Software Languages.Lab

BENEVOL 2011

10th edition of the
BElgian-NEtherlands software
eVOLution seminar

Technical Report VUB-TR-SOFT-01-12
Software Languages Lab
Vrije Universiteit Brussel

Sponsored by:

MoVES
Modelling, Verification and Evolution of Software

FRFC Project
Research Center on Software Adaptability

Universiteit
Antwerpen

Vrije Universiteit Brussel

Contents

1 Preface

The goal of the *BElgian-NEtherlands software eVOLution seminar (BENEVOL)* series is to bring together researchers from Belgium, The Netherlands and the neighboring countries that are working in the field of software evolution. BENEVOL offers an informal forum to meet and to discuss new ideas, important problems and obtained research results.

The 10th edition of BENEVOL took place in Brussels on the 8th and 9th of December 2011. This edition attracted 38 participants and featured an interesting program consisting of 2 keynotes (Prof. Michele Lanza and Prof. Yann-Gaël Guéhéneuc) and 15 technical talks. This document (technical report VUB-TR-SOFT-01-12 – Vrije Universiteit Brussel) provides a compilation of the abstracts of these talks.

We would like to take this occasion to thank our sponsors:

- the MoVES research network, funded the IAP Program of the Belgian State
- the FRFC Research Center on Software Adaptability funded by FNRS, and in particular the contributions of Prof. Kim Mens and Prof. Tom Mens
- Universiteit Antwerpen, and in particular Prof. Serge Demeyer

Without their financial support, it would not have been possible to organize this seminar.

The local organizers:

Andy Kellens and Carlos Noguera

Keynote: Requiem for Software Engineering

Michele Lanza
University of Lugano

Software engineering (1968 - 2011) had an amazing life, even though it was rather short.

It was a great friend; the kind of friend that stands by you when you need somebody to be there. It once saved my life. We were both young then and we weren't very close. I was hastily hacking some code when software engineering saw a speeding bug. If it wasn't for software engineering, my program would have died right there. That's how I became close to software engineering, and we have been inseparable ever since.

Software engineering was well-loved and it has done so many things on earth and I am sure it will do much more in heaven [or hell]. I will forever be grateful to have known it. All the memories I have shared with it will forever be cherished and remembered. It will forever live in my heart, in our hearts. Don't ever forget software engineering. It never wanted to see people cry. It wanted to make everyone happy. So at this moment when we are about to lay its body to rest, let us all think back and remember how it touched our lives, how it made us laugh. This is not the moment for us to shed our tears but we should all be thankful that we were given the chance to have known software engineering.

The death of software engineering was sudden, but not unexpected. Let me tell you how this all came about ...

Keynote: Multi-objective Miniaturization of Software

Yann-Gaël Guéhéneuc
University of Montreal

Joint work with Nasir Ali, Wei Wu, Giuliano Antoniol, Massimiliano Di Penta and Jane Huffman Hayes
Smart phones, gaming consoles, and wireless routers are ubiquitous; the increasing diffusion of such devices with limited resources, together with society's unsatiated appetite for new applications, pushes companies to miniaturize their programs.

Miniaturizing a program for a hand-held device is a time-consuming task often requiring complex decisions. Companies must accommodate conflicting constraints: customers' satisfaction with features may be in conflict with a device's limited storage, memory, or battery life. Also, the miniaturization process is doubly impacted by quality considerations.

First, the (poor) quality of the program to be miniaturized may impede the miniaturization process. Second, the expected quality of the miniaturized program constrains the process. In this talk, we describe a process, MoMS, for the multi-objective miniaturization of software to help developers miniaturize programs while satisfying multiple conflicting constraints. The process directs the elicitation of customer pre-requirements, their mapping to program features, and the selection of the features to port. We then present two case studies based on Pooka, an email client, and SIP Communicator, an instant messenger, to demonstrate that MoMS supports optimized miniaturization and helps reduce effort by 77%, on average, over a manual approach. Then, we discuss the use of the the process to support other software engineering problems, such as the reverse engineering of programs, the next release problem, and the porting of both software and product lines. Finally, we present challenges relating the miniaturization process with quality assessment models.

Similar Tasks, Different Effort: Why the Same Amount of Functionality Requires Different Development Effort?

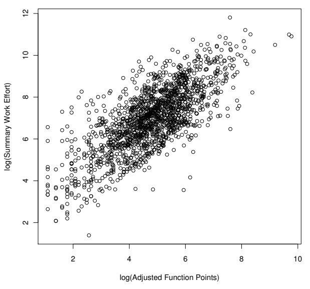
Alexander Serebrenik, Bogdan Vasilescu, Mark van den Brand
Technische Universiteit Eindhoven,
Den Dolech 2, P.O. Box 513,
5600 MB Eindhoven, The Netherlands
{a.serebrenik, b.n.vasilescu, m.g.j.v.d.brand}@tue.nl

***Abstract*—Since the appearance of Albrechts pioneering work, function points have attracted significant attention from the industry. In their work, project managers can benchmark function point counts obtained for their projects against large publicly available datasets such as the ISBSG development & enhancement repository release 11, containing function point counts for more than 5000 projects. Unfortunately, larger amount of functionality as reflected in the function points count does not necessarily correspond to a more significant development effort. In this paper we focus on a collection of ISBSG projects with a similar amount of functionality and study the impact of different project attributes on the development effort.**

In our study we consider the ISBSG development & enhancement repository release 11, the largest publicly available dataset with function point counts, containing data about more than 5000 projects developed in a variety of different countries using a variety of different design and development techniques [1]. The ISBSG repository contains information about 118 different project attributes, including its functional size as well as organizational (e.g., scheduling, development team size and its productivity), technical (e.g., architecture and the main programming language), and problem-specific attributes (e.g., business or application area). Data is provided by the project owners themselves. Functional size is for most of the projects measured by applying such methods as IFPUG function points (3799 out of 5052 projects or 75.2%) [2], hence we solely focus on the IFPUG projects.

For the IFPUG projects, the ISBSG repository contains data on the development effort. First, we exclude projects that report the effort figures only for some project phases. Second, while some projects report on the *recorded* effort, some other projects report on the *estimated* effort based, e.g., on the amount of functionality. To ensure validity of our study we solely consider projects that report on the actual recorded effort. Furthermore, we focus only projects that record the effort in staff hours (as opposed, e.g., "productive time") and that only record time spent on software development, including project management and project administration, but excluding non-project specific supporting activities such as control and audit or hardware support. Overall, excluding projects that report the effort figures only for some project phases, that report estimated rather than recorded effort, that report recorded effort expressed in some other unit than staff hours, or that include the effort dedicated to non-project specific supporting activities, reduces the number of considered projects to 1661.

Finally, as the data in the ISBSG repository is provided by the project owners themselves, it might become polluted by imprecise or unreliable values. Therefore, ISBSG quality reviewers assess the soundness of the data submitted. The result of the assessment ranges from "A" to "D", where "A" indicates that the data "was assessed as being sound with nothing being identified that might affect its integrity", while for "B" the data "appears sound but some aspects might have affected the data or count integrity". Assessment "C" indicates impossibility of assessment due to incompleteness of the data provided, while little credibility should be given to data assessed "D". Restricting our attention to the "A"-projects reduces the number of the eligible projects to 84, while 1609 projects remain if both "A"- and "B"-projects are considered. Hereafter we consider both "A"- and "B"-projects.

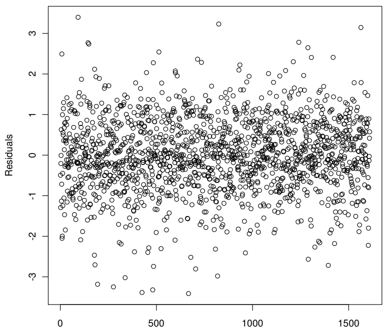


As mentioned, for each project the ISBSG repository contains information about 118 different project attributes including the summary work effort, functional size (measured using the so called unadjusted and adjusted function points counts), as well as organizational, technical and problem-specific attributes. We prefer adjusted function points count to unadjusted function points count (cf. [3]), and consider the following project attributes: primary programming language, language type, organization type, intended market, year of project, development type, platform, and architecture.

Plotting the summary work effort against the adjusted function points count reveals presence of four outliers, projects containing more than 5000 function points. Moreover smaller projects up to 500 function points cover the entire range of work effort. The log-scale plot recommended in [4] shows a clear linear relation between the summary work effort (SWE) and the adjusted function points count (AFP). Using linear regression we obtain the following model:

$$\log(\text{SWE}) = 0.20735 + 0.65569 * \log(\text{AFP})$$

The fitted linear model is adequate: F-statistic equals 2003 on 1 and 1607 degrees of freedom with the corresponding p-value not exceeding 2.2×10^{-16}, and p-values both for the intercept and for the coefficient do not exceed 0.05. The residuals plot exhibits a clear chaotic pattern.



We would like to get a better insight into the distribution of the residuals, i.e., we would like to explain their diversity by investigating to what extent they can be explained using one of the remaining project attributes such as the primary programming language or the intended market. We carry out the explanation step by means of econometric inequality indices, recently applied in the context of software engineering [5]. Due to the nature of residuals, the chosen inequality index should be applicable to negative as well as positive values. Moreover, to calculate the explanation percentage [6], the index should be *decomposable*, i.e., representable as $I^{between}(G) + I^{within}(G) = I$ for any partition G into mutually exclusive and completely exhaustive groups. From all the inequality indices studied in [7], there is only one that satisfies the requirements of decomposability and applicability to the negative numbers, namely the Kolm index [8]. Explanation percentages are shown in the column "With NA" below.

Project attribute	Explanation %	
	With NA N = 1609	Without NA N = 151
Primary programming language	21.59	36.86
Organization type	14.05	48.15
Year of project	11.60	31.40
Development type	11.45	24.92
Architecture	5.25	20.11
Development platform	4.10	28.91
Language type	2.59	6.18
Intended market	2.15	22.09

The table data clearly indicates that such attributes as the primary programming language, the organization type, and the year of the project explain a higher share of the inequality in the residual values than the development platform, language type and intended market. Language type is a type of the programming language, e.g., 3GL, 4GL or an Application Generator. Since many primary programming languages belong to one language type, and one programming language can belong solely to one language type, the language type induces a more coarse grained partition of the projects considered. Therefore, the explanation value of the language type is lower than that of the primary programming language [5]. High explanation values related to the organization type are caused by association of different organization types to the same project, e.g., "Wholesale & Retail Trade" *and* "Financial, Property & Business Services". Since explanation provided by the inequality indices is applicable solely to mutually exclusive decompositions, we had to introduce a very fine-grained partitioning, including a group containing only projects associated with both "Wholesale & Retail Trade" *and* "Financial, Property & Business Services". Extending inequality indices to non-mutually exclusive groups is considered as future work. The high explanation percentage obtained for the year of the project corroborates the earlier findings of [3] that stress the importance of the project age in effort estimation.

One of the main issues arising when analyzing the ISBSG data, recognized already in [4], is related to presence of missing values. Indeed, since the ISBSG data is based on self-reporting, many project aspects are not being reported. In particular, this would mean that that all projects with unreported value for, e.g., development type, would be put together in the same group. To evaluate the impact of missing values on the explanation percentages we have eliminated all the projects having a missing value in at least one of the project attributes considered, and recalculated the Kolm indices based on the remaining 151 projects. These values are present in the "Without NA" column. Overall, the explanation percentages are higher, which may be explained by the decrease in the number of projects, and, therefore, by a more fine grained partition induced by the same project attributes. We see that the primary programming language, the organization type, and the year of the project still provide high explanation values.

To conclude, in this paper we have applied econometric inequality indices to study how different project attributes can explain diversity of the residuals of the logarithm of the summary work effort with respect to the logarithm of the adjusted function points, i.e., how different project attributes can explain why projects with similar amount of functionality require different development effort.

References

[1] *ISBSG Development & Enhancement Repository, release 11*, International Software Benchmarking Standards Group, 2009.

[2] *Function Point Counting Practices Manual. Release 4.2*, International Function Point Users Group, 2004.

[3] B. Kitchenham, S. L. Pfleeger, B. McColl, and S. Eagan, "An empirical study of maintenance and development estimation accuracy," *Journal of Systems and Software*, vol. 64, no. 1, pp. 57–77, 2002.

[4] B. Kitchenham and E. Mendes, "Why comparative effort prediction studies may be invalid," in *PROMISE'09*. ACM, 2009, pp. 4:1–4:5.

[5] A. Serebrenik and M. G. J. van den Brand, "Theil index for aggregation of software metrics values," in *ICSM'10*, 2010, pp. 1–9.

[6] F. A. Cowell and S. P. Jenkins, "How much inequality can we explain? a methodology and an application to the United States," *Economic Journal*, vol. 105, no. 429, pp. 421–430, March 1995.

[7] B. Vasilescu, A. Serebrenik, and M. G. J. van den Brand, "You can't control the unfamiliar: A study on the relations between aggregation techniques for software metrics," in *ICSM'11*, 2011, pp. 313–322.

[8] S.-C. Kolm, "Unequal inequalities I," *Journal of Economic Theory*, vol. 12, no. 3, pp. 416–442, 1976.

Code Decay Triangulation using Metrics, Experts' Opinion and Defect Counts

Juan Fernández-Ramil
Computing Dept. The Open University, UK
j.f.ramil@open.ac.uk

David Reed
IT Division of a UK Government Department
ydj_reed@yahoo.co.uk

10th edition of the BElgian-NEtherlands software eVOLution seminar BENEVOL,
Brussels, 8-9 Sept 2011 http://soft.vub.ac.be/benevol2011/

Motivation

In contrast to physically engineered artefacts, software does not deteriorate through use. Code quality, however, may *decay* (i.e. deteriorate) through the process of software evolution (a.k.a. maintenance). Such decay may have negative human, technical and economic consequences. For example, software maintainers may find that the code is becoming excessively complex. Evolution may become more time consuming and difficult than it should. Other stakeholders may not receive the functional improvements they are waiting for in time. Unexpected side-effects may emerge when new changes are implemented. Defect fixing may get harder. And so on...

The problem of code decay (a.k.a. code aging, excessive complexity, 'spaghetti' code) has been identified and discussed a long time ago [e.g., Lehman 1974, Parnas 1994]. There are many code decay empirical studies in the literature [e.g. Eick et al 2001]. There are, at least, three different ways of trying to assess the level of code decay in a particular system: direct measuring of the code through software metrics, surveying experts' opinion about code quality and using indirect measures (e.g. process related measures such as defect counts). It isn't known whether these three different ways will converge to the same insights when applied to a particular system.

In this extended abstract, we briefly report the findings of a case study in which software metrics, a developers' questionnaire and defect counts were compared and used in an attempt to rank the software's components with respect to their level of decay. We aimed at achieving a greater clarity on the 'details' of how to measure code decay in a particular context. We also wanted to provide the organisation owing the software with a ranked list of components which could be use to prioritise any refactoring or replacement efforts.

The Case Study

The software used as a case study was a proprietary business critical information system. The system handled an important database which is used nationally by many stakeholders. Any errors in the system and in the database may have serious legal and financial implications. The system was initially implemented in 2004 following the PRINCE2 methodology and using mainly Borland Delphi, a variant of Object Pascal. At the time of the study the system had evolved for four years, with 19 releases. At the most recent release the system consisted of approximately 225,000 lines of Delphi code including comments. Further details can be found in [Reed 2009].

Data Collection

The study involved the collection of three types of data: code metrics, defect counts and expert opinion. The code metrics included McCabe complexity, coupling (CBO), afferent and efferent coupling and lack of cohesion (LCOM2). The expert opinions were gathered via a specially designed questionnaire. The number of reported defects was obtained from the documentation and manually assigned to each of the subsystems.

The metric data was visualised by plotting *point values* (average values) per release and box-plots (i.e. abstracted views of the distributions) for the first and most recent releases. From the box-plots, the *tail length* and the *tail volume* was calculated for each of the 11 subsystems. Changes in metrics values were measured relative to the first release rather than in absolute terms. A first version of the questionnaire was generated and sent to a small number of experts who could give comments on it. The questionnaire was then revised based on their feedback and then sent to the real developers. It was answered by 10 out of 12 developers. In order to normalise defect counts by the size of the system, the cumulative number of defects was divided by the current size of the system in number of lines of code.

Main Results

The three types of data provided some evidence that could be interpreted as decay being present. However, the convergence was not complete. For example, the average McCabe complexity increased slightly from 2.96 to 3.08 (4.1%) during the 4 years of evolution. Tail volumes increased for complexity, CBO and afferent coupling, showing evidence of code decay. Surprisingly, tail volumes decreased for efferent coupling and LCOM2, showing improvement rather than decay. Six developers said that the system has become more complex; three developers indicated that the complexity has stayed the same and one developer said that the system has become less complex. Cumulative defect values (normalised by size) showed a positive slope (increasing trend) from month 22.

Seven ranking pairs (based on point values, tail length, tail volume, questionnaire and defects) for the 11 subsystems were compared using Kendall's and Spearman's rank correlation measures. The results ranged widely from positive correlations (e.g. Spearman's Rho value of 0.7 for the pair 'point values – defects' to negative correlations (e.g. Kendall's Tau value of -0.4 for the pair 'tail volume – questionnaire').

Despite the evidence showing, overall, that the code has decayed, it was found that different types of measurement may lead to different results. Code decay is multi-dimensional. Careful examination is needed to interpret which measures are more meaningful in a given context. In general, expert opinion seems to be the most reliable source of information, followed by code metrics (at distribution level) and finally defect counts. Defect counts can vary widely due to, for example, changes in the testing effort, without necessarily indicating code decay. Within code metrics, the analysis based on distributions (box-plots) was found to be more insightful than point values (averages). The latter generally 'compress' the tail of the distribution where the most complex code elements reside and in this way may hide the parts of the code where the actual code decay is actually happening.

Conclusion

Code decay symptoms are not easy to triangulate, that is, to confirm (or not) through different types of measurement whether the code has suffered from quality deterioration. In this case study an initial approach based on code metrics, questionnaire and defect counts showed mixed results. For example, some metrics showed deterioration while others showed improvement. Moreover, subsystem decay rankings of possible decay based on different types of information are not always leading to the same results. Despite all this, the methodological approach used in this case study could be used by a software organisation to start an internal discussion and reflection on the evolutionary 'trajectory' of the system and on the possible measures to improve code's quality where it is most needed. How to apply code decay measurement approaches in a given context or project is not immediately clear and needs experimentation. All this, makes code decay detection a difficult problem for practitioners and an interesting area of research which combines the software evolution and the software measurement topics.

References

[Eick et al 2001] S.G. Eick et al, Does Code Decay? Assessing the Evidence from Change Management Data, IEEE TSE, 27(1), pp. 1-12 , 2001.

[Lehman 1974] M.M. Lehman, *Programs, Cities and Students – Limits to Growth?,* Inaugural lecture, Imperial College of Science, Technology, London, 14th May 1974

[Parnas 1994] D.L. Parnas, Software Aging, Proc 16th ICSE, Sorrento Italy, pp 279-287

[Reed 2009] D. Reed, *Code Decay – Examining Evidence from Expert Subjective Assessment and Metrics*, M801 Master's Dissertation, Computing Dept., The Open University, Milton Keynes, U.K., March 2009.

An Approach for Refactoring Planning

Javier Pérez[1,2], Yania Crespo[2]

[1] University of Mons; Software Engineering Lab
[2] University of Valladolid; Department of Computer Science
javipeg@gmail.com, yania@infor.uva.es

Refactorings are source code transformations that change the system's internal design while preserving its observable behaviour [?]. Refactorings can be used to improve, in general, certain quality factors such as reusability, understandability, maintainability, etc. More specifically, refactorings can help to achieve a particular system structure or consolidating the system's architecture [?].

Refactoring operations are meant to be executed in small steps, so that more complex refactorings can be executed by the composition of simpler ones. Behaviour preservation is also easier to check in the case of simpler refactorings. When a refactoring process aims to solve a complex problem, such as the correction of design smells [?], a significant amount of changes is needed. Refactorings' preconditions can help to assure behaviour preservation, but at the same time they hinder the application of complex transformation sequences because they restrict the applicability of refactoring operations. If any precondition of any operation in the intended transformation sequence, is not fulfilled at the time of its application, the whole sequence can not be applied. This makes it hard for the developer to perform complex refactoring processes.

We have developed a technique that uses Hierarchical Task Network (HTN) Planning [?] to tackle this problem [?]. The proposal is based at the definition of refactoring strategies and refactoring plans. A Refactoring Strategy is a heuristic-based, automation-suitable specification of a complex behaviour-preserving software transformation which is aimed at a certain goal. It can be instantiated, for each particular case, into a Refactoring Plan. A Refactoring Plan is a sequence of instantiated transformations, aimed at achieving a certain goal, that can be effectively applied over a software system in its current state, while preserving its observable behaviour. It can be an instance of a Refactoring Strategy.

To develop this proposal we have focused in how refactorings are used for design smell correction. We have first analysed the current correction specifications from different authors, identified the main characteristics of these specifications and unified them in a single model. We have then identified the current problems and the requirements that correction specifications have to meet in order to be automation-suitable. We have defined a model for refactoring strategies that fulfills these requirements and defined a language in order to ease writing these specifications. We have identified the requirements an underlying approach has to meet to support the computations of refactoring plans from refactoring strategies and we have selected HTN automated planning for this purpose. We have implemented this proposal as a reference prototype and it has been evaluated by performing two case studies over a set of open-source systems.

The assembled prototype is composed of a small HTN domain, which is our main contribution to this prototype, and some third-party tools. The refactoring planning domain we have written, addresses *Feature Envy* and *Data Class* design smells and comprises the specifications of a set of refactorings, refactoring strategies, other transformations and system queries, all of which have been represented as task networks. Regarding the other tools, we have used JTransformer, a program transformation tool based in Prolog, to obtain the predicate-based representation of Java systems. We have also used iPlasma, a design smell detection tool, to obtain reports of the two design smells we have addressed. The logic-based representation of a Java program, and the information about which entities are affected of which smells constitute the initial state of the system for the planner. A set of scripts compiles the refactoring planning domain and all these inputs as a refactoring planning problem for JSHOP2, the planner we had selected. The planner searches refactoring sequences, for applying the requested strategies and produces refactoring plans.

To conclude, two case studies have been carried out to evaluate our approach, and to test the reference prototype in terms of effectiveness, efficiency and scalability. The case studies used are addressed for removing the *Feature Envy* and *Data Class* design smells and have been performed over 9 software systems of different sizes ranging from small to medium size. The results of the study confirm that our approach can be used to automatically generate refactoring plans for complex refactoring processes in a reasonable time. The studies performed also demonstrate that the efficiency of the HTN family of planners and the expressiveness of the JSHOP2 domain specification language makes it the appropriate planner to support the refactoring planning problem.

Aknowledgements

This work has been partially funded by the spanish government (*Ministerio de Ciencia e Innovación*, project TIN2008-05675).

References

1. Kent Beck and Martin Fowler. *Bad Smells in Code*, chapter 3. Refactoring: Improving the Design of Existing Code. Addison-Wesley, 1 edition, June 1999.
2. Dana S. Nau, Tsz-Chiu Au, Okhtay Ilghami, Ugur Kuter, J. William Murdock, Dan Wu, and Fusun Yaman. SHOP2: An HTN Planning System. *Journal of Artificial Intelligence Research (JAIR)*, 20:379–404, 2003.
3. Colin J. Neill and Phillip A. Laplante. Paying down design debt with strategic refactoring. *IEEE Computer*, 39(12):131–134, 2006.
4. W.F. Opdyke. *Refactoring Object-Oriented Frameworks.* PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, 1992. also Technical Report UIUCDCS-R-92-1759.
5. Javier Pérez. *Refactoring Planning for Design Smell Correction in Object-Oriented Software.* PhD thesis, University of Valladolid, 2011.

Refactoring In Scheme Using Static Analysis

Jens Nicolay

Scheme is a small but elegant and powerful programming language with clean syntax. It allows for both imperative and functional programming and is able to express several different programming paradigms. As a consequence, Scheme has influenced the design of many other more widely-used and industrially-relevant languages, while also making the language especially suited for experimentation. Even with all this power, expressivity and influence, no refactoring catalog or well-known refactoring tools exist for Scheme. There are a couple of reasons why this is might be the case. Firstly, most refactorings are based on static analysis of code, and static analysis is far from trivial to perform in dynamic languages such as Scheme. Also, Scheme isn't widely used outside of academia, although this does not change the fact that it is very well suited as a research language, especially in the context of the growing interest in dynamic languages that we see today. We argue that refactoring in Scheme should enjoy the same status as the language itself: it should deepen our general understanding of techniques for, and implementations of, program analysis and transformation, with influences far beyond the Scheme language. Our aim is twofold. First of all we want to compile a refactoring catalog for Scheme, containing the exact specifications of general refactorings like RENAME, ADD PARAMETER, and so on, expressed as program transformations guarded by pre- and postconditions. We also want to discover refactorings that are not readily identified as general refactorings and see how these might carry over into other languages. Our second goal is to design a specification language that allows us to express Scheme refactorings. For this specification language we again choose Scheme, but with built-in backtracking for convenience and with a library of primitives that allow reasoning over Scheme programs. This reasoning is based on the results of a sufficiently precise, powerful and fast static analysis, on which several layers of primitives are layered so that the right level of abstraction can be selected by the designer of refactorings. Th analysis approximates value flow, control flow and interprocedural dependencies. Note that our two aims go hand in hand. In order to write down refactoring specifications we need a language. At the same time this specification language will determine what the refactoring specifications will look like. To validate our work, we select existing refactoring scenarios and see how our approach deals with them, assessing strengths and investigating weaknesses. In order to facilitate experimentation and as a way to make our work publicly available, we have also build an Eclipse plugin aimed at programming in Scheme and containing several refactorings. The provided Scheme editor also detect certain patterns in the source code in order to provide feedback during development. The plugin also allows a developer to perform program analysis and transformation using a meta-programming approach, which is useful for prototyping refactorings.

Cloud–Computing: from Revolution to Evolution

Sébastien Mosser[⋆], Eirik Brandtzæg[⋆,†], Parastoo Mohagheghi[⋆]

[⋆] SINTEF IKT
[†] University of Oslo
{firstname.lastname}@sintef.no

Extended Abstract, submitted to BENEVOL'11

Introduction. Cloud–Computing [1] was considered as a *revolution*. Taking its root in distributed systems design, this paradigm advocates the share of distributed computing resources designated as "*the cloud*". The main advantage of using a cloud-based infrastructure is the associated scalability property (called *elasticity*). Since a cloud works on a *pay–as–you–go* basis, companies can rent computing resources in an elastic way. A typical example is to temporary increase the server–side capacity of an e–commerce website to avoid service breakdowns during a load peak (*e.g.*, Christmas period). However, there is still a huge gap between the commercial point of view and the technical reality that one has to face in front of "*the cloud*". As any emerging paradigm, and despite all its intrinsic advantages, Cloud–Computing still relies on fuzzy definitions[1] and lots of buzzwords (*e.g.*, the overused "*IaaS*", "*PaaS*" and "*SaaS*" acronyms that does not come with accepted definitions).

Problem Statement. A company that wants to migrate its own systems to the cloud (*i.e.*, be part of the cloud *revolution*) has to cope with existing standards. They focus on cloud modeling, but does not provide any support for software evolution. Thus, the *evolution* of a legacy system into a cloud–based system is a difficult task. On the one hand, an immediate issue is the paradigm shift (*e.g.*, migrating a centralized COBOL system to a service–oriented architecture deployed in the cloud). This issue is not directly related to Cloud–Computing, and considered as out of the scope of this work (*e.g*, see the SMART method [2]). On the other hand, the Cloud–Computing paradigm introduces several key concepts that must be used during the evolution process to accurately promote a given legacy system into a cloud–based one. For example, deploying an application to the cloud does not automatically transform it into a scalable entity: the evolution process must carefully identify the components that can be replicated to ensure elasticity using resources available in the cloud. Consequently, the evolution of a legacy system into a cloud–based system requires a dedicated entity that support *(i)* reasoning mechanisms dedicated to

[1]The Cloud–Standard initiative (`http://cloud-standards.org/`) lists dozens of overlapping standards related to Cloud–Computing. They focus on infrastructure modeling or business modeling.

cloud concepts and *(ii)* technical implementation of cloud deployment for such systems.

Objectives. Instead of designing yet another cloud standard, our goal (part of the REMICS project [3]) is to define a language that supports evolution to the cloud. The objectives of this language are the following:

- *Platform–independence.* It is an abstract modeling language to support the description of the software that will be deployed on the cloud. This architecture description language includes cloud–specific concepts (*e.g.*, elastic capability, deployment geographic zone, failure recovery, parallelism, data protection). The language acts as an intermediary pivot between legacy applications and the cloud. Thus, the evolution process does not target concrete cloud providers entities anymore. Moreover, it is possible to reason on the modeled element using a cloud–based vocabulary.
- *Transparent projection.* Based on the modeled entities, the framework associated to the language handle the projection of the abstract description into concrete cloud entities. A matching step is used to accurately bind the abstract resource described in the language with available resources published by cloud providers. For example, at the infrastructure level, it identifies which virtual images must be deployed on what cloud resources.
- *Automated deployment.* The language comes with an interpreter that implements the actual deployment. It abstracts the underlying platform complexity and programming interfaces. Thus, assuming that the evolution process targets the modeling language previously described, the application can be deployed on existing clouds in an automatic way.

Perspectives: Cloud Evolution. The definition of this language and its associated engine is a first step. Based on this experience, we will consider evolution in cloud context according to two complementary axis: *(i)* the evolution of the application after its initial deployment and *(ii)* the evolution of the cloud infrastructure itself.

Acknowledgments. This work is funded by the EU Commission through the REMICS project (`http://remics.eu`), contract number 257793 (FP7).

References

[1] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. H. Katz, A. Konwinski, G. Lee, D. A. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. Above the Clouds: A Berkeley View of Cloud Computing. Technical Report UCB/EECS-2009-28, EECS Department, University of California, Berkeley, Feb 2009.

[2] G. Lewis, E. Morris, D. Smith, and L. O'Brien. Service-Oriented Migration and Reuse Technique (SMART). In *Proceedings of the 13th IEEE International Workshop on Software Technology and Engineering Practice*, pages 222–229, Washington, DC, USA, 2005. IEEE Computer Society.

[3] P. Mohagheghi and T. Sæther. Software Engineering Challenges for Migration to the Service Cloud Paradigm: Ongoing Work in the REMICS Project. In *SERVICES*, pages 507–514. IEEE Computer Society, 2011.

“Why Humans Can’t *Green* Computers” An Autonomous Green Approach for Distributed Environments

Adel Noureddine[⋆]

INRIA Lille – Nord Europe, Project-team ADAM
University Lille 1 - LIFL CNRS UMR 8022, France
adel.noureddine@inria.fr

Energy-aware solutions and approaches are becoming broadly available as energy concerns is becoming mainstream. The usage of computers and other electronic devices (*e.g.*, smartphones, sensors, or digital equipments) is increasing, thus impacting the overall energy consumption. Although ICT accounts for 2% of global carbon emissions in 2007 [4], ICT solutions could help in reducing the energy footprint of other sectors (*e.g.*, building, transportation, industry). In [10], the Climate Group estimates that ICT solutions could reduce 15% of carbon emissions in 2020. However, in 2007, ICT footprint was 830 $MtCO_2e$ and is expected to grow to 1,430 $MtCO_2e$ in 2020 [10]. These numbers show the need for efficient ICT solutions in order to reduce carbon emissions and energy consumption.

Reducing the energy consumption of connected devices and computers requires a comprehensive view of the different layers of the system. Sensors and actuators, used to monitor energy consumption and modify devices’ options, need to be controlled by intelligent software. Applications running on the devices and the hardware itself also need to be monitored and controlled in order to achieve efficient energy savings. The middleware layer positions itself as a relevant candidate for hosting energy-aware approaches and solutions.

Many approaches have been proposed to manage the energy consumption of the hardware, operating system, or software layers. In particular, more and more architectural or algorithmic solutions are now emerging in the middleware layer. The widespread usage of mobile devices and the high coverage of networks (WiFi, 3G) have led to a new generation of communicating and moving devices. Therefore, the middleware layer requires a flexible approach to manage efficiently the energy consumption of such devices at a large-scale level.

Many middleware platforms, architectures, optimization techniques or algorithms already exist for energy management of hardware or software. Rule-based approaches offer a high degree of architectural autonomy, but with a limited decisional autonomy. The architecture of the middleware is flexible and evolutive, and can easily cope with changes in the environment. Rules, on the other hand, need to be predefined and updated on environment’s evolutions. Current rule-based solutions ([3, 5, 11]) use predefined and manually updated rules and policies.

Predefined approaches vary from algorithmic adaptations ([1, 6]), to protocol ones [8], to modeling ones [12]. The approach in [7, 9] uses predefined allocation and prediction

[⋆] Ph.D. student under the supervision of Prof. Lionel Seinturier and Dr. Romain Rouvoy algorithms and predefined coordination. The approach in [2] takes a wider approach with adapting itself to the user habits and the environments events. This context-based event learning offers better autonomic management than other approaches.

Although these approaches offer a certain degree of autonomic management of energy consumption, a full autonomous energy management is yet to be defined. An energy-aware autonomous approach should therefore imitate the human body metabolism: the platform needs to be transparent to the user and to devices and applications, but without limiting users' high-level decisions. In the human body, when energy becomes low, the system starts by using its reserves and notifying the human about the situation (*e.g.*, the human feels hunger). Therefore, the human could apply high-level decisions, such as eating (to recharge his energy and reserves), or reduce his activity, or go to sleep (low power mode). We therefore believe that middleware approaches should take inspiration from biologic systems and provide a similar autonomous functioning for energy-awareness because the complexity of systems is rapidly increasing.

We propose an approach that adapts the software components and hardware parameters in an autonomous manner. User interaction is therefore limited to defining some of the user preferences and very high-level energy-aware policies (through an energy-aware DSL). We also propose to use distributed services (locally and on the cloud) in order to provide flexibility and evolution to the architecture. The local services will store energy information, but also usage patterns and user preferences. Remote servers on the cloud will take the role of a worldwide knowledge repository. They will also be used for utility services (*e.g.*, electricity price), user activity synchronization (*e.g.*, the user's online agenda), and computation and intelligence offloading. A modular monitoring environment is reponsible for collecting energy-aware information from the environment. The degree of the collected information can vary from fine-grained information (*e.g.*, per class or per method in an application) or coarse-grained (*e.g.*, per process or per device). Preliminary results on our monitoring environment provide fine-grained information on the energy consumption of classes and methods inside a Java application. We are currently working on extending and experimenting the monitoring environment in order to optimize its overhead and integrate it into our architecture.

References

[1] Binder, W., Suri, N.: Green Computing: Energy Consumption Optimized Service Hosting. In: Proceedings of the 35th Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM '09). pp. 117–128. Springer-Verlag, Berlin, Heidelberg (2009)

[2] Capone, A., Barros, M., Hrasnica, H., Tompros, S.: A new architecture for reduction of energy consumption of home appliances. In: Towards eEnvironment, European conference of the Czech Presidency of the Council of the EU (e-Envi'2009). Prague, Czech Republic (2009)

[3] Demeure, I., Paroux, G., Hernando-ureta, J., Khakpour, A.R., Nowalczyk, J.: An energy-aware middleware for collaboration on small scale manets. In: Proceedings of the Autonomous and Spontaneous Networks Symposium (ASN'08). Paris, France (November 2008)

[4] Gartner: Green it: The new industry shockwave. Presentation at Symposium/ITXPO Conference (2007)

[5] Klein, A., Jerzak, Z.: Ginseng for sustainable energy awareness: flexible energy monitoring using wireless sensor nodes. In: Proceedings of the 4th ACM International Conference on Distributed Event-Based Systems (DEBS '10). pp. 109–110. ACM, New York, NY, USA (2010), http://doi.acm.org/10.1145/1827418.1827443

[6] Mohapatra, S., Venkatasubramanian, N.: Parm: Power aware reconfigurable middleware. In: Proceedings of the 23rd International Conference on Distributed Computing Systems (ICDCS'03). p. 312. IEEE Computer Society, Washington, DC, USA (2003)

[7] Sachs, D.G., Yuan, W., Hughes, C.J., Harris, A., Adve, S.V., Jones, D.L., Kravets, R.H., Nahrstedt, K.: Grace: A hierarchical adaptation framework for saving energy (2004)

[8] Schiele, G., Handte, M., Becker, C.: Experiences in designing an energy-aware middleware for pervasive computing. In: Proceedings of the 6th Annual IEEE International Conference on Pervasive Computing and Communications (PERCOM'08). pp. 504–508. IEEE Computer Society, Washington, DC, USA (2008), http://portal.acm.org/citation.cfm?id=1371610.1372873

[9] Vardhan, V., Yuan, W., III, A.F.H., Adve, S.V., Kravets, R., Nahrstedt, K., Sachs, D.G., Jones, D.L.: Grace-2: integrating fine-grained application adaptation with global adaptation for saving energy. International Journal of Engineering Science 4(2), 152–169 (2009)

[10] Webb, M.: SMART 2020: enabling the low carbon economy in the information age, a report by The Climate Group on behalf of the Global eSustainability Initiative (GeSI) (2008)

[11] Xiao, Y., Kalyanaraman, R.S., Ylä-Jääski, A.: Middleware for energy-awareness in mobile devices. In: Proceedings of the 4th International ICST Conference on Communication System software and middleware (COMSWARE'09). pp. 1–6. ACM, New York, NY, USA (2009)

[12] Zeng, H., Ellis, C.S., Lebeck, A.R.: Experiences in managing energy with ecosystem. IEEE Pervasive Computing 4, 62–68 (January 2005), http://dx.doi.org/10.1109/MPRV.2005.10

Managing Runtime Evolution in Dynamic Software Systems

Extended Abstract

Nicolás Cardozo[1,2], Sebastán González[1], Kim Mens[1], and Theo D'Hondt[2]

[1] ICTEAM Institute, Université catholique de Louvain
Place Sainte-Barbe 2, 1348 Louvain-la-Neuve, Belgium
[2] Software Languages Lab, Vrije Universiteit Brussel
Pleinlaan 2, 1050 Brussels, Belgium

In the context of mobile and pervasive computing [7] contextual information (like personalization, location, internal device's state, and on-spot environmental information) is becoming central to application development. Current-day systems are required to incorporate and react to contextual information, which emphasizes the growing importance of runtime software evolution [3]. To address this need, the context-oriented programming (COP) paradigm has been proposed to provide the ability of writing programs that can adapt, correct, or extend their behavior dynamically at runtime according to the surrounding execution environment. Different COP languages have been defined as either new languages or extensions of existing languages [1,4].

COP languages introduce different language abstractions that enable the definition, dynamic (de)activation, and composition of contexts and context-dependent behavior. Let us illustrate the idea by means of a particular COP language, *Subjective-C* [5], although similar concepts can be found in most COP languages. Contexts are defined as first-class program entities and are usually given by assigning a semantic meaning to internal/external characteristics of the environment (e.g., the GPS coordinates 50°51'0"N 4°21'0"E correspond to the `Brussels` context), which is defined as `@context(Brussels)`. Behavior is associated to a context by annotating partial method definitions with the corresponding contexts for which the method is applicable as follows, `@contexts Brussels -(void) getCoordinates{...}`. Such method definitions become available dynamically in the main application only if their context of definition is active. Contexts can be activated and deactivated dynamically, using respectively the `@activate(Brussels)` and `@deactivate(Brussels)` constructs.[3]

In the context of COP applications, activations and deactivations of contexts are assumed to happen concurrently and without warning, which may lead to incoherences or inconsistencies with respect to the expected application behavior. To deal with this, different proposals have been made to manage the definition and composition of dependencies between contexts [2,5]. Such proposals only provide models that constrain the dynamics of (de)activating a context, according to the state of its related contexts. Nonetheless, in most cases, the

[3] Activations and deactivations are triggered by a sensed change in the internal or external information.

corresponding language abstractions or runtime support for the model is not provided. Furthermore, the informal and high-level definition of such constraints makes their verification difficult and computationally expensive especially in the highly dynamic settings encountered in mobile and pervasive computing.

We propose to address the problem of consistency management in systems that dynamically evolve at runtime, along two fronts. First, to cope with the dynamic nature of COP systems, we propose a module for the precise definition and management of interaction between context dependencies. We use the Petri net [6] based formalisms for this. In addition to the advantages given by the formal definition, the model also provides a first-hand view on the dynamics and state of COP systems. Moreover, it serves as an underlying implementation in our context management system, thus providing a lightweight verification mechanism for the activation and deactivation of contexts.

Second, an static analysis module could be used to provide an upfront fine-grained reasoning about consistency and validity properties of the application. The introduced formalism of Petri nets already provides analysis and verification mechanisms that could be used to, for example, find whether an application may reach a conflicting configuration of active contexts. Alongside the analysis tools provided by Petri nets, another verification module could identify possible inconsistencies at the method level. Based on the static information for each method (contexts in which it may be applicable, and methods which it may call), an analysis can be performed to ensure that application behavior remains consistent whenever methods are switched. For example, in a localization application, detecting faulty message sends to `getCoordinates` on a GPS device, just as the location context changes from `GPS` to `Extrapolate`, where the method is not defined. This module thus gives meaningful feedback about potential errors, and different possibilities on how to solve them for example, by creating a dependency between the two contexts.

The modules proposed here constitute a clear step forward in the support for runtime evolution of COP applications. We provide a lightweight runtime system for managing dependencies between adaptations, and an upfront analysis to identify possible runtime conflicts at a fine-grained level of granularity. Other modules will be explore in the future, to widen the family of problems addressed.

References

1. Costanza, P., Hirschfeld, R.: Language constructs for context-oriented programming: an overview of ContextL. In: Proceedings of the Dynamic Languages Symposium. pp. 1–10. ACM Press (Oct 2005), collocated with OOPSLA'05
2. Desmet, B., Vallejos, J., Costanza, P., De Meuter, W., D'Hondt, T.: Context-oriented domain analysis. In: Modeling and Using Context. pp. 178–191. Lecture Notes in Computer Science, Springer-Verlag (2007)
3. Gabriel, R.P., Goldman, R.: Conscientious software. In: Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications. pp. 433–450. OOPSLA'06, ACM Press, New York, NY, USA (2006)

4. González Montesinos, S.: Programming in Ambience: Gearing up for dynamic adaption to context. Ph.D. thesis, Université catholique de Louvain (October 2008)
5. Gonzlez, S., Cardozo, N., Mens, K., Cdiz, A., Libbrecht, J.C., Goffaux, J.: Subjective-C: Bringing context to mobile platform programming. In: Proceedings of the International Conference on Software Language Engineering. Lecture Notes in Computer Science, vol. 6563, pp. 246–265. Springer-Verlag (2011)
6. Murata, T.: Petri nets: Properties, analysis and applications. Proceedings of the IEEE 77(4), 541 – 580 (April 1989)
7. Satyanarayanan, M.: Pervasive computing: Vision and challenges. IEEE Personal Communications 8(4), 10–17 (Aug 2001)

Modelling the behaviour and recognizing 3D virtual objects

Romuald Deshayes
Univesité de Mons,
Service de Génie Logiciel,
Faculté des Sciences
romuald.deshayes@umons.ac.be

October 26, 2011

Abstract

In this document, we present an overview of our research topic, mixing both software modelling and computer vision. Our work will focus on two problems, the first is the research of a generic solution to specify the interactive behaviour of 3D virtual objects, the second is the improvement of the robustness of the 3D object recognition. Eventually, resolving those two problems will allow the computer, equiped with a 3D sensor, to automatically associate a behavioural model to recognized objects, allowing advanced interaction in a virtual scene. In order to be able to use next generation 3D sensors we will develop an extensive framework to deal with the rapid evolution of sensors and will allow to use our new recognition algorithm on various 3D sensors.

1 Introduction

Objects around us differ in nature, composition, function... Also in their behaviour, every object reacts differently to stimuli depending on many factors such as those aforementioned. In this context, the first part of our research work consists in identifying and classifying different types of objects based on their behaviour. We will then model this behaviour using a visual formalism (such as statecharts or petri nets). The purpose of this approach is to provide a generic and expressive way to reduce the complexity of interaction with 3D virtual objects (we already used such a formalism to specify the behaviour of a 3D graphical application [1]). Interaction is here expressed in a broad meaning; it can be interaction between two virtual objets (e.g. physical contact), between a real object and a virtual one or it can be a gestural interaction between a virtual object and a user (e.g. hand movement).

In computer vision, a common problem for a machine is to detect and recognize the objects around it, using a capture device (e.g. a camera). Currently, this kind of problem is often resolved by using learning-based algorithms along with a set of 2D views of the object to be recognized. Nowadays, thanks to 3D sensors (such as Microsoft Kinect) and 3D reconstruction algorithms, we can exploit the geometric shape of objects to be identified to provide a greater robustness for recognition algorithms, especially when the 3D objects are located in a complex scene containing many objects that can be partially occluded. Current techniques based only on 2D views can not directly exploit the geometric shape of the objects , making recognition difficult to achieve. Methods exploiting 3D vision have started to appear, but their use in complex scenes with partially occluded objects is still a topic under research.

2 Research challenges

A first scientific challenge of our research is to develop a domain-specific visual modelling language (DSML) based on visual languages such as statecharts or petri nets to model the interactive behaviour of real world objects. These objects will be usable in Virtual Reality (VR) applications bringing a substantial opening in the field of computer simulation. It is therefore necessary to study in depth the advantages of both formalisms mentionned to see if one of them is most appropriate to meet the needs of our research.

A second scientific challenge consists in proposing a new algorithm for 3D object recognition based on knowledge of the geometry of the object to be recognized using a 3D sensor (e.g. a Kinect sensor). Recognition algorithms typically use databases containing 2D pictures of an object in order to recognize it. Thanks to the 3D information that is available using such 3D sensors, we have a lot more informations that are interesting to exploit. Using those additional informations will help to improve the robustness of the algorithm because it will no longer be necessary to make assumptions about the shape of the objects to be recognized (which is dificult using 2D only). We will also improve the complexity of these algorithms because fewer calculations must be made for recognition. Indeed, the search of particular features is a very expensive process on 2D images. Lastly, we will use 3D vision to allow our recognition algorithm to segment a 3D scene and more precisely to detect discontinuities in the scene.

In order to be able to use next generation 3D sensors (which will probably be more accurate and work at a higher resolution), we will develop an extensive framework coupled with high level routines to access raw data of the sensors in a generic way. Developing this framework will help us to deal with the rapid evolution of capture devices and will allow to use our new recognition algorithm on various 3D sensors.

References

[1] R. Deshayes and T. Mens. Statechart modelling of interactive gesture-based applications. In *First International Workshop on Combining Design and Engineering of Interactive Systems through Models and Tools (ComDeisMoto), organized at INTERACT 2011*, 2011.

Continuous Architecture Evaluation

Eric Bouwers and Joost Visser
Software Improvement Group, Amsterdam, The Netherlands
E-mail {e.bouwers,j.visser}@sig.eu

Most software systems start out with a designed architecture which documents the important design decisions. These decisions include, but are not limited to, responsibilities of each of the main components and the way in which the components interact. Ideally, the implemented architecture of the system corresponds exactly with the design, only those components which are described are implemented and all components interact through pre-defined communication channels. Unfortunately, in practice we often see that the original design is not reflected in the implementation. Common deviations are more (or less) components, undefined dependencies between components and components implementing unexpected functionality.

There are many reasons for these discrepancies to occur. For example, the choice for a technology can lead to an unwanted implementation because the chosen technology does not allow a particular construct. Deviation can also arise because of process-related issues, for example because new functionality is added to the system without taking into account the design. Lastly, it could simply be the case that there is an error in the designed architecture.

In these situations the development team can decide to solve the issue by means of a quick fix outside of the designed architecture to meet a deadline. Even though everybody knows that this type of fixes should be temporarily, the priority of solving these 'cosmetic' issues is low. After all, the system is working correctly, so why change something which is not broken?

The examples illustrate legitimate reasons for deviating from the designed architecture. By involving both the development team as well as the architects in an evaluation of the implemented architecture both the implementation and the design can evolve together. Many methods are available for such evaluations, varying greatly in depth, scope and required resources. The end-result of such evaluations are, amongst others, an up-to-date overview of the implemented architecture and the corresponding design.

But when should such an evaluation take place? Depending on the amount of resources required the evaluation can take place once or twice during a project, or periodically (for example every month). Unfortunately, in between the evaluations issues can still arise, which still leads to deviations between the design and the implementation. The later these deviations are discovered the more costly it is to fix them.

A solution to these problems is to continuously monitor important aspects of the implemented architecture. This can be done automatically by the means of software metrics. A basic metric, such as the number of components, is easy to calculate after each change and can serve as a trigger to perform a quick manual evaluation to see whether the change fits into the current design. If this is not the case a more detailed evaluation can be performed, potentially leading to a full-scale architecture evaluation.

Basic metrics (number modules, number of connections) are easy to measure and provide relevant information. However, just examining these two metrics does not make it possible to detect all types of changes, for example when a single components is implementing too much of the overall functionality.

In our current research project we are extending the set of available architecture metrics by new metrics which are related to quality aspects as defined in ISO 9126. More specifically, we have designed and validated two new metrics which quantify the Analyzability and the Stability of an implemented software architecture.

The first metric we designed is called "Component Balance" [1]. This metric takes into account the number of components as well as the relative sizes of the components. Due to the combination of these two properties, both systems with a large number of components (or just a few components) as well as systems in which one or two components contain most of the functionality of the system receive a low score. We validated this metric against the intuition and opinion of experts in the field of software quality assessments by means of interviews and a case study. The overall result is a metric which is easy to explain, can be measured automatically and can therefore be used as a signaling mechanism for either light-weight or more involved architecture evaluations.

The other concept we introduced is the "dependency profile" [2]. For this profile each module (i.e. source-file or class) is placed inside one of four categories; hidden inside a component, being part of the requires interface of a component, being part of the provides interfaces of a component, or being part of both interfaces. The summation of all sizes of the modules inside a category provides a system-level quantification of the encapsulation of a software system. This metric has been validated by an empirical experiment in which the changes which occurred to a system are correlated to the values of each of the four categories. The main conclusions of the experiment is that with more code encapsulated within the components of a system more of the changes remain localized to a single component.

Both of the metrics have shown to be useful in isolation. We are taking the next step by determining how these metrics can best be combined in order to reach a well-balanced evaluation of an implemented architecture. In order to answer the question when a more elaborate evaluation should take place we are planning to determine appropriate thresholds for these two metrics. The combined results of these studies ensures that these metrics can be embedded within the services currently offered by the Software Improvement Group.

References

[1] E. Bouwers, J. Correia, A. van Deursen, and J. Visser. Quantifying the analyzability of software architectures. In *Proceedings of the 9th Working IEEE/IFIP Conference on Software Architecture (WICSA 2011)*. IEEE Computer Society, 2011.

[2] E. Bouwers, A. van Deursen, and J. Visser. Quantifying the encapsulation of implemented software architectures. Technical Report TUD-SERG-2011-031, Delft Software Engineering Research Group, Delft University of Technology, 2011.

Evolution Mechanisms of Automotive Architecture Description Languages

Yanja Dajsuren, M. G. J. van den Brand, and Alexander Serebrenik

Eindhoven University of Technology
5600 MB, Eindhoven, The Netherlands
{Y.Dajsuren | M.G.J.v.d.Brand | A.Serebrenik}@tue.nl

As software becomes more and more important for automotive systems, introducing and adopting existing solutions from software engineering discipline are becoming common practice. One of the approaches being recognized as an important contribution to the automotive industry is Architecture Description Language (ADL) [1]. ADLs are used to describe a system at different phases of its development and to facilitate communication between different parties. Although many general-purpose ADLs exist, ADLs for safety critical systems and specifically for automotive systems have been developed to address the need of expressing quality attributes such as dependability, safety, timing aspects and variability issues. Automotive ADLs like EAST-ADL [2], SAE AADL [3], AML [4], and TADL [5] are introduced to improve the software development process of automotive systems. Since evolution is considered as one of the costliest software development activities [6], ADLs need to provide explicit mechanisms to support it. However, there has been so far no attempt to evaluate the evolution mechanisms in the automotive ADLs.

This paper aims to analyze mechanisms of supporting design-time evolution by two widely researched automotive ADLs, namely EAST-ADL [7] and AADL [8][1]. We used the evolution features defined in the ADL classification framework of Medvidović and Taylor [9]. The framework defines architecture modeling features based on *components*, *connectors* and *architectural configurations*. A component is defined as a unit of computation or a data store with an explicit interface, which is an interaction point with other components and external world. The component evolution is informally defined as the change of a component's properties such as interface, behavior or implementation. Subtyping of component types and refinement of component features are considered as common techniques to support systematic evolution of components. A connector is used to model the interactions between different components and to define the rules that govern the interactions. The connectors may not result in compilation units, but they can be implemented as messages between components for example. The connector evolution is a modification of its properties such as interface, semantics, or connector constraints. Evolution of components and configurations are closely related to connectors thus existing connectors may be modified or refined further by evolution mechanisms like incremental information filtering, subtyping, and refinement. An architectural configuration describes architectural structure by connecting appropriate components. Descriptions of configurations enable analyses of architectures for adherence to design heuristics. The configuration evolution is supported by incremental addition, reconnection, replacement, and removal of components and connectors.

After analyzing EAST-ADL and AADL using the Medvidović and Taylor framework, we conclude that evolution mechanisms for connectors are not explicitly addressed in the definition of automotive ADLs. In EAST-ADL, higher level design models are refined by the lower level components containing more implementation-oriented aspects. In AADL, a component evolution is supported by extensions (by enabling component type to have multiple implementations and by refinement of existing elements of a component). Connectors are not modeled as first-class objects in EAST-ADL and AADL, therefore no explicit evolution mechanisms are provided. However, in AADL ports are declared as features in component types and can be refined into concrete features from abstract definitions. In terms of enabling evolution mechanisms for the architecture configuration, EAST-ADL and AADL provide addition and modification of new components and connectors.

[1] We refer to EAST-ADL 2.0 and AADL 2.0.

References

1. N. Navet and F. Simonot-Lion, *Automotive Embedded Systems Handbook.* Industrial information technology series, CRC Press, 2009.
2. P. Cuenot, P. Frey, R. Johansson, H. Lönn, and et al., "The EAST-ADL Architecture Description Language for Automotive Embedded Software," in *Model-Based Engineering of Embedded Real-Time Systems*, vol. 6100 of *Lecture Notes in Computer Science*, pp. 297–307, Springer Berlin, 2010.
3. SAE International, "Architecture Analysis and Design Language." http://www.aadl.info/.
4. U. Freund, M. von der Beeck, P. Braun, and M. Rappl, "Architecture Centric Modeling of Automotive Control Software," 2003.
5. K. Klobedanz, C. Kuznik, A. Thuy, and W. Mueller, "Timing Modeling and Analysis for AUTOSAR-based Software Development - A Case Study," in *Design, Automation Test in Europe Conference Exhibition (DATE)*, pp. 642 –645, 2010.
6. C. Ghezzi, M. Jazayeri, and D. Mandrioli, *Fundamentals of Software Engineering.* Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1991.
7. The ATESST Consortium, "EAST-ADL 2.0 Specification."
http://www.atesst.org/home/liblocal/docs/EAST-ADL-2.0-Specification_2008-02-29.pdf.
8. SAE International, "AADL 2.0 Specification."
http://standards.sae.org/as5506a/.
9. N. Medvidovic and R. Taylor, "A Classification and Comparison Framework for Software Architecture Description Languages," *IEEE Transactions on Software Engineering*, vol. 26, no. 1, pp. 70 –93, 2000.

Reverse Engineering Architectural Feature Models

Mathieu Acher[1], Anthony Cleve[1], Philippe Collet[3], Philippe Merle[2], Laurence Duchien[2], and Philippe Lahire[3]

[1] PReCISE Research Centre, University of Namur, Belgium
{mac, acl}@info.fundp.ac.be
[2] INRIA Lille-Nord Europe, Univ. Lille 1 - CNRS UMR 8022, France
{philippe.merle,laurence.duchien}@inria.fr
[3] Université de Nice Sophia Antipolis - I3S (CNRS UMR 6070), France
{collet,lahire}@i3s.unice.fr

Software product line (SPL) engineering aims at generating tailor-made software variants for the needs of particular customers or environments. SPL principles and techniques are gaining more and more attention as a means of efficiently producing and maintaining multiple similar software products, exploiting what they have in common and managing what varies among them.

It is not always feasible to design and implement a complete mass customization production line to support the full scope of products needed on the foreseeable horizon. In many cases, SPL practitioners rather have to deal with (legacy) software systems, that were not initially designed as SPLs. It is the case of FraSCAti, a large and highly configurable component and plugin based system, that have constantly evolved over time and now offers a large number of variants, with many configuration and extension points. The variability of such existing and feature rich systems should be properly modeled and managed.

A first and essential step is to explicitly identify and represent the variability of a system, including complex constraints between architectural elements. We rely on feature models that are widely used to model the variability of an SPL in terms of mandatory, optional and exclusive features as well as Boolean constraints over the features [5]. Feature models characterize the scope [3] of an SPL by specifying the set of combination of features (configurations) supported or not by an SPL. Reverse engineering the feature model of an existing system is a challenging activity [4]. The architect knowledge is essential to identify features and to explicit interactions or constraints between them. But the manual creation of feature models is both time-consuming and error-prone. On a large scale, it is very difficult for an architect to guarantee that the resulting feature model correctly represents the valid combination of features supported by the software system. The scope defined by the feature model should not be too large (otherwise some unsafe composition of the architectural elements are authorized) or too narrow (otherwise it is a symptom of unused flexibility of the architecture). Both automatic extraction from existing parts and the architect knowledge should be ideally combined to achieve this goal.

We present a comprehensive, tool supported process for reverse engineering architectural feature models [1]. At the starting point of the process, an intentional model of the variability – a feature model – is elaborated by the software architect. As the software architect feature model may contain errors, we develop automated techniques to extract and combine different variability descriptions of an architecture, namely a hierarchical software architecture model and a plugin dependencies model. Then, the extracted feature model and the software architect feature model are reconciled in order to reason about their differences. Advanced editing techniques are incrementally applied to integrate the software architect knowledge. The reverse engineering process is tool supported and made possible by the combined use of FAMILIAR [2] operators (aggregate, merge, slice, compare, etc.).

We illustrate the process when applied to a representative software system, FraSCAti. Our experience in the context of FraSCAti shows that the automated procedures produce both correct and useful results, thereby significantly reducing manual effort. First, the gap between the feature model extracted by the procedure and the feature model elaborated by the software architect appears to be manageable, due to an important similarity between the two feature models. However, it remains helpful to assist the software architect with automated support, in particular, to establish correspondences between features of the two feature models. The most time-consuming task was to reconcile the granularity levels of both feature models. For this specific activity, tool supported, advanced techniques, such as the safe removal of a feature, are not desirable but mandatory, since basic manual edits [5] of feature models are not sufficient. Second, the extraction procedure recovers most of the variability expressed by the software architect and encourages the software architect to correct his initial model. A manual checking of the five variability decisions imposed by the software architect shows that the extraction is not faulty. It correctly reproduces the information as described in the software artefacts of the project. Third, we learn that the software architect knowledge is required *i)* to scope the SPL architecture (e.g., by restricting the set of configurations of the extracted feature model), especially when software artefacts do not correctly document the variability of the system and *ii)* to control the accuracy of the automated procedure.

An open issue is to provide a mechanism and a systematic process to reuse the software architect knowledge, for example, for another evolution of the architectural feature model of FraSCAti.

References

1. Mathieu Acher, Anthony Cleve, Philippe Collet, Philippe Merle, Laurence Duchien, and Philippe Lahire. Reverse engineering architectural feature models. In *ECSA'11*, LNCS. Springer. material and experiments: `https://nyx.unice.fr/projects/familiar/wiki/ArchFm`.
2. Mathieu Acher, Philippe Collet, Philippe Lahire, and Robert France. In *Symposium on Applied Computing (SAC)*, pages 1333–1340, Taiwan, March. Programming Languages Track, ACM.
3. Isabel John and Michael Eisenbarth. A decade of scoping: a survey. In *SPLC'09*, volume 446 of *ICPS*, pages 31–40. ACM, 2009.
4. S. She, R. Lotufo, T. Berger, A. Wasowski, and K. Czarnecki. Reverse engineering feature models. In *ICSE'11*, pages 461–470. ACM, 2011.
5. T. Thüm, D. Batory, and C. Kästner. Reasoning about edits to feature models. In *ICSE'09*, pages 254–264. IEEE, 2009.

Squinting at the data: Investigating software artifact provenance using KISS techniques

Michael W. Godfrey
University of Waterloo
Visiting Distinguished Scientist at CWI-Amsterdam

"Provenance" is a term from archaeology and the arts that refers to a set of evidence supporting the claimed origin of an artifact, such as a piece of pottery or an oil painting. Recently, the term has been used in an electronic context — "digital provenance" — to indicate an artifact such as a software component or set of data, really is what it claims to be and should be permitted to be used within sensitive operating environments.

In this talk, I suggest how we can stretch the definition further to encompass "software artifact provenance". That is, for a give software development artifact such as a user-visible feature, a source code function, or a third-party library, we might want to ask the question: Where did this come from and what is the evidence? For example, one might wonder how a given feature was decided upon during a mailing list discussion, how it manifested itself in the code, and how it has been maintained since the initial implementation. For a given function, one might wonder about its history within the design of the system: Was it designed to fit exactly here, or was it moved or cloned from elsewhere? And for a given third-party jar file that has been included in a Java system distribution, one might ask: What version of the library is this, and how do we know?

In this talk I will sketch some of the ideas behind this work, and show how we might phrase some of these questions in terms of concrete criteria. In particular, we will concentrate on simple techniques for reducing a large search space of candidates down to a small handful that can be examined in detail using more expensive techniques. A concrete example of investigating third-party libraries in Java systems will be presented.

This is joint work with Daniel German of the University of Victoria, Julius Davies of the University of British Columbia, and Abram Hindle of the University of Alberta.

Querying the History and Evolution of a Software Project

Reinout Stevens
Software Languages Lab
Vrije Universiteit Brussel

Developers spend most of their time understanding how a software system works. In order to do this they need answers to a wide variety of questions. There have been extensive studies concerning the nature of these questions.

Using these studies, we identified three different categories of questions. The first one contains simple questions that can be answered using a proper IDE. An example of such a question is finding out the places where a method is called by the rest of the system.

The second category contains more complex questions, which can be answered using program query tools. Program query tools allow developers to describe the behaviour or characteristics the source code has to exhibit, and identifies source code that corresponds to the specified query. An example of such a question is finding out whether there is code that accesses the database outside of the persistence infrastructure.

The final category contains questions regarding the history and evolution of a software project. For example finding out how often a method was changed can only be answered by consulting the history of the project. A more complex example is finding who added an if-test in a method, and for what reason it was added. These sorts of questions cannot be easily answered using existing tools.

We propose a new program query tool that allows developers to answer these kinds of questions. This tool needs access to the history of a project, and provide this information to the developer. The history of a software project can be found in a version repository, which nowadays are an industry best practice.

Developers need to be able to specify the following characteristics:

1. The characteristics of the source code in one version.

2. The evolution of the source code across different versions.

3. The characteristics of a version in which the source has to be found.

4. The temporal relations between versions in which the evolution of the source code has to be found.

There are several challenges for such a query tool. We list some of them:

Code analyses are computation intensive, and doing these for each version will not scale. There is need for an incremental analysis that reuses information from a previous version.

A complete representation of each version consumes too much memory. The number of changed entities across versions is limited, allowing reusing these entities in different versions.

In this presentation we show the current state of this tool. We explain the workings by several illustrative examples.

Exploring Risks in the Usage of Third Party Libraries

Steven Raemaekers
Technische Universiteit Delft

Third party libraries can be beneficial to reduce development time and costs, but there are also various risks associated with it, such as lower quality standards or security risks. The goal of this paper is to explore to what extent risks

involved in the use of third party libraries can be assessed automatically. To that end, we first of all propose a 'commonality rating' which is based on frequency of use in a corpus of 103 open source and 178 proprietary systems.

Furthermore, we propose an 'isolation rating' which measures concentration and distribution of import statements of third party libraries through packages of a system. We evaluate the proposed rating methodology by conducting a number of case studies on both open source and commercial systems, in which we compare our ratings with a qualitative assessment of library risks involved in those systems. Other system properties which could serve as risk indicators are also investigated.

An empirical study on the specialisation effect in Open Source communities

Mathieu Goeminne and Tom Mens

October 28, 2011

1 Introduction

Since a couple of decades, open source software has gained popularity due to the savings they represent and the ability for the users to modify and improve the software themeselves. As the number of projects which the entire history is available grows over time, the number of empirical studies on them grows as well. Most of these empirical studies are carried out with no consideration for other artefacts but source code. Unfortunately, a restriction to the study of source code evolution only is not sufficient to understand and explain some evolutionary behaviour. In order to gain a better insight into how a software project evolves, information about persons involved in the software development, and in particular developers, must be taken into account.

We are carrying out an empirical study on the evolution of the GNOME[1] ecosystem, a collection of 1325 open source projects. Our aim is to study how the developers involved in an open source software community organize themselves to share development tasks.

2 Methodology

In a first phase, we extracted information from the source code repositories of 1325 GNOME projects, and stored them in FLOSSMetrics-compliant[2] databases. We defined 13 file categories, such as *code*, *image*, *documentation*, based on the file types, file names and directories in which files are stored. For instance, *.png* files will be classified as images. Each file category represents a type of activity done by the developer who touched the file. We assigned such an *activity category* to each file that has been created, modified or deleted during the project's life. Our classification process is inspired by the one of Robles *et al.* [2].

A recurrent problem when carrying out empirical studies involving source code repositories is identity matching [1]. Persons involved in open source projects can have several user accounts they use. In order to get a more accurate model of the developer's activities, all accounts belonging to the same physical person must be merged in a single identity representing the person. In our approach, we have addressed this problem into account and used unique identities representing physical persons instead of their multiple accounts.

3 Research questions

In order to gain a better insight of how developers organize themselves over time, we are studying the following questions:

- Are developers mainly active in a small number of projects?
- Are developers mainly involved in a small number of activity types?

[1] `www.gnome.org/`

[2] `www.flossmetrics.org/`

- Is there a correlation between certain activities related to the software development?
- Does the number of developers involved in an activity type affect the extent to which these developers specialise themselves?

These questions must be refined in subquestions. Preliminary results of our study on the GNOME projects lets us hypothesise that the developers are highly specialised in some categories of activities, like coding (`code`) or translation (`i18n`), whereas in other categories the development process is not subject to specialisation, as the boxplots of Figure 1 show. In the figure each developer is represented by a value between 0 and 1 for each category. The value expresses the specialisation degree of the considered developer. A value of 1 means that the developer only works on this type of files, a value of 0 means that the developer has never worked on this type of file. Categories *code*, *i18n* and *develdoc* reveal a specialisation degree higher that is significantly higher than the other ones.

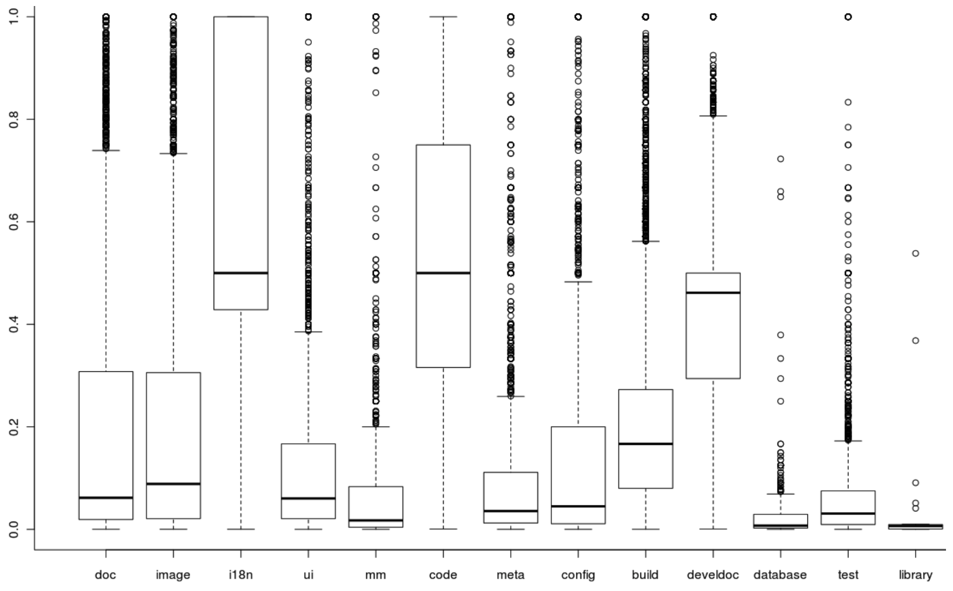


Figure 1: Developer's specialisation across GNOME projects for each activity category.

References

[1] Mathieu Goeminne and Tom Mens. A comparison of identity merge algorithms for software repositories. *Science of Computer Programming*, 2012.

[2] Gregorio Robles, Jesus M. Gonzalez-Barahona, and Juan Julian Merelo. Beyond source code: the importance of other artifacts in software development (a case study). *J. Syst. Softw.*, 79(9):1233–1248, 2006.