In essence, by means of first-order logic annotations *points of uncertainty* can be introduced in the model, each denoting a possible *concretization*, i.e., a model where the uncertainty is resolved.

Non-bijective model transformations are strictly related to uncertainty, that is introduced during the transformation writing but it is often evident only after the execution, when more than one models may be generated. Especially when the set of generated models is large, designers need to be supported by suitable mechanisms and tools able to manage uncertainty. For this reasons, our aim is to provide the designer a tool for represent the different models into a new one that contains all generated alternatives which abstracts from the calculation method and permits to harness the potential offered by generic modeling platforms such as Eclipse/EMF ([2]]). Furthermore, it represents the starting point for extending a language like JTL semantics and its transformation engine towards an uncertainty-aware solution, capable of dealing with the intrinsic non-determinism of non-bijective transformation in terms of uncertain or partial models.

4 A metamodel-independent approach to uncertainty

Uncertainty as it is known in literature (e.g., [12]) does not have a characterization in terms of metamodels. It is mainly based on annotations which can be processed by tools which are outside, for instance, the EMF ecosystem. In this section, we introduce a metamodel-independent approach to uncertainty representation, i.e., starting from a *base* metamodel M we are interested to understand what are the characteristics of the corresponding metamodel with uncertainty $U(M)$ and how constructively define it.

Since, it has to be used in modeling environment and must be processed by means of automated transformation, in according to our view, we identified a number of natural properties that representation technique should have, as described below

- *model-based*, a set of models representing different alternatives must be represented with a model with uncertainty enabling a range of operations, including analysis or manipulations during the decision process;
- *minimality*, a model with uncertainty is a concise representation of all the alternative solutions; it should not contain any other information besides what needed for representing both the common elements and alternative elements, i.e. alternative designed choices grouped by a point of uncertainty;
- *metamodel-independence*, the metamodel must be agnostic of the base metamodel, i.e., it must be defined in a parametric way such that the definition procedure can be applied in an automated way to any metamodel;
- *interoperability*, each model containing uncertainty must be applicable an unfolding operation, such that whenever applied to it returns all the correspondent concretizations models or the specific concretization selected by the designer.

Starting from these requirements, an automated procedure U : *Ecore* $\rightarrow$ *Ecore* is proposed. The transformation is written in ATL and takes a metamodel M and returns the corresponding metamodel with uncertainty $U(M)$ as described in the rest of the section.

4.1 The uncertainty metamodel

The metamodel with uncertainty is obtained by extending the base metamodel with given connectives to represent the multiple outcomes of a transformation (as showed in Sect. 2). These connectives denote points of uncertainty where different model element are attached. Moreover, such points of uncertainty are traceable in order to ease the traversal of the whole solution space and permit the identification of specific concretizations.

As an example, let us consider *HSM*, the metamodel of the hierarchical state machines given in Fig. 4. Then the corresponding metamodel with uncertainty U(*HSM*) illustrated in Fig. 5 can be automatically obtained as follows:

- the abstract metaclass `TracedClass` with attributes `trace` and `ref` is s added to U(*HSM*);
- for each metaclass *c* in *HSM*, such that it is non-abstract and does not specialize other metaclasses, *i)* a corresponding metaclass *uc* is created in U(*HSM*) such that *uc* specializes *c*, and *ii) c* is generalized by `TracedClass`;
- each metaclass *uc* is composed with *c*, enabling the representation of a point of uncertainty and its alternatives;

– the cardinality of attributes and references derived from *HSM* are relaxed and made optional in $U(HSM)$ in order to permit to express uncertainty also over them.

In particular, the metaclasses `UStateMachine`, `UState` and `UTransition` in $U(HSM)$ derive from `StateMachine`, `State` and `Transition` in *HSM*, whereas the latter ones are generalized by `TracedClass`. The scope of this abstract class is to maintain information about the relationships between the points of uncertainty and the correspondent own alternatives in the concretization models.

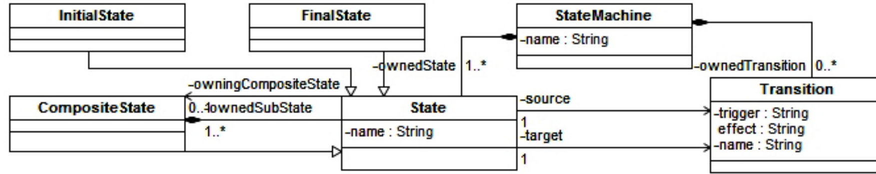


Fig. 4. The *HSM* metamodel

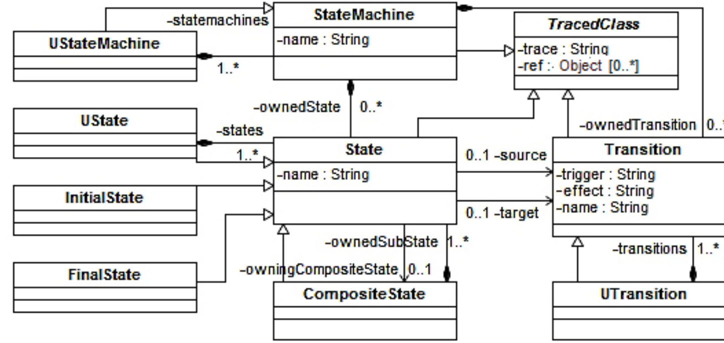


Fig. 5. The $U(HSM)$ metamodel

As said, the above procedure is implemented as an endogenous model transformation in ATL. For the sake of brevity, only an excerpt of the transformation is presented in Listing 1.2[5] containing solely those rules which build the specific constructions of the uncertainty metamodel. More in detail, the rule `EClass2UEClass` (lines 4-20) *a)* propagates base metaclasses (and their associations) which are not abstract and do not have non-abstract ancestors (lines 5-6) and *b)* for each of them generates a corresponding uncertainty metaclass (lines 11-14), as in Fig. 5 where the metaclass `UState` is generated and composed with `State`. Moreover, the target pattern (lines 7-20) is composed of a set of elements, each of them specifies a target type from the target metamodel and a set of bindings. In particular, the element `t` (lines 7-10) copies the metaclass `s` in the target metamodel; the element `u` of the target pattern (lines 11-14) generates uncertainty metaclass as specialization of the matched source class `s`; and finally, the reference `r` is created as a structural feature of the element `u` in order to refer alternative elements contained in `u`.

```
1 module MM2UMM;
2 create OUT : UMM from IN : MM;
```

[5] The *MM2UMM* transformation implementation is available at http://jtl.di.univaq.it/

```
3 [...]
4 rule EClass2UEClass {
5     from s : MM!EClass ((thisModule.inElements->includes(s)) and
6         ((s.eSuperTypes->size()=0) or (s."abstract"='false')))
7     to t : UMM!EClass (
8         name <- s.name,
9         eSuperTypes <- s.eSuperTypes->append(thisModule.traceableMetaclass),
10        ...),
11    u : UMM!EClass (
12        name <- 'U'+s.name,
13        eReferences <- Sequence{}->append(r),
14        eSuperTypes <- Sequence{}->append(s)),
15    r:UMM!EReference(
16        name <- s.name + 's',
17        containment <- true,
18        lowerBound <- 1,
19        upperBound <- -1)
20 }
21 [...]
```

Listing 1.2. A fragment of the MM2UMM transformation

To better understand how a point of uncertainty is realized, please consider the alternative solutions in the right-hand side of Fig. 3 and how they are denoted by the corresponding point of uncertainty illustrated in Fig. 5. In particular, the alternative transitions are collected in a point of uncertainty (`UTransition`) which contains the transitions `print` targeting each one of the nested states within `Active` as well as to the composite state itself.

It is important to notice that models with uncertainty may be an *over-approximation* of the sets of transformation candidates. This is due to the "combinatorial" nature of these models since each point of uncertainty collects the different alternatives. Consequently, it can happen that certain combinations produce concretizations which are not part of the solution space. For instance, in the scenario in Fig. 1, probably only one `print` transition can exist in the final model. However, the generated model with uncertainty admits models with multiple `print` transitions giving place to more concretizations than those expected. Therefore, besides the models with uncertainty it is important to generate also those constraints which limit the solution to the admissible concretization, in our case, in order to avoid multiple `print` transitions, the model with uncertainty in Fig. 5 is augmented with a constraint that reduces the concretizations to cases with one `print` transition only. According to the uncertainty metamodel described in Sect. 4, each metaclass provide an attribute `ref` to maintain the reference to the corresponding concretization(s).

5 Related work

Uncertainty is one of the factors prevalent within contexts as requirements engineering [7], software processes [16] and adaptive systems [18]. Uncertainty management has been studied in many works, often with the aim to express and represent it in models. In [12] *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. Model transformation techniques typically operate under the assumption that models do not contain uncertainty. Nevertheless, the work in [13] proposes a technique for adapting existing model transformations in order to deal with models containing uncertainty. The main is a lifting operation which permits to adapt unidirectional transformations for being used over models with uncertainty preserving their original behavior.

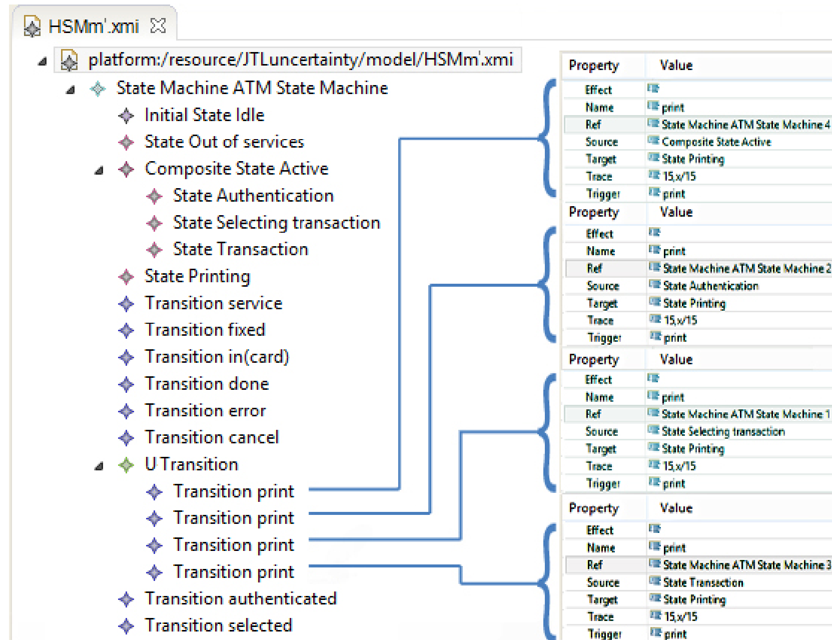


Fig. 6. UHSMm model

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 [1, 23, 6] . The ability to deduce and generate all the possible solutions of an uncertain transformation has been achieved by few approaches, including JTL [8, 10]. In such case, may be useful to manage non-determinism during the design process in order to detect ambiguities and support designers in solving non-determinism in their specification as faced in [9].

6 Conclusion

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. S. M. Becker, S. Herold, S. Lohmann, and B. Westfechtel. A graph-based algorithm for consistency maintenance in incremental and interactive integration tools. *SOSYM*, 2007.
2. J. Bézivin, F. Jouault, P. Rosenthal, and P. Valduriez. Modeling in the Large and Modeling in the Small. In *Procs of European MDA Workshops*, 2004.
3. G. Callow and R. Kalawsky. A Satisficing Bi-Directional Model Transformation Engine using Mixed Integer Linear Programming. *JOT*, 12(1):1: 1–43, 2013.
4. A. Cicchetti, D. Di Ruscio, R. Eramo, and A. Pierantonio. JTL: a bidirectional and change propagating transformation language. In *SLE10*, pages 183–202, 2010.
5. K. Czarnecki, J. N. Foster, Z. Hu, R. Lämmel, A. Schürr, and J. F. Terwilliger. Bidirectional Transformations: A Cross-Discipline Perspective - GRACE meeting. In *Procs. of ICMT2009*.
6. Z. Diskin, Y. Xiong, and K. Czarnecki. From state- to delta-based bidirectional model transformations. pages 61–76, 2010.
7. C. Ebert and J. D. Man. Requirements uncertainty: influencing factors and concrete improvements. In *Procs. of ICSE*, pages 553–560. ACM Press, 2005.
8. R. Eramo, I. Malavolta, H. Muccini, P. Pelliccione, and A. Pierantonio. A model-driven approach to automate the propagation of changes among Architecture Description Languages. *SOSYM*, 1(25):1619–1366, 2010.
9. R. Eramo, R. Marinelli, A. Pierantonio, and G. Rosa. Towards Analyzing Non-Determinism in Bidirectional Transformations. In *Procs. of AMT 2014*, 2014.
10. R. Eramo, A. Pierantonio, J. R. Romero, and A. Vallecillo. Change management in multi-viewpoint system using asp. In *EDOCW08*, pages 433–440. IEEE Computer Society, 2008.
11. R. Eramo, A. Pierantonio, and G. Rosa. Uncertainty in bidirectional transformations. In *Procs. of MiSE 2014*, 2014.
12. M. Famelis, R. Salay, and M. Chechik. Partial models: Towards modeling and reasoning with uncertainty. In *ICSE*, pages 573–583, 2012.
13. M. Famelis, R. Salay, A. D. Sandro, and M. Chechik. Transformation of models containing uncertainty. In *MoDELS*, pages 673–689, 2013.
14. M. Famelis and S. Santosa. Mav-vis: A notation for model uncertainty. In *MiSE*, 2013.
15. M. Gelfond and V. Lifschitz. The Stable Model Semantics for Logic Programming. In *Procs of ICLP*, pages 1070–1080, Cambridge, Massachusetts, 1988. The MIT Press.
16. 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.
17. N. Macedo and A. Cunha. Implementing QVT-R Bidirectional Model Transformations Using Alloy. In *FASE*, pages 297–311, 2013.
18. 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.
19. B. Schätz, F. Hölzl, and T. Lundkvist. Design-space exploration through constraint-based model-transformation. In *ECBS*, pages 173–182, 2010.
20. D. Schmidt. Guest Editor's Introduction: Model-Driven Engineering. *Computer*, 39(2):25–31, 2006.
21. S. Sendall and W. Kozaczynski. Model Transformation: The Heart and Soul of Model-Driven Software Development. *IEEE Software*, 20(5):42–45, 2003.
22. A. Sillitti, M. Ceschi, B. Russo, and G. Succi. Managing uncertainty in requirements: A survey in documentation-driven and agile companies. In *IEEE METRICS*, page 17. IEEE Computer Society, 2005.
23. P. Stevens. Bidirectional model transformations in QVT: semantic issues and open questions. *SOSYM*, 8, 2009.
24. S. Witkop. MDA users' requirement for QVT transformations. In *OMG doc 05-02-04*, 2005.

Towards a Taxonomy for Bidirectional Transformation

Romina Eramo, Romeo Marinelli, and Alfonso Pierantonio

Dipartimento di Ingegneria e Scienze dell'Informazione e Matematica (DISIM),
Università degli Studi dell'Aquila, Italy
name.surname@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 their semantic implications hampering their applicability in practice. This paper illustrates a set of relevant properties pertaining to bidirectional model transformations. It is a first step towards a taxonomy that can help developers to decide which bidirectional language or tool is best suited to their task at hand. This study is based on the existing literature and characteristics of existing approaches.

1 Introduction

Bidirectionality is an important feature of 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 [31, 37]. In this context, bidirectional model transformations have arrived as a key mechanism, in fact they describe not only a forward transformation from a source model to a target model, but also a backward transformation that reflects the changes on the target model to the source model so that consistency between two models is maintained.

Many typical situations, that arise when both models are modified by humans, demand bidirectional transformations; for instance, a conceptual model is transformed into a platform specific model; a subview of selected data is modified and contextually the entire database is updated; many kinds of integration between systems or parts of systems, which are modeled separately, must be consistent (e.g., a database schema must be kept consistent with the application that uses it) [31]. Hence, bidirectional transformations have many potential applications in software development, including model synchronization, round-trip engineering, software evolution by keeping different models coherent to each other, multiple-view software development.

In this paper we propose a taxonomy for bidirectional model transformations. It represent a first step towards a complete set of objective criteria, that designers may consider in order to select the approach that is more suitable for their needs. The study is based on the discussions of the working group of the Dagstuhl seminars on Language Engineering for Model-Driven Software Development [6, 20] and the existing taxonomies for model transformations (e.g., [23, 7]).

The paper is organized as follows. Section 2 introduces the background. Section 3 proposes a taxonomy of bidirectional model transformations. Section 4 discusses how this taxonomy can be applied on existing approaches. Finally, Section 5 describes related work and Section 6 draws some conclusion and future work.

2 Background

Despite its relevance, bidirectionality has rarely produced anticipated benefits as demonstrated by the lack of a language comparable to what ATL[1] represents for unidirectional transformations. However, a number of languages and tools have been proposed due to the intrinsic complexity of bidirectionality; each one is characterized by a set of specific properties pertaining to a particular applicative domain [32].

The attempt to advocate a language rather than another can not neglect the important differences in circumstances. In fact, each language or tool presents specific characteristics that make it more suitable for a certain circumstance; as a consequence, a solution which seems good with one set of assumptions made about the situation can be infeasible with another. Understanding these characteristics is useful for distinguishing between existing approaches; in particular, in order to choose the most suitable approach, designers need to know how factors, that may influence their choice, affect bidirectional approaches.

Several works have been proposed for defining characterization and classification schemes of model transformation, others illustrate the state-of-the-art. Many of the proposed aspects characterizing model transformations are considered in this study, many other aspects need to be explored to shift the focus on intrinsic characteristics of bidirectional model transformations. Taking inspiration from [23], we consider some fundamental issues as following:

- What needs to be transformed into what;
- Which mechanisms can be used for bidirectional transformation;
- What are the application domains;
- What are the characteristics of a bidirectional transformation;
- What are the quality requirements for a bidirectional language or tool;
- What are the success criteria for a bidirectional language or tool.

These issues demand for the definition of pertaining concepts, terms and base characteristics for bidirectional transformation. To this end, in the next section a taxonomy for bidirectional transformation is proposed to provide specific and intrinsic features and requirements.

3 Taxonomy

The proposed taxonomy is based on a set of features that are divided into three main categories: General Requirements (GR), Functional Requirements (FR) and Non Functional Requirements (NFR). They are described in detail in the following.

[1] http://www.eclipse.org/atl/

3.1 General Requirements (GR)

This section defines general requirements that concerns generic and important aspects for bidirectional model transformations. These requirements are useful to understand what are some important characteristics of a bidirectional approach or tool.

Complexity. Transformations can be considered as small (e.g., refactoring), or heavy-duty (e.g., compilers and code generators) [23]. The large difference among them requires an entirely different set of tools and techniques. Considering the complexity of transformations is out of our scope; it demands for more effort and can be treated, for instance, by using metrics.
Level of automation. A bidirectional transformation can be performed in a completely automatic way (*fully-automated*), otherwise it may need to a certain amount of manual intervention (*human-in-the-loop*). In particular, manual intervention is needed to address and resolve ambiguity, incompleteness and inconsistency in the requirements, that may be partially expressed in natural language [23].
Visualization. It refers to the way in which models, metamodels and model transformations are presented to the user; it can be *graphical* or *textual*. Some existing tools allow developers to create artifacts in a completely graphical way (e.g., [28]), others require transformations are written entirely in textual way (e.g., [3]).
Level of industry application. Some existing tools are used in both academic and industrial world (e.g., [28]), others are used exclusively in academic setting (e.g., [5]), in which the usefulness is due to particular issues related to the bidirectionality.
Maturity level. The main existing approaches for bidirectional transformations are implemented by means a tool (*practical approaches*), often available on internet (e.g., [28, 26]). However, other approaches, despite the theoretical development (e.g., [9, 10]) have not been yet implemented by meas of a tool (*theoretical approaches*).

3.2 Functional Requirements (FR)

Functional requirements refer to the product capabilities and define how a bidirectional approach behaves to meet user needs. These requirements define important characteristics of a bidirectional approach that contribute to the success of such a language or tool. Furthermore, part of these requirements concerns the source and target artifacts of the transformations and the mechanisms that can be used.

Correctness. The simplest notion of correctness is the *syntactic correctness*: given a well-formed source model, it must guarantee that the target model produced by the transformation is wellformed. A significantly more complex notion is the *semantic correctness*: does the produced target model must have the expected semantic properties [23]. In other words, if the target model conforms to the target metamodel specification and wellformednes rules, then the model transformation is syntactically correct; whereas if the model transformation preserves the behavior of the source model, then it is semantically correct [11].
Inconsistency management. It concerns the ability to deal with incomplete or inconsistent artifacts. In fact, in the early phases of the software development life-cycle, requirements may not yet be fully understood; this often gives rise to ambiguous, incomplete

or inconsistent specifications, demanding for mechanisms for inconsistency management. These mechanisms may be used to detect inconsistencies in the transformation or in transformed models [23, 22, 24].

Modularity. It is the ability to compose existing transformations into new composite ones. Decomposing a complex transformation into small steps may require mechanisms to specify how these smaller transformations are combined [29, 4].

Traceability. It is the property of having a record of links between the source and target elements of a transformation as well as the various stages of the transformation process. Traceability links can be stored either in the target model or separately [29, 4, 8]. To support it, tools need to provide mechanisms to maintain an explicit link between the source and target models [23, 22, 24].

Change Propagation. Bidirectional approaches have to correctly propagate changes occurring in models from a direction to another. In order to support change propagation, bidirectional approaches may provide mechanism for an incremental updates or consistency management. Unfortunately, some approaches require to translate the source model into some standardized format (e.g., XML) before the transformation execution, and to translate again to obtain the target model. A clear disadvantage of such an approach concerns the difficult to synchronize models when changes are made [23, 22, 24]. Whereas, [35] propose a change propagating transformation language that supports the preservation of target changes by back propagating them toward the source. On the one hand, conflicts may arise each time the generated target should be merged with the existing one; on the other hand, the back propagation poses some problems related to the invertibility of transformations.

Incrementality. It refers to the ability to update existing target models on the base on changes made in the source models, and vice versa [7]. In particular, incremental transformations synchronize two models by propagating modifications such that information not covered by the transformation can be preserved and the computational effort can be minimized. In contrast, a classical batch transformation synchronizes two models taking a source model as input and computes the resulting target model from scratch, and vice versa [17]. This property can be achieved, for example, by using traceability links. When any of the source models are modified and the transformation is executed again, the necessary changes to the target are determined and applied. At the same time, the target elements may be preserved. In general, incremental transformations are able to avoid loss of information (e.g., elements discarded by the mapping).

Uniqueness. It refers to the number of solutions generated by the bidirectional approach [4]. If the transformation is non-deterministic, it may exist more than one way to keep models in a consistent way. Most of the existing tools generate a single solution. Some recent approaches are able to generate all the possible solutions according to a non-deterministic specification (e.g., [5]).

Termination. A model transformation provides termination, if it always terminates and leads to a result [4].

Symmetric/Asymmetric behavior. Transformations are asymmetric when they work between a concrete set of models and a strict abstraction of this, and the abstract set of models contains strict less information [31]. For instance, [15, 36] are asymmetric trans-

formations since two models that need to be synchronized must be one an abstraction of the other. On the contrary, the transformation is symmetric.

Type of Artifact. The distinction concerns the kinds of artifacts being transformed [23]. *Program transformation*s involve programs (i.e., source code, bytecode, or machine code); *Model transformation*s involve software models. According to [23], the latter term encompasses the former since a model can range from abstract analysis models, over more concrete design models, to very concrete models of source code. Hence, model transformations also include transformations from a more abstract to a more concrete model (e.g., from design to code) and vice versa (e.g., in a reverse engineering context). Model transformations are obviously needed in common tools such as code generators and parsers.

Data Model. It refers to the way in which data are represented into the tool. In particular, data can be represented by means of a graphs or trees.

Endogenous/Exogenous transformations. Models need to be expressed in some modeling language (e.g., UML for design models, or programming languages for source code models). The syntax and semantics of the modeling language itself is expressed by a metamodel (e.g., the UML metamodel). Based on the language in which the source and target models of a transformation are expressed, a distinction can be made between endogenous and exogenous transformations [23]. *Endogenous transformations* are transformations between models expressed in the same language. *Exogenous transformations* are transformations between models expressed using different languages.

Transformation Mechanisms. The major distinction between transformation mechanisms is whether they rely on a *declarative* or an *operational* (or *imperative*) approach. Declarative approaches focus on the what aspect, i.e., they focus on what needs to be transformed into what by defining a relation between the source and target models. Operational approaches focus on the how aspect, i.e., they focus on how the transformation itself needs to be performed by specifying the steps that are required to derive the target models from the source models. Declarative approaches (e.g., [1]) are attractive because particular services such as source model traversal, traceability management and automatic bidirectionality can be offered by an underlying reasoning engine.

There are several aspects that can be made implicit in a transformation language: (i) navigation of a source model, (ii) creation of target model and (iii) order of rule execution. As such, declarative transformations tend to be easier to write and understand by software engineers. Operational (or constructive) approaches (e.g., [30]) may be required to implement transformations for which declarative approaches fail to guarantee their services. Especially when the application order of a set of transformations needs to be controlled explicitly, an imperative approach is more appropriate thanks to its built-in notions of sequence, selection and iteration. Such explicit control may be required to implement transformations that reconcile source and target models after they were both heavily manipulated outside that transformation tool.

Declarative approaches include, but are not limited to, all of the following approaches: functional programming, logic programming and graph transformation.

- *Functional programming*. Such an approach towards model transformation is appealing, since any transformation can be regarded as a function that transforms some input (the source model) into some output (the target model). In most func-

tional languages, functions are first class, implying that transformations can be manipulated as models too. An important disadvantage of the functional approach is that it becomes awkward to maintain state during transformation.
- *Logic programming*. A logic language (e.g., Prolog or Mercury) has many features that are of direct interest for model transformation: backtracking, constraint propagation (in the case of constraint logic programming languages), and unification. Additionally, logic languages always offer a query mechanism, which means that no separate query language needs to be provided.
- *Graph transformation*. It is a set of techniques and associated formalisms that are directly applicable to model transformation [16]. It has many advantages. It is a visual notation: the source, target and the transformation itself can be expressed in a visual way. It offers mechanism to compose smaller transformations into more complex ones.

In-place/Out-of-place transformations. If the number of involved models is only one, the source and target model are the same and all changes are made *in-place*. Other endogenous transformations create model elements in one model based on properties of another model (regardless of the fact that both models conform to the same metamodel). Such transformations are called *out-place* [24]. Note that exogenous transformations are always out-place.

3.3 Non Functional Requirements (NFR)

Non functional requirements may be considered as the quality attributes for a bidirectional transformation.
Extensibility and Modifiability. The extensibility of a tool refers to the ease in which it can be extended with new features. The modifiability of an artifact refers the ability of a bidirectional transformation to be modified and adapted to provide different or additional features [25].
Usability and Utility. The language or tool should be useful, which means that it has to serve to a practical purpose. On the other hand, it has to be usable too, which means that it should be intuitive and efficient to use [23].
Scalability. It is the ability to cope with large and complex transformations or transformations of large and complex software models without sacrificing performance [23].
Robustness. It is the ability to manage invalid models. If unexpected errors are handled and invalid source models are managed, then the approach provides robustness [4].
Verbosity and Conciseness. *Conciseness* means that the transformation language should have as few syntactic constructs as possible. From a practical point of view, however, this often requires more work to specify complex transformations. Hence, the language should be more *verbose* by introducing extra syntactic sugar for frequently used syntactic constructs. It is always a difficult task to find the right balance between these two conflicting goals [23, 25].
Interoperability. It refers to the ability of a tool to integrate itself with other tools, used within the (model-driven) software engineering process [25]. Sometimes designers wish to use and interchange models among different modeling tools and modeling languages. A typical example is the translation of some tool-specific representation of UML models

into XMI (and vice versa), OMG's XML-based standard for model interchange. This facilitates exchanging UML models between different UML modeling tools [22].

Reference Platform (Standardization). It indicates whether the transform tool is compliant to all the relevant standards (e.g., XML, UML, MOF) [25]. Many tools can export and import models in a standard form, typically XML; an external tool can then take the exported model and transform it [29]. Most of them are implemented within EMF [5, 12, 13, 21]. Others adopt an algebraic approach [18, 3] and work on files or data strings, which are not directly integrated with the environment EMF.

Verifiability and validity. It concerns the ability to test, verify and validate models and transformations. Since transformations can be considered as a special kind of software programs, systematic testing and validation techniques can be applied to them to ensure that they have the desired behavior [23, 4]. Verification of (sets) of model transformations is needed to assure that they produce well-formed and correct models, and preserve (or improve) desirable properties such as (syntactical or semantical) correctness, consistency, and so on [22].

Existing approaches and tools involve testing (e.g., test case generation for model transformations), model checking, or analysis performed only on executing programs (e.g., run-time monitoring) [2]. In contrast, [2] and [14] propose an approach to analyze model transformation by means of a logic environment, able to verify if the transformation satisfy certain properties (correctness, well-formedness, non-determinism, etc.). Furthermore, existing functional approaches perform model transformation analysis [18, 27], in order to evaluate the transformation validity (generally, in terms of correctness) before its execution.

4 Applications

The proposed taxonomy can be applied for assessing characteristics of the existing bidirectional approaches. As already said, each language or tool provides different characteristics, and designers can choose the desired approach by using the set of evaluation criteria proposed in this taxonomy. The application of this taxonomy on the existing approaches aims to emphasize strengths and weaknesses.

The most common paradigms used in the existing approaches are the declarative and the functional paradigms. Among the existing declarative approaches, we are interested to consider: (i) TGGs [28], a bidirectional languages based on graph grammars; (ii) QVT-Relations [26], a declarative bidirectional language part of the QVT standard; (iii) JTL [5], a constraint-based bidirectional transformation language. Moreover, we are interested in considering some functional approaches, that are: (i) Lenses [3], an asymmetric bidirectional programming language between a concrete structure and a correspondent abstract view, (ii) GRound-Tram [18], a functional language-based modeling framework; (iii) BiFlux [27], an integrated modeling framework for developing bidirectional model transformations based on graph query language.

The proposed taxonomy represents a a preliminary work; in fact, the real benefit is represented by the result of the application over the existing work. The application of the selected features will highlight weaknesses and criticality of the bidirectional approaches .

5 Related Work

This work is essentially based on the existing works which propose taxonomies and characterizations for model transformation and bidirectionality.

Important aspects of bidirectional transformations have been discussed during the working group of the Dagstuhl seminars on Language Engineering for Model-Driven Software Development in 2005 and 2011 [6, 20]. In general, model transformations can be characterized by different orthogonal concerns (see [7] for a detailed classification). Czarnecki and Helsen in [7] present a survey of model transformation techniques with a particular emphasis on rule-based approaches such as those based on graph transformations. They mention directionality, but do not focus on it. In [23] the authors propose a taxonomy for model transformation, in particular they consider functional requirements that contribute to the success of the tool or language and non-functional requirements or quality requirements. In [24], the taxonomy is applied to graph transformations, in particular AGG [33] and Fujaba[2]. In [22] the same taxonomy is presented again, but emphasizing the importance of other concepts such as composition, interoperability and bidirectionality. [25] proposes a taxonomy based only on non-functional features, referring to languages and artifacts. [29] puts emphasis on standardization and languages for model transformation. [4] considers existing taxonomies and proposes a set of most important features. Finally, [34] performs a comparison among existing taxonomies.

The above mentioned works consider general model transformation. A specific taxonomy for bidirectional transformations has not yet been proposed, however there have been several works analyzing characteristics and semantic issues of bidirectional model transformations. Among them, in [32] the QVT-R bidirectional transformation language is illustrated and semantic issues and open questions about bidirectionality are discussed. [31] explores the landscape of bidirectional model transformations until the 2007. [19] proposes a survey on TGGs tools.

6 Conclusion

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. This paper proposed a set of relevant properties pertaining bidirectional model transformations. It is based on the existing literature and the characteristics of existing approaches. This work aims to represent a first step towards a taxonomy that can be used, among others, to help developers in deciding which bidirectional transformation language or tool is more suitable for different types of tasks. In order to do this, as future work, we plan to extend the taxonomy with other features for bidirectionality and apply it to existing languages and tools for bidirectionality.

[2] http://www.fujaba.de/

References

1. D. H. Akehurst and S. Kent. A Relational Approach to Defining Transformations in a Metamodel. In *Procs of the 5th Int. Conf. on The UML*, pages 243–258. Springer-Verlag, 2002.
2. K. Anastasakis, B. Bordbar, and J. M. Küster. Analysis of model transformations via Alloy. In *ModeVVa'07*, pages 47–56, 2007.
3. A. Bohannon, J. N. Foster, B. C. Pierce, A. Pilkiewicz, and A. Schmitt. Boomerang: Resourceful lenses for string data. In *Proc. of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL 2008, pages 407–419, 2008. http://www.seas.upenn.edu/ harmony/.
4. D. Cetinkaya and A. Verbraeck. Metamodeling and Model Transformations in Modeling and Simulation. In *Proceedings of the Winter Simulation Conference*, WSC '11, pages 3048–3058, 2011.
5. A. Cicchetti, D. Di Ruscio, R. Eramo, and A. Pierantonio. JTL: A Bidirectional and Change Propagating Transformation Language. In *3rd Int. Conf. on Software Lang. Engineering (SLE)*, SLE 2010. http://jtl.di.univaq.it.
6. K. Czarnecki, J. N. Foster, Z. Hu, R. Lämmel, A. Schürr, and J. F. Terwilliger. Bidirectional Transformations: A Cross-Discipline Perspective–GRACE meeting. In *ICMT2009*, volume 5563 of *LNCS*, pages 260–283. Springer, 2009.
7. K. Czarnecki and S. Helsen. Feature-based Survey of Model Transformation Approaches. *IBM Systems J.*, 45(3), 2006.
8. M. Dehayni, K. Barbar, A. Awada, and M. Smaili. Some model transformation approaches: a qualitative critical review. *Journal of Applied Sciences Research*, 5(11):1957–1965, 2009.
9. Z. Diskin, Y. Xiong, and K. Czarnecki. From state-based to delta-based bidirectional model transformation. In *3rd Int. Conf. on Model Transformation*. Springer, 2010.
10. Z. Diskin, Y. Xiong, and K. Czarnecki. From state- to delta-based bidirectional model transformations: the asymmetric case. In *Journal of Object Technology*, volume 10, 2011.
11. H. Ehrig and C. Ermel. Semantical correctness and completeness of model transformations using graph and rule transformation. In *ICGT*, pages 194–210, 2008.
12. EMoflon. http://www.moflon.org/.
13. EMorF. http://www.emorf.org.
14. R. Eramo, R. Marinelli, A. Pierantonio, and G. Rosa. Towards analysing non-determinism in bidirectional transformations. In *Procs. of AMT 2014*, 2014.
15. 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.
16. A. Gerber, M. Lawley, K. Raymond, J. Steel, and A.Wood. Transformation: The Missing Link of MDA. In *1st Int. Conf. on Graph Transformation*, pages 90–105, 2002.
17. H. Giese and R. Wagner. From model transformation to incremental bidirectional model synchronization. *Software and Systems Modeling*, 8(1):21–43–43, 2009.
18. S. Hidaka, Z. Hu, K. Inaba, H. Kato, and K. Nakano. Groundtram: An integrated framework for developing well-behaved bidirectional model transformations. In *ASE 2011*, 2011. http://www.biglab.org/.
19. 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 *BX 2013*, volume 57, pages 1–18. EC-EASST, 2013.
20. Z. Hu, A. Schürr, P. Stevens, and J. F. Terwilliger. Bidirectional transformation "bx" (dagstuhl seminar 11031). *Dagstuhl Reports*, 1(1):42–67, 2011.
21. MediniQVT. http://projects.ikv.de/qvt/.

22. 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.
23. T. Mens, K. Czarnecki, and P. V. Gorp. A Taxonomy of Model Transformations. In *Language Engineering for Model-Driven Software Development*, volume 04101 of *Dagstuhl Seminar Proceedings*, 2004.
24. 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, 2006.
25. S. Nalchigar, R. Salay, and M. Chechik. Towards a Catalog of Non-Functional Requirements in Model Transformation Languages. In *AMT @ MoDELS*, 2013.
26. Object Management Group (OMG). MOF QVT Final Adopted Specification, 2005. OMG Adopted Specification ptc/05-11-01.
27. H. Pacheco and Z. Hu. Biflux: A Bidirectional Functional Update Language for XML. In *BIRS workshop: Bidirectional transformations (BX)*, 2013. http://www.prg.nii.ac.jp/projects/BiFluX/.
28. 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.
29. S. Sendall and W. Kozaczynski. Model Transformation: The Heart and Soul of Model-Driven Software Development. *IEEE Softw.*, 20(5):42–45, 2003.
30. J. Sprinkle, A. Agrawal, T. Levendovszky, F. Shi, and G. Karsai. Domain Model Translation Using Graph Transformations. In *10th IEEE Int. Conf. and Workshop on the Engineering of Computer-Based Systems*, pages 159–168. IEEE Computer Society, 2003.
31. P. Stevens. A Landscape of Bidirectional Model Transformations. In R. Lämmel, J. Visser, and J. Saraiva, editors, *GTTSE 2007, Braga, Portugal*, volume 5235 of *Lecture Notes in Computer Science*, pages 408–424. Springer, 2008.
32. P. Stevens. Bidirectional Model Transformations in QVT: semantic issues and open questions. *Software and Systems Modeling*, 8, 2009.
33. G. Taentzer. AGG: A Graph Transformation Environment for Modeling and Validation of Software. In *Int. Workshop on Applications of Graph Transformations with Industrial Relevance, AGTIVE 2003*, volume 3062 of *LNCS*, pages 446–453. Springer-Verlag, 2004.
34. G. Tamura and A. Cleve. A Comparison of Taxonomies for Model Transformation Languages. *Paradigma*, 4(1):1–14, 2010.
35. L. Tratt. A Change Propagating Model Transformation Language. Technical report, Department of Computer Science, King's College London, TR-06-07, 2006.
36. A. Wider. Towards Combinators for Bidirectional Model Transformations in Scala. In *Proc. of the 4th Int. Conf. on Software Language Engineering*, SLE 2011, pages 367–377. Springer-Verlag, 2012.
37. Y. Xiong, D. Liu, Z. Hu, H. Zhao, M. Takeichi, and H. Mei. Towards Automatic Model Synchronization from Model Transformations. In *ASE 2007*, pages 164–173, 2007.

Languages, Models and Megamodels
A Tutorial

Anya Helene Bagge[1] and Vadim Zaytsev[2]

[1] BLDL, University of Bergen, Norway, anya@ii.uib.no
[2] University of Amsterdam, The Netherlands, vadim@grammarware.net

Abstract. We all use software modelling in some sense, often without using this term. We also tend to use increasingly sophisticated software languages to express our design and implementation intentions towards the machine and towards our peers. We also occasionally engage in metamodelling as a process of shaping the language of interest, and in megamodelling as an activity of positioning models of various kinds with respect to one another.
This paper is an attempt to provide an gentle introduction to modelling the linguistic side of software evolution; some advanced users of modelware will find most of it rather pedestrian. Here we provide a summary of the interactive tutorial, explain the basic terminology and provide enough references to get one started as a software linguist and/or a megamodeller.

1 Introduction

This paper is intended to serve as very introductory material into models, languages and their part in software evolution — in short, it has the same role as the tutorial itself. However, the tutorial was interactive, yet the paper is not: readers familiar with certain subtopics would have to go faster through certain sections or skip them over.

In §2, we talk about languages in general and languages in software engineering. In §3, we move towards models as simplifications of software systems. The subsections of §4 slowly explain megamodelling and different flavours of it. The tutorial paper is concluded by §5.

2 Software Linguistics

Let us start by examining what a *language* is in a software context.

In Wikipedia, the concept is described[3] as follows:

> *Language* is the human ability to acquire and use complex systems of communication, and *a language* is any specific example of such a system. The scientific study of language is called *linguistics*.

[3] http://en.wikipedia.org/wiki/Language.

Even leaving aside the anthropocentricity of this definition, we see that languages are communication systems — spoken, written, symbol, diagrammatic. As communication systems, languages have several properties, including *structure*, *meaning* and *abstraction*.

Structure, often also referred to as *syntax* [7], is about how sentences (programs) of a language are constructed or deconstructed, and in general what components of sentences (programs) can be identified and how the language allows us to put them together. In natural and software languages, the structure is often recursive, allowing to create an infinite number of statements of arbitrary complexity.

Meaning, also called *semantics* [12], assigns sense and value to language constructs — for the sake of simplicity, we mostly assume they are syntactically correct before being concerned with their meaning; in some rare cases like automated error correction we could also contemplate the meaning of incorrect programs. There is usually a tight interplay between structure and meaning, so that by changing the structure of a sentence, you change its meaning — the activity commonly referred to as "programming".

Abstraction is what allows discussion of arbitrary ideas and concepts, that may be displaced in time and space. Abstractions allow engineers to reason about physical systems by focusing on relevant details and ignoring extraneous ones [5]. Of course, the most interesting results are the ones that could not be obtained from the real system — so, *predictions* are preferred to *measurements* [20]. A crucial feature of natural languages as well as many software languages is the ability to define and refine abstractions — for instance, in the way this introduction defines English language abstractions for discussing software languages.

Early written communication (cave paintings) had symbols, but their meaning (if any) remains unknown. Early writing systems used pictures with grammatical structure. Such picture is, in fact, a *model* of a concrete object: a picture of a bull can confer the idea of doing something with it, but cannot feed you; one does not simply smoke an image of a pipe. Once used for abstraction, the symbols can be composed in nontrivial ways. For instance, in hieroglyphics, the word "Pharaoh" is written as a combination of a duck and a circle, because the Pharaoh is the son of Ra, since "son" is pronounced similarly to "duck" and Ra is a god of sun, which is modelled by a circle for its shape [14]. In a similarly nontrivial way, "a butcher's" means "a look" in Cockney rhyming slang, since "look" rhymes with "a butcher's hook" and "butcher's" is a shortened version thereof [19]. Such combinations and combining ways are the main reason new software languages are difficult to learn, if they are paradigmatically far from the already familiar languages: the idioms of C are too different from the idioms of English; and the idioms of Haskell are too different from the idioms of C.

Languages in software engineering as used in multiple ways. There are *natural languages*, which are reused and extended (by jargon) by developers. There are also *formal languages* which are also largely reused after their underlying theories being proposed and developed in fundamental research — essentially they are the same as natural languages, but much easier for automated processing and

reasoning. Finally, there are also *artificial languages* which are specifically made by humans — such as C or Esperanto. Usually all kinds of automation-enabling languages that are used in construction and maintenance of software, are referred to as *software languages*: these are programming languages, markup notations, application programming interfaces, modelling languages, query languages, but also ontologies, visual notations with known semantics, convention-bound subsets of natural languages, etc.

For instance, any API (application programming interface) is a software language [1], because it clearly possesses linguistic properties such as:

- API has structure (described in the documentation)
- API has meaning (defined by implementation)
- API has abstractions (contained in its architecture)

However, API does not typically allow definition of new abstractions. For classical programming languages, we would have a similar list, but in domain-specific languages we would have abstractions limited by a particular domain, not by the system design (which means possibly infinite number of them, even if the abstraction mechanism is still lacking), while general purpose programming languages usually leave it to the programmer to define arbitrary abstractions (though not necessarily abstract over arbitrary parts of the language).

3 Moving to models

A *model* is a simplification of a system built with an intended goal in mind: a list of names is a model of a party useful for planning sitting arrangements; "CamelCase" is a model of naming that compresses multiple words into one. Any model should be able to answer some questions in place of the actual system [2]. Models are abstractions that can provide information about the consequences of choosing a specific solution before investing into implementation of the actual software system [29].

- Typically, a model represents a system.
- Some models represent real systems (programs, configurations, interfaces)
- Some models represent abstract systems (languages, technologies, mappings)
- Some models are descriptive/illustrative (used for comprehension)
- Some models are prescriptive/normative (used for conformance)

A model may be written (communicated) as a diagram or a text or some other representation — possibly even as a piece of software that allows to simulate behaviour. One might draw a model as an ad hoc illustration — similar to a crude cave painting — but for clarity and ease of communication across time and space, one may want to use a modelling language such as UML, BNF, XSD, CMOF, Z, ASN.1, etc.

Systematically discussing, researching and dissecting software languages has inevitably led to a special kind of models — called *metamodels* — that define

software languages. For example, a grammar [36] as a definition of a programming language, is a metamodel, and programs written in such a language, are behavioural models conforming to that metamodel. Similarly, a database schema, an protocol description or an algebraic data type definition are examples of metamodels, since they all encapsulate knowledge about allowable (grammatical) structures of a software language, each in their corresponding technical spaces.

Formally speaking, a metamodel models a modelling language [25], in which models are written, and such models are told to conform to this metamodel: an XML file conforms to an XML Schema definition; a Haskell program conforms to the metamodel of Haskell; a program depending on a library uses function calls according to its API.

4 Megamodels explain relations between models

A model of a system of models is called a *megamodel*. For example, the last paragraph of the previous section is a megamodel (in a natural language), since it models the relations between software artefacts (model, metamodel, language). Megamodels are crucial for big-picture understanding of complex systems [4]. In literature they can be called megamodels [4,16,17], macromodels [28], linguistic architecture models [15,34] or technology models [22]. Megamodels can be partial in the sense of not being complete deterministic specifications of underlying systems [13], and they can also be presented in a way that gradually exposes the system in an increasingly detailed way [34,23,35].

A cave painting of a bison may be useful to understand the concept of hunting by abstracting from the personalities of the hunters and the measurements of the animal. However, to surface and understand its implications such as the near extinction and recovery of the species, one must also have models of bison populations, ecology, human society, USA politics, Native American politics, and so on — and be able to see how they relate to each other. In the same way, megamodels can aid in understanding software technologies, comparing them and assessing the implications of design choices in software construction.

4.1 Informal megamodelling

A cave-painting approach to megamodelling could be as minimalistic as follows:

- ⋄ draw a diagram with models as nodes
- ⋄ add relations between them
- ⋄ describe relations in a natural language

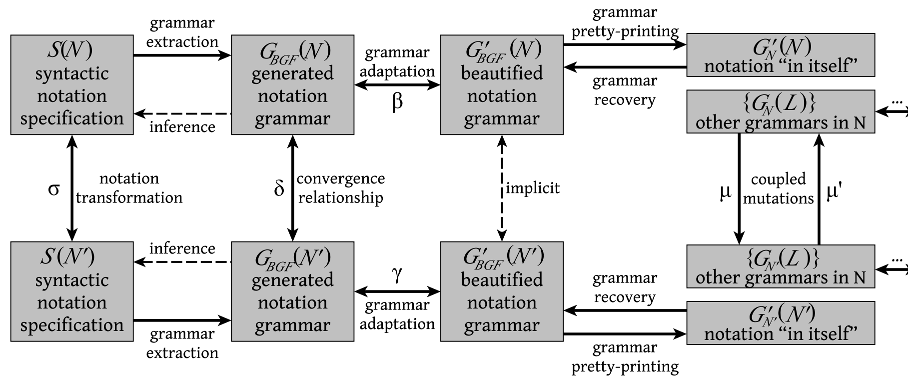
The focus of this approach is on understanding and communication [30,4]. For example, many papers, books and specifications in MDE contain an explanation of the stack of M1, M2 and M3 models (models, metamodels and metametamodels correspondingly) which positions them with respect to one another by

postulating that models conform to a metamodel and both M2 and M3 conform to a metametamodel. Such an explanation, as well as its visual representation, is a megamodel. We have to draw your attention here to the fact that such a megamodel leaves many questions open and on a certain level of understanding it is incorrect: many models conform to one metamodel, and many metamodels can conform to one metametamodel, and the fact that the metametamodel conforms to itself, is no more than an implementation detail from MDA. That is the reason for various more formal attempts to exist to express the same situation in UML or another universal notation.

There is a big subset of informal megamodelling techniques referred to as "natural" [30] — it happens all the time in unstructured environments, whenever we use conveniently available salt and pepper dispensers as proxies for entities at a conference banquet discussion, or in general whenever we use throwaway abstractions to get to the point in a quick and dirty (volatile) way.

4.2 Ad hoc megamodelling

A slightly more detailed and yet still concrete approach is to explain relations between models and languages by showing mappings between them, without trying to generalise them to relations. Such mappings are easier to define and formalise and may be enough to understand the system. Thus, instead of saying "this model belongs to this language", we show that there is a tool which processes that model and that this tool is a software language processor. Usually such models mix architectural and implementational elements and when it comes to comprehension, almost impenetrable without extensive study of the system at hand. Here is an example [32]:



After some frustration we are free to observe here how $S(N)$, whatever it is[4], becomes $G_{BGF}(N)$ after a process called "grammar extraction" [33], and that

[4] In fact, $S(N)$ is a specification of a syntactic notation such as "an Extended Backus-Naur Form dialect that uses dots to separate production rules, same level indentation to list alternatives, ..." [31].

$G_{BGF}(N)$ is linked either bijectively or bidirectionally to $G'_{BGF}(N)$, and all these boxes titled with symbols, subscripts, dashes and parentheses, are linked to their counterparts from a similarly looking chain of transformations that seem to be related to N' rather than to N.

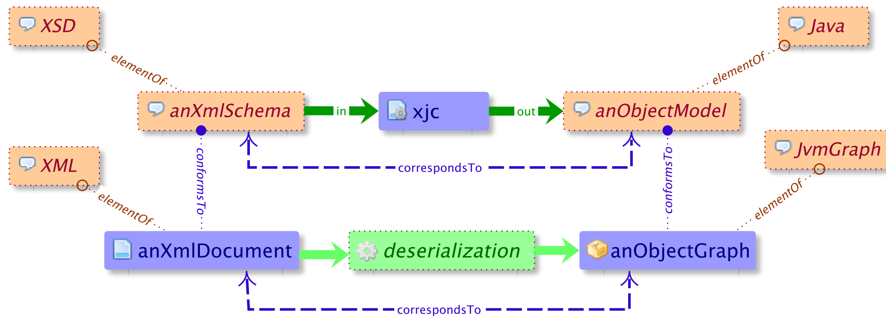
Even with a fair share of guesswork, this megamodel does not immediately bestow its observer with any piece of freshly granted knowledge. This megamodel basically encapsulates everything one could learn from the corresponding paper [32], condensing 17 pages into one diagram. It is more of a visualisation tactic than a comprehension strategy.

Many methods of ad hoc megamodelling are transformational: they use a newly introduced notation, different for each of them, to demonstrate how some software artefacts get turned into other artefacts. Unlike natural megamodelling, some ad hoc megamodelling approaches have very clearly defined semantics for their components instead of a natural language description. Unlike formal megamodelling that we will introduce below, they are typically fairly idiosyncratic and are not expressive enough to unambiguously model a situation sufficiently different from the study showcasing their application.

4.3 Instrumental megamodelling

One of the alternative approaches is to rely on some instrumental support: a tool or a language, perhaps both, that can do what a megamodel should — express relations between models, model transformations and languages. Hence, by using such a tool we can focus on providing such descriptions for a given system, perfecting them, reflecting on their evolution, etc. Committing to a framework means sacrificing at least some of the flexibility that natural and ad hoc megamodelling provide, in exchange of a much more precise understanding and definition of each component. An instrumental megamodel is not a cave painting anymore — it is a Latin text. Latin is a language everyone *kinda* understands, thus enabling its dissemination to a broader public. It might not be the best language to deliver you particular ideas, but once you get a hold on its cases, declensions and conjugations, you can use it again and again for many other tasks.

Here is an example megamodel by Favre, Lämmel and Varanovich [15]:



For a software engineer using such a megamodel "in Latin" means that each of these components is clickable and resolvable to a (fragment of a) real software artefact. In this particular case, the megamodelling language is MegaL[5], it supports entities such as "language", "function", "technology", "program", etc, and relations such as "subsetOf", "dependsOn", "conformsTo", "definitionOf" and many others. There are other megamodelling languages: AMMA[6], MEGAF[7], SPEM[8], MCAST[9], etc, some people use categorical diagrams, which are closer to the next kind of megamodelling.

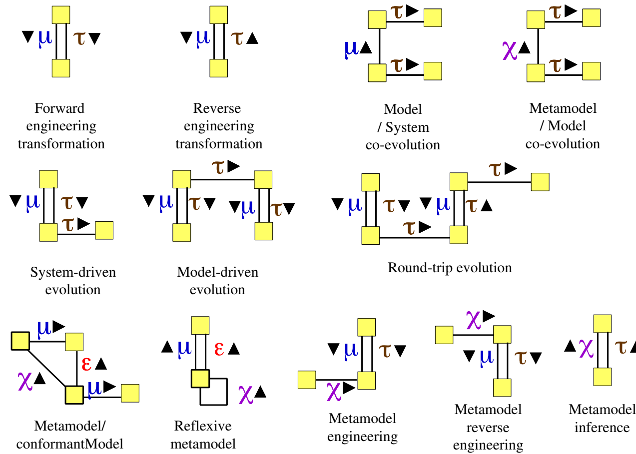
The process of navigating a megamodel and assigning a story to it, is called renarration [34]. This technique is needed quite often, since detailed megamodels can get bulky and rather intimidating — yet the same megamodels are supposed to be used to simplify the process of understanding a software system or communicating such an understanding, not to obfuscate it. Indeed, when a megamodel is drawn step by step with increasing level of detail (or vice verse, in increasing level of abstraction), it lets the user treat and comprehend one element at a time while slowly uncovering the intentions behind them. For MegaL, renarration operators include addition/removal of declarations, type restriction/generalisation, zooming in/out, instantiation/parametrisation, connection/disconnection and backtracking [23].

4.4 Formal megamodelling

Relying on tool support can be nice, but it is even better to be backed up by a theory that allows you to prove certain properties and verify your megamodels through solid analysis. Such approaches have rich mathematical foundations and vary greatly in form and taste. The choice is wide, but let us consider two different examples a little closer.

---

[5] MegaL: Megamodelling Language [15].
[6] AMMA: Atlas Model Management Architecture [3].
[7] MEGAF: Megamodelling Framework [18].
[8] SPEM: Software & Systems Process Engineering Metamodel [27].
[9] MCAST: Macromodel Creation and Solving Tool [28].



Suppose that instead of trying to come up with all kinds of relations that system fragments can have among themselves, we limit the relations to the most essential ones. Such relations can be well-understood and defined with relative ease for any particular technological space. We can refer to Jean-Marie Favre's relations [16,14]: μ — representationOf, ϵ — elementOf, δ — decomposedIn, χ — conformsTo, τ transformedTo. Then, we can on one hand afford to define each of them for our particular domain (e.g., grammarware, XML, Cobol, EMF); and on the other hand see megapatterns in them [16]:

For instance, in the top left corner we see an entity (say, X) that models another entity (say, Y), while X is also being transformed to Y. This is classical forward engineering, as opposed to reverse engineering where X models Y while the system Y is being transformed into the model X [6]. By now you can recognise the diagram of the original taxonomy by Chikofsky and Cross as an ad hoc megamodel, which also contains much more details than such a pattern, which is why the text of the paper is an important renarration of it. A similar pattern is displayed in the right bottom corner where X is being transformed into Y while also conforming to it — this could be grammatical inference, or constructing and XML schema from a selection of documents, or deriving a partial metamodel from a modelbase, or anything of that kind. All megapatterns are simplifications of real scenarios and as such, they are in some sense "wrong" — as are all models.

As another example of formal megamodelling, here is Diskin's definition of complex heterogeneous model mappings [9]:

$$\begin{array}{ccccc}
\boldsymbol{\mathcal{A}} & \xLeftarrow{\;\boldsymbol{v}\;} & \boldsymbol{\mathcal{O}} & \xRightarrow{\;\boldsymbol{w}\;} & \boldsymbol{\mathcal{B}} \\
\uparrow {:}\mu^{\mathbf{QL}} & & \uparrow {:}\mu^{\mathbf{QL}} & & \uparrow {:}\mu^{\mathbf{QL}} \\
A{:}\boldsymbol{\mathcal{A}} & \overset{f:\boldsymbol{v}}{\Lleftarrow} & M{:}\boldsymbol{\mathcal{O}} & \overset{g:\boldsymbol{w}}{\Rrightarrow} & B{:}\boldsymbol{\mathcal{B}}
\end{array}$$

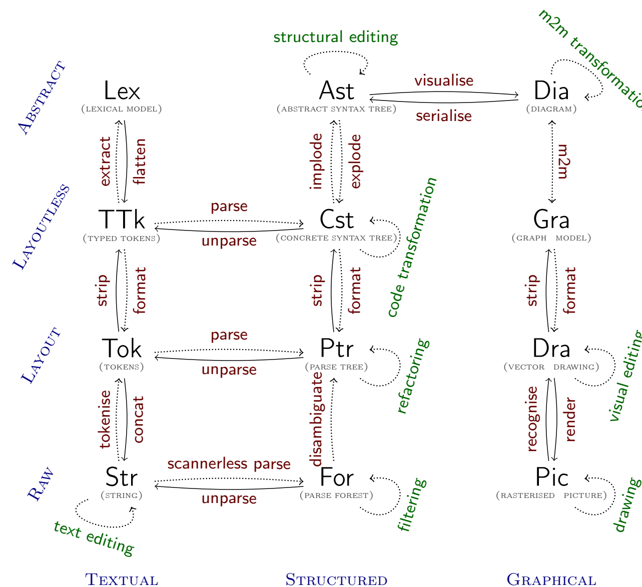
Single-line arrows are links, double arrows are graph mappings, triple arrows are diagrams of graph mappings that encapsulate type mappings inside nodes

and metamodel mappings inside arrows. Even this extremely condensed tile diagram is a simplification — since v and w are complex mappings, they should be drawn as triple arrows, while $f : v$ and $g : w$ become quadruple arrows. Still, the diagram itself remains structurally simple while still being unshakably formal. If we provide an accurate definition of our language's syntax, compositionality of metamodel mappings (a routine categorical process of defining Kleisli triples [24]), this graph turns into a (Kleisli) category. By going through some trouble or by limiting ourselves to monotonic queries, we can do the same for model mappings (not just metamodel mappings) [9].

Within this approach, each megapattern — divergence, convergence, revision of match, revision of update, improvement of match, conflict resolution — forms a tile of four involved software artefacts and labelled arrows between some of them. Then, tile algebra provides uniform rules to compose such tiles together [10].

4.5 Space megamodelling

Recall that *a metamodel is a model of a language*. (The previous sentence is a megamodel). Then, *a megamodel is a model of a technology*, since it shows how all involved fragments fit together to facilitate the process. In the previous section we have also made acquaintance with megapatterns — *models of processes within a technology*. Just one more step brings us to a abstract megamodel of an entire *technological space* [21]. For example [38]:



This megamodel models *everything* that can possibly happen when you are doing parsing, unparsing, pretty-printing, formatting, templating, stropping, etc.

Each element here is not resolvable to a concrete artefact, but rather to a subspace with its own stack of models and metamodels. For example, a concrete syntax tree (Cst) element found near the centre of the megamodel, represents concrete syntax trees, their definition as a concrete grammar, and all the techniques and tools that create, transform and validate them. A vector drawing (Dra, think SVG or GraphML), on the other hand, implies having a metamodel defining graphical elements, their coordinates and other attributes, as well as transformations such as a change of colour or realignment.

When renarrated, such a megamodel commits to becoming a representation of one particular technology, hence removing some of the elements that do not exist there and detailing the others so that they become resolvable [35]. We can also use the megamodel as a classificatory tool to look at existing techniques and positioning them with respect to others [37]. For example, what is "model-to-text transformation" commonly used in modelware frameworks and papers and deliberately omitted from being explicitly mentioned on the megamodel? In fact, it is a very particular path through this megamodel starting at Ast or Dia and going to Lex (commonly referred to as a "template" in this particular scenario) and then dropping straight to Str.

One can reasonably claim that such megamodels are in fact ontologies [8].

5 Conclusion

The tutorial was highly interactive and its biggest contribution to SATToSE was the discussion. This paper is a humble attempt to summarise (some of the) issues raised during both lecturing[10] and the hands-on parts, and provide bibliographical pointers for the most interested participants. There are many issues in megamodelling that we did not sufficiently cover — in particular, modelling the very nature of modelling [25,26] and taking both ontological and linguistical aspects into account [20,11,8].

Language is an important instrument of structured and meaningful communication, whether we use natural languages to convey information or create artificial ones tailored to the domain. We model languages with metamodels, since they are models of how software models can be put together. In practice, metamodels take many different forms such as programming language grammars, UML domain models, XML schemata and document types, library API definitions. Megamodels are used to model software technologies as systems of models, aimed first and foremost at understanding software systems, languages, tools and relations between them. Megamodelling makes relations explicit, identifies roles that software artefacts play and thus helps to understand technologies, compare them, validate, debug and deploy in a broad sense.

[10] Slides: http://grammarware.github.io/sattose/slides/Bagge.pdf.

References

1. T. Bartolomei, K. Czarnecki, R. Lämmel, and T. van der Storm. Study of an API Migration for Two XML APIs. In *SLE*, volume 5969 of *LNCS*, pages 42–61. Springer, 2010.
2. J. Bézivin and O. Gerbé. Towards a Precise Definition of the OMG/MDA Framework. In *ASE*, page 273. IEEE CS, 2001.
3. J. Bézivin, F. Jouault, P. Rosenthal, and P. Valduriez. Modeling in the Large and Modeling in the Small. In *MDAFA*, volume 3599 of *LNCS*, pages 33–46. Springer, 2004.
4. J. Bézivin, F. Jouault, and P. Valduriez. On the Need for Megamodels. *OOPSLA & GPCE, Workshop on best MDSD practices*, 2004.
5. A. W. Brown. Model Driven Architecture: Principles and Practice. *SoSyM*, 3(3):314–327, 2004.
6. E. J. Chikofsky and J. H. C. II. Reverse Engineering and Design Recovery: A Taxonomy. *IEEE Software*, 7(1):13–17, 1990.
7. N. Chomsky. *Syntactic Structures*. Mouton, 1957.
8. C. Coral, R. Francisco, and P. Mario. *Ontologies for Software Engineering and Software Technology*. Springer, 2006.
9. Z. Diskin. Model Synchronization: Mappings, Tiles and Categories. In *GTTSE*, volume 6491 of *LNCS*. Springer, 2011.
10. Z. Diskin, K. Czarnecki, and M. Antkiewicz. Model-versioning-in-the-large: Algebraic Foundations and the Tile Notation. In *ICSE CVSM*, pages 7–12. IEEE CS, 2009.
11. D. Djurić, D. Gašević, and V. Devedžić. The Tao of Modeling Spaces. *JOT*, 5(8):125–147, 2006.
12. M. Erwig and E. Walkingshaw. Semantics First! In *SLE'11*, pages 243–262. Springer, 2012.
13. M. Famelis, R. Salay, and M. Chechik. Partial Models: Towards Modeling and Reasoning with Uncertainty. In *ICSE*, pages 573–583. IEEE, 2012.
14. J.-M. Favre. Megamodelling and Etymology. A story of Words: from MED to MDE via MODEL in five millenniums. In *GTTSE*, number 05161 in Dagstuhl, 2006.
15. J.-M. Favre, R. Lämmel, and A. Varanovich. Modeling the Linguistic Architecture of Software Products. In R. B. France, J. Kazmeier, R. Breu, and C. Atkinson, editors, *MoDELS*, LNCS, pages 151–167, 2012.
16. J.-M. Favre and T. NGuyen. Towards a Megamodel to Model Software Evolution through Transformations. *ENTCS*, 127(3), 2004.
17. R. Hebig, A. Seibel, and H. Giese. On the Unification of Megamodels. *EC-EASST*, 42, 2011.
18. R. Hilliard, I. Malavolta, H. Muccini, and P. Pelliccione. Realizing Architecture Frameworks through Megamodelling Techniques. In *ASE*, pages 305–308, 2010.
19. J. Jones. *Rhyming Cockney Slang*. Abson Books, 1971.
20. T. Kühne. Matters of (Meta-)Modeling. *Software and Systems Modeling*, 5(4):369–385, 2006.
21. I. Kurtev, J. Bézivin, and M. Akşit. Technological Spaces: an Initial Appraisal. In *CoopIS, DOA*, 2002.
22. R. Lämmel. Programming Techniques and Technologies. http://softlang.wikidot.com/course:ptt13, 2013.
23. R. Lämmel and V. Zaytsev. Language Support for Megamodel Renarration. In *XM*, volume 1089 of *CEUR*, pages 36–45. CEUR-WS.org, Oct. 2013.

24. E. Manes. *Algebraic Theories*. Graduate Text in Mathematics. Springer, 1976.
25. P.-A. Muller, F. Fondement, and B. Baudry. Modeling Modeling. In *MoDELS*, LNCS, pages 2–16, 2009.
26. P.-A. Muller, F. Fondement, B. Baudry, and B. Combemale. Modeling Modeling Modeling. *Software and Systems Modeling*, 11(3):347–359, 2012.
27. Object Management Group. Software & Systems Process Engineering Metamodel (SPEM). Language Specification, OMG, 2007.
28. R. Salay, J. Mylopoulos, and S. Easterbrook. Using Macromodels to Manage Collections of Related Models. In *CAiSE*, pages 141–155. Springer, 2009.
29. B. Selic. The Pragmatics of Model-Driven Development. *IEEE Software*, 20(5):19–25, 2003.
30. Z. Zarwin, J.-S. Sottet, and J.-M. Favre. Natural Modeling: Retrospective and Perspectives an Anthropological Point of View. In *XM'12*, pages 3–8. ACM, 2012.
31. V. Zaytsev. BNF WAS HERE: What Have We Done About the Unnecessary Diversity of Notation for Syntactic Definitions. In *SAC PL*, pages 1910–1915. ACM, Mar. 2012.
32. V. Zaytsev. Language Evolution, Metasyntactically. *EC-EASST; Bidirectional Transformations*, 49, 2012.
33. V. Zaytsev. Notation-Parametric Grammar Recovery. In *LDTA*. ACM DL, June 2012.
34. V. Zaytsev. Renarrating Linguistic Architecture: A Case Study. In *MPM*, pages 61–66. ACM DL, Nov. 2012.
35. V. Zaytsev. Understanding Metalanguage Integration by Renarrating a Technical Space Megamodel. In *GEMOC*, volume 1236 of *CEUR*, pages 69–77. CEUR-WS.org, Sept. 2014.
36. V. Zaytsev. Grammar Zoo: A Corpus of Experimental Grammarware. *Fifth Special issue on Experimental Software and Toolkits of Science of Computer Programming (SCP EST5)*, 98:28–51, Feb. 2015.
37. V. Zaytsev and A. H. Bagge. Modelling Parsing and Unparsing. In *Second Workshop on Parsing at SLE 2014*, Aug. 2014. Extended Abstract.
38. V. Zaytsev and A. H. Bagge. Parsing in a Broad Sense. In *MoDELS*, volume 8767 of *LNCS*, pages 50–67. Springer, Oct. 2014.