Contents lists available at ScienceDirect

Science of Computer Programming

journal homepage: www.elsevier.com/locate/scico

Editorial

Preface to the special issue on software evolution, adaptability and variability

Welcome to this special issue on Software Evolution, Adaptability and Variability. *Software evolution* is the term used in software engineering to refer to the process of developing an initial version of the software and then repeatedly updating it to satisfy changing user requirements. Software evolution is an inevitable activity: successful and long-living software applications do not only have to cope with changes in their application domain, their success also stimulates users to request new and improved features [1–3]. In addition, a single software implementation cannot always satisfy the various requirements demanded by different groups of users. One of the techniques to cope with such differing user requirements is *variability*, i.e., the introduction of variation points in the software's implementation that enable its instantiation in different contexts or for different purposes [4]. Another enabling technology to support software evolution is *software adaptability*, a way to leverage the current software investment by engineering adaptability into the system. Adaptable software implementations are prepared today to fit tomorrow's needs [5].

Software evolution is an important and complex research topic, especially because almost all software development organizations are exposed to it. Software systems are typically business critical and are required to run continuously and flawlessly. Software engineers are pressured to adjust existing systems to new business opportunities, emerging technologies, law changes, and so on. These changes are to be made in a cost-effective manner, without loss of quality of the existing functionality [1]. Effective software evolution therefore also comprises activities that focus on the implementation itself (e.g., engineering variability and adaptability into the system) and represents an investment in the future of the software's implementation.

This special issue originates from the BElgian-NEtherlands software eVOLution (BENEVOL) seminar, a yearly gathering of software evolution researchers from Belgium, the Netherlands and the surrounding countries like Luxembourg, Germany and France. Each year the BENEVOL seminar assembles around 40 researchers and features high-quality and highly innovative contributions in the larger area of software evolution, often leading to interesting discussions and collaborations between researchers.

This special issue showcases a number of recent advances in the area of software evolution. The open call for this special issue received 33 submissions and after two rounds of intense reviewing by at least 3 referees per submission, we are pleased to present you 12 of these papers in this special issue.

The topics covered in this special issue cover a wide variety of aspects that are highly relevant for software evolution in general and for software adaptability and variability in particular. We hope that readers will enjoy this special issue and through these papers gain useful insights into the larger domain of software evolution. We would like to extend our gratitude to all the authors who submitted papers to this special issue. Also, many thanks to the referees who ensured that all authors received valuable feedback on how to improve their work; their constructive feedback helped to shape the papers in this issue. Our thanks also go to Jan Bergstra, editor of the Science of Computer Programming journal, for hosting this special issue and to Bas van Vlijmen, editorial assistant of the Science of Computer Programming journal, for all his advice.

We now briefly introduce the twelve papers of this special issue.

Arbuckle proposes a new method for measuring software evolution in terms of artefacts' shared information content in his paper entitled *"Studying software evolution using artefacts' shared information content"*. In order to do this, Arbuckle uses a similarity value representing the quantity of information shared between artefact pairs. Similarity values for releases are then collated over the software's evolution to form a map quantifying change through lack of similarity. The method has general applicability: it can disregard otherwise salient software features such as programming paradigm, language or application domain because it considers software artefacts purely in terms of the mathematically justified concept of information content.

0167-6423/$ – see front matter © 2011 Elsevier B.V. All rights reserved.
doi:10.1016/j.scico.2011.04.004

Callo Arias et al. report on their experiences with reverse architecting a large software-intensive system in *"A top-down strategy to reverse architecting execution views for a large and complex software-intensive system: An experience report"*. In particular, they describe their experiences with using a top-down strategy to use and embed an architecture reconstruction approach in the incremental software development process of the Philips Magnetic Resonance Imaging (MRI) scanner. Their study focuses on improving the start-up time of the scanner, for which they constructed an up-to-date execution view. They report that this view enabled the developers to reduce the start-up time of the MRI scanner with about 30%.

Castro et al. present an abductive reasoning approach to detect and correct inconsistencies between a software system's source code and design rules in the *"Diagnosing and correcting design inconsistencies in source code with logical abduction"*. Key to the presented approach are the design rules, which are documented as relationships between sets of source code elements. These sets are defined intensionally, through a logic query that quantifies over the program's source code. Once an inconsistency has been detected, the approach provides a library of corrective actions.

In *"A text-based approach to feature modelling: Syntax and semantics of TVL"*, Classen et al. introduce TVL, a text-based alternative to feature models as a way of representing variability in software product line engineering. The main goal of designing TVL was to provide engineers with a human-readable language with a rich syntax to make modelling easy and models natural, but also with a formal semantics to avoid ambiguity and allow powerful automation. This contrasts feature models, which are said to lack conciseness, naturalness and expressiveness.

Di Cosmo et al. introduce a model-driven approach to support upgrading Free and Open Source Software (FOSS) systems in their work entitled *"Supporting software evolution in component-based FOSS systems"*. The approach promotes the simulation of upgrades to predict failures before affecting the real system. Both static and dynamic aspects are taken into account.

The paper *"Predicting the maintainability of XSL transformations"* from Karus and Dumas investigates the maintainability of Extensible Stylesheet Language Transformations (XSLT). Nowadays, XSLT is widely used, amongst others as message converters in enterprise applications, thereby raising the challenge of efficiently managing significant amounts of XSLT code. This paper addresses the question of whether the maintainability of XSLT transformations, measured in terms of code churn, can be predicted using a combination of simple metrics. Karus and Dumas present a set of statistical models that allow one to predict the maintainability of XSLT with relatively high accuracy.

Laval et al. consider that software maintainers should be able to assess multiple change scenarios for a given maintenance goal. In their paper *"Supporting simultaneous versions for software evolution assessment"*, they present Orion, an interactive prototyping tool for reengineering, that allows one to simulate changes and compare their impact on multiple version of the software source code models.

In the paper *"An open implementation for context-oriented layer composition in ContextJS"*, Lincke et al. focus on behavior adaptation. More specifically, they look at scenarios for behavior adaptation and they identify the need for a new scoping mechanism. They present an open implementation that allows developers to extend the context-oriented programming language core to suit their specific scoping needs.

Mannaert et al. present a theoretical approach to how the concept of systems theoretic stability can be applied to the evolvability of software architectures in their paper *"The transformation of requirements into software primitives: Studying evolvability based on systems theoretic stability"*. They show that systems theoretic stability can assist in striving towards establishing an objective and scientific foundation to analyze the evolvability characteristics of information systems.

In the paper *"A framework for evolution of modeling languages"*, Meyers and Vangheluwe discuss that not only individual software systems, the so-called instance models, evolve, but also the domain-specific languages that are used to generate software systems change throughout time. This does not come as a surprise, as the domain-specific languages are used in very specific application domains, which also undergo continuous changes. Meyers and Vangheluwe present a framework that describes all the steps that are required to allow (semi-)automated evolution of modelling languages.

In their paper *"Unifying Design and Runtime Software Adaptation Using Aspect Models"*, Parra et al. introduce an approach to unify adaptation at design and runtime based on aspect-oriented modelling. Their approach proposes a unified aspect meta-model and a platform that realizes two different weaving processes to achieve design and runtime adaptations. They use their approach in a dynamic software product line which derives products that can be configured at design time and adapted at runtime in order to dynamically fit new requirements or resource changes.

In *"Applying a dynamic threshold to improve cluster detection of LSI"*, van der Spek and Klusener focus on improvements to the Latent Semantic Indexing (LSI) approach that is often used for extracting and representing the meaning of words in a large set of documents, of which source code can be an example. In particular, they focus on the tree cutting strategy of LSI, which plays an important role in obtaining the clusters in LSI. In this contribution, the authors compare two tree cutting strategies: the Dynamic Hybrid cut and the commonly used fixed height threshold.

We hope you enjoy reading this special issue.

Acknowledgements

Both the BENEVOL series of seminars and this special issue are supported by the Interuniversity Attraction Poles Programme – Belgian State – Belgian Science Policy, project MoVES.

References

[1] A. Zaidman, M. Pinzger, A. van Deursen, Software evolution, in: P.A. Laplante (Ed.), Encyclopedia of Software Engineering, Taylor & Francis, 2010, pp. 1127–1137.
[2] T. Mens, S. Demeyer (Eds.), Software Evolution, Springer, 2008.
[3] T. Mens, Introduction and roadmap: history and challenges of software evolution, in: Software Evolution, Springer, 2008, pp. 1–11.
[4] J. Bosch, Software variability management, in: Proceedings of the 26th International Conference on Software Engineering (ICSE), IEEE Computer Society, Washington, DC, USA, 2004, pp. 720–721.
[5] M. Fayad, M. P. Cline, Aspects of software adaptability, Communications of the ACM 39 (10) (1996) 58–59.

Andy Zaidman *
Software Engineering Research Group,
Delft University of Technology,
Mekelweg 4, 2628 CD Delft, The Netherlands
E-mail address: a.e.zaidman@tudelft.nl.

Johan Brichau
Département d'Ingénierie Informatique (INGI),
École Polytechnique de Louvain,
Université catholique de Louvain (UCL),
Place Sainte Barbe, 2, B-1348 Louvain-la-Neuve, Belgium
E-mail address: johan@inceptive.be.

Available online 22 April 2011

* Corresponding editor.