— Guest Editorial —
Software Reverse Engineering

Arie van Deursen [a,b] Liz Burd [c]

[a]*CWI, P.O. Box 94079, 1090 GB Amsterdam, The Netherlands*

[b]*Delft University of Technology, Faculty of Electrical Engineering, Mathematics, and Computer Science, Mekelweg 4, Delft, The Netherlands*

[c]*Durham University, Durham, UK*

Introduction Reverse engineering aims at obtaining high level representations of programs. Reverse engineers typically start with a low level representation of a system (such as binaries, plain source code, or execution traces), and try to distill more abstract representations from these (such as, for the examples just given, source code, architectural views, or use cases, respectively). Reverse engineering methods and technologies play an important role in many software engineering tasks, such as program comprehension, system migrations, and software evolution.

Reverse Engineering Dimensions Reverse engineering is concerned with the following key questions:

(1) What are suitable high-level representations of software systems?
(2) How can a given high-level representation be obtained in a systematic manner from a system's sources or execution traces?
(3) How can the obtained representation be presented to software engineers to help them in their day to day software development work?

When reading the papers selected for this special issue, you will encounter representations such as module dependence graphs, subsystem partitionings, program dependence graphs enriched with symbolic execution information, sequential constrains on function calls, and thread interactions. The method used to reconstruct these vary from genetic algorithms to program analysis techniques. The applications of the various papers include remodularization, software miniaturization, and

Email addresses: `Arie.van.Deursen@cwi.nl` (Arie van Deursen), `Liz.Burd@durham.ac.uk` (Liz Burd).
URL: `http://www.cwi.nl/~arie/` (Arie van Deursen).

protocol validation. In some cases existing systems are actually altered (leading to *reengineering* or *renovation*): in other cases the high level representations obtained are primarily used for program comprehension or validation purposes.

As illustrated by the papers in this issue, the field of reverse engineering is inherently practical in nature: existing software systems are subjected to analysis and an attempt is made to extract as much useful information from them as possible.

An interesting side effect of reverse engineering research is that we try to learn something about the nature of programming. We can analyze a traditional procedural system written in C or Cobol, and then we can actually find structures that are akin to objects, aspects, contracts, design patterns, and so on. It appears that these concepts are inherent to software, and that making them explicit (as reverse engineering does), can help in understanding and manipulating the software, independent of the fact whether these concepts were "designed in" when the system was being built.

This Issue The papers in this special issue provide an interesting cross section of the field of reverse engineering, addressing remodularization, renovation, two flavors of program analysis, and concurrent programming.

Remodularization The first paper of this special issue deals with finding optimal subsystem decompositions for existing systems by analyzing module dependencies. Earlier work by Mancoridis and Mitchell describes an algorithm, implemented in a tool called Bunch, which uses metaheuristic search techniques such as hill climbing, simulated annealing, and genetic algorithms. Their approach aims at optimizing an objective function that characterizes the trade-off between coupling and cohesion. Since the algorithm applied uses heuristic search techniques, it is not clear how close to the optimal solution the results from Bunch actually are. The first paper in this special issue by Shokoufandeh et al exactly addresses that issue. It does so by recasting the clustering problem as a graph bisection problem. The paper's conclusion is that Bunch yields answers within a bounded approximation of the optimal solution, and that it does so efficiently.

Renovation The paper by Di Penta *et al.* is also related to remodularization. In particular, the authors aim at applying a number of reengineering techniques to actually improve the internal structure of software libraries. The techniques adopted include remodularization using a genetic algorithm, as well as the removal of duplicated code by means of clone detection. These techniques have been applied to an open source geographical information system comprising one million lines of code. The authors report a significant improvement in the system's internal organization. Moreover, the number of library objects linked by each application was reduced by

50% on average, yielding significant savings in the application's memory requirements.

Program conditioning The paper by Danicic *et al.* addresses *program conditioning*, an analysis technique that can be used in order to simplify a program under certain conditions. Program conditioning is akin to program slicing, and aims at removing as many statements as possible if we take into account that a certain condition of interest holds at some point in the program. It can be used for program understanding purposes (which statements are relevant if I know that the `age` variable must be higher than 65?) and for eliminating infeasible paths (dead code).

Program conditioning involves symbolic execution (`age > 65`) as well as theorem proving (having both `age > 65` and `age < 50` is inconsistent). The paper reports how the existing FermaT theorem prover can be used in a simple way for these purposes, and explains the symbolic execution algorith adopted. The technique has been applied to several example programs illustrating the feasibility and potential savings that can be obtained from using the approach.

Static Traces The paper by Eisenbarth *et al.* deals with static program analysis as well. In this paper the objective is to identify sequential constraints on function invokations. Such constraints can be used for the purpose of protocol reconstruction and validation. For many components, the underlying protocols are implicit, which may well lead to improper use of the component.

The identification of the order in which functions are to be applied is often done by analyzing traces obtained from actually executing the program. This has the disadvantage that suitable test data has to be generated. The present paper takes a *static* approach, analyzing the function calls in the source code. The paper addresses the various problems raised by (pointers to) stack and heap objects, resulting in *object process graphs* summarizing all possible sequences.

Static object trace extraction has been applied to several sizable case studies, including the source code of the Apache web server which comprises over a 100,000 lines of code.

Discovering Thread Interactions The final paper of this special issue, by Cook and Du, deals with dynamic analysis. It looks at the way in which threads interact in a concurrent system, and tries to make these interactions explicit rather than implicit. Two types of interaction are recognized: mutual exclusion and synchronization. While discovering, for example, mutual exclusion areas may be done statically by identifying all `synchronized` methods in a Java system, the dyanmic analysis proposed will identify other areas as well that may also exhibit mutual exclusion. If this is unintended this may well lead to performance problems.

The techniques proposed have been implemented in a tool called SyncMex, and applied to the Spark98 sparse matrix kernel which uses parallel computations to compute sparse matrix vector products.

Paper Selection Procedure The articles in this special issue were selected from the full set of papers and presentations given at the 9th Working Conference on Reverse Engineering (WCRE) held in Richmond, Virginia, from 29 October until November 1st, 2002, the proceedings of which were published by the IEEE Computer Society. Based on WCRE review reports and discussions during the conference, an initial list of eight papers was selected. These were subsequentely subjected to two regular rounds of full reviewing, resulting in the five papers presented in this special issue.

Acknowledgements We would like to thank all authors (including those whose papers were eventually not selected) and all reviewers for the effort they made for this special issue. We would also like to thank David Card, the editor in chief of the Journal of Systems and Software, for his encouragement and assistance in preparing this special issue.