Editorial

Introduction to the special issue on software analysis, evolution and reengineering

Abstract

Software analysis, evolution, and reengineering are important areas of the software life cycle. The quest to maintain and keep operational large mission critical systems has always been a challenge for software practitioners. This special issue presents a compilation of papers covering six major areas namely, program understanding, tools and environments, source code modeling, component recovery, migration to network-centric platforms, and binary reverse engineering and reengineering. These papers are re-worked and extended versions of papers that appeared in the Working Conference on Reverse Engineering. We hope that the readers will find this collection of papers useful as it provides an indicative view of the current work in the areas of software analysis, evolution, and reengineering.

© 2006 Published by Elsevier B.V.

Keywords: Software engineering; Software reverse engineering; Software evolution; Software reengineering

1. Introduction

Software analysis, evolution, and re-engineering are areas that have gained significant attention in the software industry and in academia over the past decade. The Y2K problem and, the Euro conversion problem, are just two of the many reengineering and evolution problems that affected the software industry in a world-wide scale. Software analysis, evolution, and reengineering techniques have matured significantly over the past years and have fostered strong technology links with several other areas of Software Engineering such as parsing and compiler technology, software modeling and visualization, pattern analysis and integrated development environments, to name a few. Furthermore, more recently, the introduction of iterative and incremental development models such as, Extreme Programming (XP) and the Unified Process (RUP) are now providing the opportunity for a new direction in the area of software analysis, evolution, and re-engineering. Techniques in these areas can be also applied for the development of new systems and not merely for the migration of existing legacy systems. In the context of such iterative and incremental models, software development may take the form of analysis, evolution and transformation of models that pertain to a wide spectrum of software artifacts such as business models, requirements models, architecture and design models, and implementation and testing models. In this respect, we anticipate that new and exciting research directions can be spawned whereby software analysis and evolution techniques can be incorporated to such Greenfield forward engineering models.

In our quest to capture the core areas of software analysis, evolution, and reengineering this special issue provides a collection of indicative papers and research work conducted in six major areas, namely:

- Program understanding: This area deals with techniques to support source code analysis and design recovery.
- Tools and environments: This area focuses on practical approaches and tool support to system analysis and reengineering.
• Source code modeling and representation: This area deals with techniques to model source code at a higher level of abstraction.
• Component recovery: This area deals with techniques for the architectural analysis and modularization of large software systems.
• System evolution to network-centric platforms: This area deals with techniques that support the evolution of legacy systems to Web enabled environments.
• Reverse engineering of binary code: This area deals with techniques to analyze binary code and decompile complex systems.

The readers may also find interesting articles and information on the software analysis and reengineering community in the Reengineering Forum [7] site, and the IEEE Technical Committee on Software Engineering-Committee on Reverse Engineering and Reengineering site [4]. In the following section we provide an overview of the papers that appear in this Special Issue, and we outline the challenges these papers aim to address. We believe that the readers will find the collection of these papers interesting and that the collection will be a stepping stone for new research activities in these areas to be initiated.

2. Overview of the papers

Program understanding has long been recognized as a central activity for Reverse Engineering and Software Maintenance. In 1989, Corbi recognized that up to 50 per cent of the Maintenance effort was devoted to program understanding tasks [1] while more recent studies indicated that even more time is involved, up to even 70 per cent. Program understanding is defined as the task of “building mental models of the underlying software at various abstraction levels, ranging from models of the code itself to ones of the underlying application domain” [6]. Therefore, it is the process through which the developer identifies, depending on the precise maintenance task s/he is facing, what the program does, how it works, what is its architecture, or which resources it uses. Research in program understanding has so far mainly focused on two questions: which kind of representation of the program is susceptible to help a developer recover the appropriate models during her/his tasks and how can such representations be automatically built by tools.

The article Software Visualizations for Improving and Measuring the Comprehensibility of Source Code by D.A. Umphress, D. Hendrix, J.H. Cross II and S. Maghsoodloo from Auburn University (AL) introduces two variants of software visualizations and experimentally determines how they help program understanding. More specifically, the Control Structure Diagram was analyzed for its aid in understanding the structure and execution of programs, while the Complexity Profile Graph was analyzed for its ability to measure source code comprehensibility. Both proved to be efficient and were included in the GRASP IDE project. However, effectiveness of software visualizations has still to be demonstrated in an industrial setting. Efforts toward industry adoption of software visualization tools are the perspective of the Auburn research group.

The article Documenting Software Systems Using Types by A. van Deursen and L. Moonen from CWI and Delft University (The Netherlands) proposes a solution to automatic program understanding tool building, in the context of Cobol programs. It introduces TYPEEXPLORE, a tool for browsing COBOL legacy systems based on inferred types, that is those types that are not specifically defined in a weakly typed language like COBOL, but that are obviously part of the program. TYPEEXPLORE first extracts information from source files, analyzes it to infer types and then builds a hypertext-based documentation, that can be browsed in order to discover the low level structure of individual variables as well as the global overview of the overall system architecture. An important perspective of this project is to support the migration of legacy COBOL-85 code to the new, object-oriented, COBOL standard. Over the years, several research groups developed program analysis techniques and tools. Early, researchers began working together, in order to cross-fertilize their results and avoid reinventing the wheel, and therefore exchanged data. The crucial problem they faced at this point was the need for tool interoperability, that is the ability for a tool to import data from another tool and to export its results to another tool. A new research area in itself was born [8], as recognized by the first Workshop on Standard Exchange Format, associated with ICSE in 2000 and in a related follow-up Dagstuhl event [2].

The article GXL: A Graph-Based Standard Exchange Format for Reengineering by R.C. Holt, A. Shuerr, S.E. Sim and A. Winter from University of Waterloo (Canada), Darmstadt University of Technology (Germany), University of California (CA) and University of Koblenz-Landau (Germany) respectively, describes this collaborative research
effort and introduces the GXL language which has been adopted as standard exchange format. This is a major step towards unified source code modeling and representation for software analysis. GXL is an XML-based language used to represent typed, attributed, directed, ordered graphs as well as hyper graphs and hierarchical graphs. Instance graphs can therefore be expressed in GXL and exchanged, together with their corresponding schema information. The article presents the underlying requirements for a standard exchange format, specifies GXL and discusses exchanging graphs using GXL. Currently, the main efforts are related to the definition of reference schemas, for representing software artifacts such as abstract syntax trees. The broader adoption of such reference schemas can greatly facilitate the integration of reengineering tools.

Reverse Engineering is not only a theoretical research discipline but also an applied field. As such, it proposes real solutions to business problems, the most important one being to minimize costs of software development. Early in 1968, McIlroy introduced the idea of software reuse [5] that proposes to reduce development costs by reusing (pieces of) software: when software must be moved to a new technology (from terminal mode to graphical interfaces in the 1995s, or later to the Web) or to a new platforms, and when a new software must be developed to resemble (at least partly) the existing application pieces (called components) of the existing software should be extracted and used as building blocks to form the new software. We have been a long way since this first paper on components by focusing on models and factories to develop applications from components [3]. However, in the context of software reengineering, two main problems must still be addressed: one is to identify which parts of the existing software are potentially reusable, another is to develop techniques enabling construction of software around building blocks of already developed pieces of code.

The article *Revisiting the dIC Approach to Component Recovery* by R. Koschke and J. Czernanski from University of Stuttgart (Germany) and G. Canfora from RCOST (Italy), presents a solution to the problem of identifying potentially reusable pieces of code. More specifically, this article relies upon the dIC approach defined by Canfora that detects abstract data objects; the dIC approach has been extended to also detect abstract data types and to integrate a good cohesion metric to identify candidate components. A strong evaluation of the approach is presented, that shows the superiority of this approach. Since literature shows that the effectiveness of any component recovery method greatly depends on characteristics of the subject system, further research has been planed to study the relationships between component recovery approaches and characteristics of systems.

The article *Synthesizing and Integrating Legacy Components as Services using Adapters* by S.V. Mudiam, G.C. Gannod and T.E. Lindquist from Arizona State University (AZ), proposes a solution to build software by reusing existing pieces of code and by migrating legacy components to network-centric platforms. Specifically, it proposes a view of components as services and to synthesize services from legacy components. Client applications can then be built to interact with these services that are seen as providers of data and behavior. The technique utilizes an architecture description language to describe components as services and achieves run-time integration of services, using wrappers or adapters to generate the ‘glue code’. Current work involves developing an environment that will assist in the creation of service-based applications by providing service browsing and the migration of the proposed techniques to the .NET and web service frameworks. All the approaches mentioned so far rely on the assumption that source code is available for analysis, allowing for the use of many standard technologies, often based on compiling, like abstract syntax tree construction or dependence identification. Unfortunately, in many real life industrial applications, program analysis must sometimes commence without having full access to the source code. Binary analysis may be then the only valid alternative, that is analyzing directly the binary machine code file of the program.

In this respect, the article *Dynamic Re-engineering of Binary Code with Runtime Feedbacks* by D. Ung from University of Queensland (Australia) and C. Cifuentes from Sun Microsystems Labs (CA), presents work in the field of binary code analysis and reengineering. Binary translation is a technique that allows software compiled for one machine to be converted to run efficiently on another machine. This re-engineering activity is decomposed in a reverse engineering phase, where the binary instructions are decoded to a higher level of abstraction, followed by a forward engineering phase, during which the abstraction is encoded into another binary form. This article presents the UQDBT binary translator which provides for adaptability through specification of machine properties and their instruction sets, performs optimizations that apply to a variety of machines and handles in a special way frequently executed code, allowing for faster translation compared to other translators. Results obtained in the x86 and SPARC platforms are also provided that show that generated programs run faster than when obtained with other techniques. However, further optimization is still under investigation, especially the removal of processor condition codes.
References


Francoise Balmas
Université Paris 8 Laboratoire IA 2,
rue de la Liberté 93526 Saint-Denis Cedex,
France

Kostas Kontogiannis
University of Waterloo,
Department of Electrical and Computer Engineering,
200 University Ave. West,
Waterloo ON,
N2L 3G1 Canada