New Frontiers of Reverse Engineering

Gerardo Canfora and Massimiliano Di Penta

Gerardo Canfora is a full professor of computer science at the Faculty of Engineering and the Director of the Research Centre on Software Technology (RCOST) of the University of Sannio in Benevento, Italy. He serves on the program committees of a number of international conferences. He was a program co-chair of the 1997 International Workshop on Program Comprehension; the 2001 International Conference on Software Maintenance; the 2003 European Conference on Software Maintenance and Reengineering; the 2005 International Workshop on Principles of Software Evolution: He was the General chair of the 2003 European Conference on Software Maintenance and Reengineering and 2006 Working Conference on Reverse Engineering. Currently, he is a program co-chair of the 2007 International Conference on Software Maintenance. His research interests include software maintenance and reverse engineering, service oriented software engineering, and experimental software engineering. He has co-authored more than 100 papers published in international journals and referred conferences and workshops. He was an associate editor of IEEE Transactions on Software Engineering and he currently serves on the Editorial Board of the Journal of Software Maintenance and Evolution. He is a member of the IEEE Computer Society

Massimiliano Di Penta is assistant professor at the University of Sannio in Benevento, Italy and researcher leader at the Research Centre On Software Technology (RCOST). He received his PhD in Computer Engineering in 2003 and his laurea degree in Computer Engineering in 1999. His main research interests include software maintenance, reverse engineering, empirical software engineering, and service-oriented software engineering. He is author of about 80 papers published on referred journals, conferences and workshops. He is program co-chair of the 14th Working Conference on Reverse Engineering (WCRE 2007) and of the 9th International Symposium on Web Site Evolution (WSE 2007), and steering committee member of STEP. He has been program co-chair of WCRE 2006, SCAM 2006 and STEP 2005. He serves and has served program and organizing committees of conferences and workshops such as CSMR, GECCO, ICSM, IWPC, SCAM, SEKE, WCRE, and WSE. He is member of the IEEE, the IEEE Computer Society and of the ACM.

Future of Software Engineering(FOSE'07)
0-7695-2829-5/07 $20.00 © 2007 IEEE

IEEE COMPUTER SOCIETY

New Frontiers of Reverse Engineering

Gerardo Canfora and Massimiliano Di Penta
RCOST - University of Sannio, Benevento, Italy
canfora@unisannio.it dipenta@unisannio.it

Abstract

Comprehending and modifying software is at the heart of many software engineering tasks, and this explains the growing interest that software reverse engineering has gained in the last 20 years. Broadly speaking, reverse engineering is the process of analyzing a subject system to create representations of the system at a higher level of abstraction. This paper briefly presents an overview of the field of reverse engineering, reviews main achievements and areas of application, and highlights key open research issues for the future.

1 Introduction

Many software engineering activities entail dealing with existing systems. Software maintenance, testing, quality assurance, reuse, and integration are only a few examples of software processes that involve existing systems. A key aspect of all these processes is the identification of the components of a system and the comprehension of the relationships existing among them. The term reverse engineering encompasses a broad array of methods and tools related to understanding and modifying software systems.

In the context of software engineering, the term *reverse engineering* was defined in 1990 by Chikofsky and Cross in their seminal paper [28] as *the process of analyzing a subject system to (i) identify the system's components and their inter-relationships and (ii) create representations of the system in another form or at a higher level of abstraction*. Thus, the core of reverse engineering consists in deriving information from the available software artifacts and translating it into abstract representations more easily understandable by humans. Of course, the benefits are maximal when supported by tools.

The IEEE-1219 [60] standard recommends reverse engineering as a key supporting technology to deal with systems that have the source code as the only reliable representation. Reverse engineering goals are multiple, e.g., coping with complexity, generating alternate views, recovering lost information, detecting side effects, synthesizing higher abstractions, and facilitating reuse. Examples of problem areas where reverse engineering has been successfully applied include redocumenting programs [11] and relational databases [94], identifying reusable assets [23], recovering architectures [67], recovering design patterns [2, 53, 68, 108], building traceability between code and documentation [1, 76], identifying clones [7, 10, 62, 81], code smells [111] and aspects [79, 106], computing change impacts [6], reverse engineering binary code [29], renewing user interfaces [84, 89], translating a program from one language to another [19], migrating [22] or wrapping legacy code [99]. Although software reverse engineering originated in software maintenance, its definition is sufficiently broad so as to be applicable to many problem areas, for example to create representations necessary for testing purposes [82], or to audit security and vulnerability [34].

This paper provides a survey (not intended to be exhaustive) of existing work in the area of software reverse engineering, discusses success stories and main achievements, and provides a road map for possible future developments in the light of emerging trends in software engineering.

The paper is organized as follows. Section 2 provides a primer of reverse engineering terminology and success stories. Section 3 reports the main achievements of reverse engineering during the last decade, organized in three main areas: program analysis, design recovery, and visualization. Section 4 uses the same three areas to identify and discuss issues that are likely to be the challenges for reverse engineering in the next few years. Section 5 identifies reverse engineering challenges that derive from emerging computing paradigms, such as service-oriented computing and autonomic computing. The issue of easing the adoption of reverse engineering is discussed in Section 6. Finally, Section 7 concludes the paper.

2 A Primer on Reverse Engineering

Reverse engineering has been traditionally viewed as a two step process: information extraction and abstraction. Information extraction analyses the subject system artifacts

Future of Software Engineering(FOSE'07)
0-7695-2829-5/07 $20.00 © 2007 IEEE

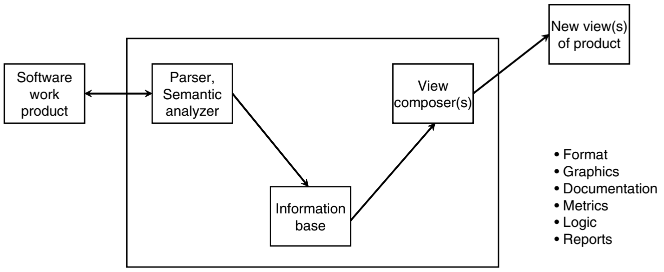

Figure 1. Reverse engineering tools architecture [28]

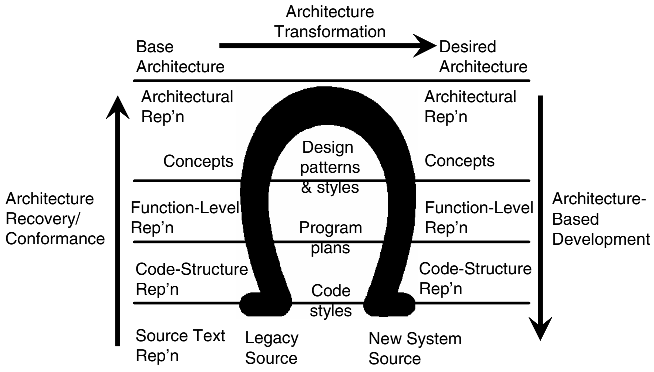

Figure 2. Architecture Reengineering: The Horseshoe model [64]

to gather raw data, whereas abstraction creates user-oriented documents and views. For example, information extraction activities consist of extracting Control Flow Graphs (CFGs), metrics or facts from source code. Abstraction outputs can be design artifacts, traceability links, or business objects. Accordingly, Chikofsky and Cross outlined a basic structure for reverse engineering tools (cf. Figure 1). The software product to be "reversed" is analyzed, and the results of this analysis are stored into an information base. Such information is then used by view composers to produce alternate views of the software product, such as metrics, graphics, reports, etc.

Most reverse engineering tools aim at obtaining abstractions, or different forms of representations, from software system implementations, although this is not a strict requirement: as a matter of fact, reverse engineering can be performed on any software artifact: requirement, design, code, test case, manual pages, etc. Reverse engineering approaches can have two broad objectives: *redocumentation* and *design recovery*. Redocumentation aims at producing/revising alternate views of a given artifact, at the same level of abstraction, e.g., pretty printing source code or visualizing CFGs. As defined by Biggerstaff [12], design recovery aims at recreating design abstractions from the source code, existing documentation, experts' knowledge and any other source of information.

Strictly speaking, reverse engineering does not include *restructuring*, which is the transformation from one representation form to another (e.g., source-to-source transformations) nor *reengineering*, which encompasses the alteration of the subject system to reconstitute it in a new form (e.g., migration). In particular, reengineering includes a reverse engineering phase in which an abstraction of the software system to be reengineered is obtained, and a forward engineering phase, aimed at restructuring the software system itself. Overall, a reengineering process can be viewed as a horseshoe model (cf. Figure 2) as the one used by Kazman *et al.* to describe a three-step architecture reengineering process [64]. The first step — represented on the left-side of the horseshoe — aims at extracting the architecture from source code. The second part of the process — represented in the upper part of the horseshoe — is related to architecture transformation towards a target architecture. The last part — on the right side of the horseshoe — represents the instantiation of the new architecture. By looking at the horseshoe from the bottom to the top, it can be noted how reengineering proceeds at different levels of abstraction: code representation, function representation, and architecture representation.

2.1 Success Stories

During the last 20 years, a lot has been done in the field of reverse engineering. The main results have been disseminated in venues such as the Working Conference on Reverse Engineering (WCRE), the International Conference on Software Maintenance (ICSM), the International Conference on Program Comprehension (ICPC, formerly IWPC), the International Workshop on Source Code Analysis and Manipulation (SCAM), in ICSM and ICSE workshops, and in other major software engineering conferences and journals.

During the early nineties, reverse engineering research focused on problems mainly related to the analysis of procedural software to understand it and to cope with upcoming problems such as the Y2K problem. Many fact extractors (i.e., tools extracting intermediate representations from the source code and storing it into databases) were created. Examples include CIA [26] or the Software Refinery toolkit [80], that used an object-oriented database, called *Refine*, to store a fine grained program model in the form of an attributed Abstract Syntax Tree (AST).

The diffusion of object-oriented languages and the ad-

Future of Software Engineering(FOSE'07)
0-7695-2829-5/07 $20.00 © 2007 IEEE

vantages they introduced, suggested to reengineer existing procedural software towards object-oriented. Bearing this in mind, approaches were developed to identify objects into legacy code (e.g., [30]). While these approaches were very promising and were also complemented by techniques aimed at transforming one language into another [19], doubts have been raised about the comprehensibility and maintainability of the target programs, that often only have the syntax of the target language, without adhering to its philosophy.

Architecture recovery has also been a mainstream research during the nineties and the early part of the new millennium. Architecture aims at identifying components forming the architecture of software systems, and the relationships (connectors) between them. As pointed out in Koschke's PhD thesis [67], architecture recovery is important to improve understandability, to promote reuse, and to support and manage software evolution. Component recovery has been performed in the past using approaches such as metrics-based [86] or Formal Concept Analysis (FCA) [98].

Further work focused on proposing clustering metrics and heuristic-based techniques for component recovery. A survey of clustering techniques applied to component identification applied to software systems have been published in the past, for example by Tzerpos and Holt [110]. An approach relying on inter-module and intra-module dependency graphs to refactor software systems was proposed by Mitchell and Mancoridis [86]. In general, when using metrics-based approaches, component identification can be treated as an optimization problem [55].

Architecture recovery is a semi-automatic approach and, as such, can require manual intervention, like for *Rigi* [116]. One of the main problems to tackle when recovering an architecture is to capture differences between the source code organization and the mental model behind high-level artifacts. Murphy *et al.* proposed an approach named Software Reflexion Model to capture and exploit these differences. In his thesis, Koschke [67] proposed an approach to combine the different techniques by using a union of fuzzy sets, to benefit from each one, and combined these results with information obtained from system experts/developers. As discussed in Linda Wills' PhD thesis [114] and in many following papers, cliché matching is an effective strategy for architecture recovery. Fiutem *et al.* [47] proposed an approach to detect architectural connectors — such as pipes, shared memory, remote procedure call, socket, etc. — by using cliché matching over control flow and data flow information. Cliché matching was, more recently, used for semi–automatic architecture recovery by Sartipi [97], who developed an architecture recovery tool called *Alborz* and a query language named Architectural Query Language (AQL).

Substantial work was done in the area of data reverse engineering [94], with the aim of understanding and/or restructuring existing data structures or databases [14] and, above all, to cope with the Y2K problem. Thanks also to the use of reverse engineering techniques, maintainers were able to identify and maintain source code portions not compliant with the change of millennium. This helped to save billions of dollars and prevent major damage which the fault could have caused. This, and the fact that legacy systems survived the introduction of new technologies (e.g., the Web) thanks to reverse engineering and reengineering techniques, are significant examples of how industry greatly benefited of reverse engineering.

Relevant work has been performed in the area of program slicing, dicing and chopping. Over 25 years after the seminal paper by Weiser [112], many different kinds of slicing techniques have been developed, and studies have been carried out to compare different slicing approaches [35]. Robust and effective slicing tools, such as CodeSurfer [52] or Indus [61] are now available.

Last, but not least, these early years of reverse engineering have seen the development of approaches for binary reverse engineering, i.e., to extract the source code or high-level representation from the binary when the source code is not available [29].

3 A Decade of Achievements in Reverse Engineering

This section discusses the main achievements of reverse engineering in the last 10 years. Many of them comes after the Müller *et al.* FoSE 2000 paper [91], and actually tackle problems and challenges outlined in their paper. The discussion is organized around three main threads: program analysis and its applications, design recovery, and software visualization, and is intended as a baseline to draft the road map for future research.

3.1 Program analysis and its applications

The past years saw the development of several program analysis techniques and tools. Some of them rely on static analysis techniques, while recent years have seen an increasing use of dynamic analysis as an effective way to complement static analysis [44, 104]: although dynamic analysis can be expensive and incomplete, it is necessary to deal with many reverse engineering problems where just static analysis does not suffice.

Currently, several analysis and transformation toolkits are available to help reverse engineers in their tasks. For example, the *Design Maintenance System (DMS)* [9] developed by Semantic Designs, the TXL [33], or the *Stratego* toolkit [15]. These toolkits provide facilities for parsing the source code and performing rule-based transformations.

Future of Software Engineering(FOSE'07)
0-7695-2829-5/07 $20.00 © 2007 IEEE

Despite the effort put into the development of source code analysis tools, and despite the maturity of parsing technology, the diffusion of a wide number of programming languages dialects — a phenomenon known as the "500 language problem" [70] — or problems such as dealing with macros and preprocessor directives, highlighted the need for alternative source code analysis approaches. Moonen [88] developed the idea of source code analysis through island parsing and lake parsing, i.e., analyzing only the code fragments relevant for a particular purpose.

Some of the most adopted source code analysis tools that have been developed during recent years are fact extractors, able to extract, even without the need for a thorough source code parsing, relevant information from the source code. Notable examples are *MOOSE* [39], *SrcML* [31], *Columbus* [45], or *Bauhaus* [67]. The diffusion of different fact extractors outlined the need for common schema to represent them [46], such as GXL [115] or the FAMIX meta-model used by *MOOSE*.

In their FOSE 2000 paper, Müller *et al.* [91] highlighted the need for incorporating reverse engineering techniques into development environments or extensible editors like *Emacs*, in order to facilitate their adoption by maintainers. Today's development environments such as *Eclipse*[1] or *NetBeans*[2] strongly favor such an idea of integration. In fact, they permit the development of tools — including reverse engineering tools — as plugins integrated into the development environment, capable of interacting with other tools, e.g. the source code editor, and capable of accessing on-the-fly the AST of the source code file the programmer is currently writing.

A substantial work on program analysis has been made to deal with peculiarities introduced by object-oriented languages, e.g. with polymorphism. Examples are the use of points-to-analysis [13] to determine the set of possible methods that a method invocation refers to [85], or the use of points-to and data flow analysis to analyze exception handling [48].

An intriguing phenomenon that has inspired a lot of successful research effort is the presence of clones in software systems (Figure 3 shows an example of clones from the Linux Kernel). The outcome was the production of different techniques, i.e., token-based [7, 62], AST-based [10], metrics-based [81], each one ensuring different advantages, such as high precision (AST-based) or high recall (token-based), language independence, or the ability to detect plagiarism (metric-based). Empirical studies have been carried out to analyze the presence and the evolution of clones in large software systems [5, 50]: cloning percentage tends to remain stable: while new clones appear, old ones are refactored. Finally, contrary to common wisdom, it has been found that the presence of clones is not necessarily a bad smell and a harmful phenomenon [63]: provided that maintainers are aware of their presence, clones constitute a widely adopted mechanism to facilitate software development (e.g., templating a similar piece of code) and reuse. Clone removal, on the other hand, may be risky and undesired [32].

Aspect oriented programming represents one of the new frontiers of software development. It addresses the issue of crosscutting concerns, i.e., of features spread across many modules, with a new modularization unit, the *aspect*, that encapsulates them. To support the maintenance of crosscutting concerns, as well as to refactor them into aspects, it is needed to identify them into the source code. With this in mind, several aspect mining approaches have been developed. They are based on the analysis of method fan-in [79], or dynamic analysis of execution traces [106].

The text contained in software systems, either in the form of comments or of identifiers, has played a central role in reverse engineering. In particular, it has been used to recover traceability links between different software artifacts, using Vector Space Models and Probabilistic Ranking [1], or Latent Semantic Indexing (LSI) [76]. Also, textual analysis using Information Retrieval techniques has been used to perform a software quality assessment based on the similarity between identifiers and comments [73], to measure the conceptual cohesion of classes [77], or to perform semantic clustering [69].

More details on the state-of-the-art and on future research trends on program analysis are discussed in a FoSE paper by David Binkley [13].

3.2 Architecture and design recovery

While research during the early nineties focused on recovering high-level architectures or diagrams from procedural code, the diffusion of object-oriented languages, on one hand, and of the Unified Modeling Language (UML) on the other hand, introduced the need of reverse engineering UML models from source code.

Relevant work in this area was carried out by Tonella and Potrich [107]. First, they proposed a static approach to recover class diagrams. For object diagrams, it was needed to combine static and dynamic information. Tonella and Potrich showed that the static views, lacking in some flow propagation information, need to be propagated with (possibly incomplete) dynamic information containing bindings of class fields to objects. While Tonella and Potrich extracted sequence diagrams through a conservative, static analysis on data flow, Systä combined static and dynamic analysis for recovering UML diagrams [104]. Briand *et al.* [17] also used dynamic analysis to recover sequence diagrams for distributed systems and, by means of transforma-

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

[2] *http://www.netbeans.org*

```
linux-2.4.0/arch/mips/mm/init.c
mips
pte_t *get_pte_slow(pmd_t *pmd,
                    unsigned long offset)
{
        pte_t *page;

        page = (pte_t *) __get_free_page(GFP_KERNEL);
        if (pmd_none(*pmd)) {
                if (page) {
                        clear_page(page);
                        pmd_val(*pmd) =
                            (unsigned long)page;
                        return page + offset;
                }
                pmd_set(pmd, BAD_PAGETABLE);
                return NULL;
        }
        free_page((unsigned long)page);
        if (pmd_bad(*pmd)) {
                __bad_pte(pmd);
                return NULL;
        }
        return (pte_t *) pmd_page(*pmd) + offset;
}
```

```
linux-2.4.0/arch/mips64/mm/init.c
MIPS64
pte_t *get_pte_slow(pmd_t *pmd,
                    unsigned long offset)
{
        pte_t *page;

        page = (pte_t *) __get_free_pages(GFP_KERNEL, 0);
        if (pmd_none(*pmd)) {
                if (page) {
                        clear_page(page);
                        pmd_val(*pmd) =
                            unsigned long)page;
                        return page + offset;
                }
                pmd_set(pmd, BAD_PAGETABLE);
                return NULL;
        }
        free_pages((unsigned long)page, 0);
        if (pmd_bad(*pmd)) {
                __bad_pte(pmd);
                return NULL;
        }
        return (pte_t *) pmd_page(*pmd) + offset;
}
```

Figure 3. Example of clones in the Linux Kernel [5]

tions, they were able to recover detailed information such as conditions, messages exchanged and data flow.

Object-oriented development was accompanied by the diffusion of design patterns. From a reverse engineering perspective, identifying design patterns into the source code aims at promoting reuse and assessing code quality. Also in this case, both static techniques [2, 53, 68, 108] and dynamic techniques [58] have been used. All the techniques are based on cliché matching either on a portion of class diagrams (static approaches) or on execution traces (dynamic approaches).

Feature identification and location represent widely and successfully addressed research problems for reverse engineering. Feature location is a technique aimed at identifying subsets of a program source code activated when exercising a piece of functionality. Bearing this in mind, techniques using static analysis, dynamic analysis, and their combination, were proposed. Dynamic analysis techniques were proposed for example by Wilde and Scully [113]. A static technique based on Abstract System Dependencies Graph (ASDG) was proposed by Chen and Rajlich [25]. Eisenbarth *et al.* [41] combined the use of both static and dynamic data to identify features. After performing static analysis, they used FCA to relate features. Antoniol and Guéhéneuc [4] also combined static and dynamic analysis. In their approach, the trace profiling was followed by a knowledge-based filtering and by a probabilistic ranking aimed at identifying events relevant to a particular feature. Marcus and Poshyvanyk [78] used a completely different technique, based on the application of LSI on the source code. Their technique was then combined with the technique proposed by Antoniol and Guéhéneuc, producing better results than the two individual techniques [93].

Finally, the large diffusion of the Web during the last ten years has triggered the need for reverse engineering techniques tied to Web Applications (WAs). Apart from reverse engineering-related conferences, research in this area has been disseminated in the International Symposium on Web Site Evolution (WSE). WAs present peculiarities that require the need to develop new reverse engineering techniques, or to adapt the existing ones. The interaction with WAs happens through pages visualized in a browser and by sending data from the browser to the Web server. As an additional level of complexity, these pages — including the scripting code to be executed by the browser — are dynamically generated.

Tilley and Huang [105] compared the reverse engineering capabilities of commercial WA development tools. Clearly these capabilities did not suffice for reverse engineering needs, and more had to be done. A significant contribution to WAs reverse engineering was provided by Ricca and Tonella, who developed the ReWeb tool to perform analyses on web sites [95], extending to WAs traditional static flow analyses. Di Lucca *et al.* proposed an approach and a tool (WARE) to recover Conallens UML documentation from WAs [37]. Architecture recovery of WAs was addressed by Hassan and Holt [56]. The dynamicity of WAs was handled by Antoniol *et al.*, who proposed WANDA [3], a tool for dynamic analysis of WAs.

3.3 Visualization

Software visualization is a crucial step for reverse engineering. The way information is presented to the developer

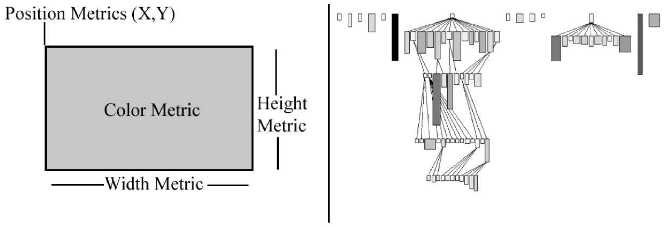

Figure 4. Polymetric views extracted using CodeCrawler [72]

or maintainer strongly impacts on the usefulness of the program analysis or design recovery techniques. In some cases, the choice of a proper visualization is straightforward, for example when the reverse engineering techniques aim at extracting well-defined, widely adopted diagrams from source code, e.g. UML diagrams, state machines, CFGs, etc. In other cases, a visualization constitutes the essence of a reverse engineering technique, for its ability to highlight relevant information at the right level of detail.

Examples of tools able to visualize statically extracted information include the *Rigi* tool [116], *CodeCrawler* [72], *Seesoft* [40], *SHriMP* [102], and *sv3D* [75]. Some of these tools — like *Rigi* or *SHriMP* — aim at showing architectural views. Other tools, like *sv3D*, provide a 3D visualization of software artifact metrics. *CodeCrawler* combines the capability of showing software entities and their relationships, with the capability of visualizing software metrics using *polymetric views*, which show different metrics using the width, the length and the color if the boxes (see Figure 4), and a similar approach, *class blueprints* [38], to visualize information about classes. Other visualization tools, such as *Program Explorer* [71], aim at visualizing dynamic information, e.g., information extracted from execution traces.

4 Future trends of Reverse Engineering

This section discusses future reverse engineering trends related to the three main areas of reverse engineering identified in Section 3.

4.1 Future trends in program analysis

One of the key challenges of program analysis for today and tomorrow is to deal with high dynamicity. Many programming languages widely used today allow for high dynamicity, which constitutes a powerful development mechanism, but makes analysis more difficult. For example, languages like Java introduce the concept of *reflection* and the ability of loading classes at run-time. This affects many analysis techniques, like static points-to analysis: with runtime class loading it is not possible to determine the set of objects a reference points to. Dynamic analysis is therefore required as a necessary complement to static analysis. On the other hand, mechanisms like *reflection* can ease some analysis tasks, for example providing access to fields and methods of a given class. Analysis facilities are also provided by the Java Virtual Machine (JVM^{TM}) 1.5 through the JVM Tool Interface (JVMTI): dynamic analysis can be performed by getting information from it rather than by instrumenting the source code. This avoids the need for a source code parsing/instrumentor and, above all, the need for the source code itself.

Another important program analysis challenge is represented by cross-language applications. New program analysis tools must be able to cope with the diversity of languages and technologies used to develop a single software system. If we consider, for example, WAs, they are often composed of HTML fragments, server-side scripting and client side scripting, database queries written in SQL. Works proposed by Moise *et al.* [87] and by Strein *et al.* [103] represent steps towards this direction.

New forms of programs will represent the source for future applications of reverse engineering. One example is represented by the need for analyzing artifacts produced by what Burnett *et al.* defined as *end-user programming* [18], i.e., the development of software by using productivity tools. Assets such as word processors and spreadsheets have to be considered as critical as the source code: errors in formulae or macros would cost millions of dollars.

Reverse engineering research has highlighted the dualism between static and dynamic analysis and the need to complement the two techniques, trying to exploit the advantages of both and limit their disadvantages. Both static and dynamic analysis techniques, however, can be used to analyze a software system configuration snapshot, ignoring how software evolves during the time. Nevertheless, it would be useful to understand how software artifacts change at a given release, whether some artifacts change together, whether such changes are correlated with other software characteristics (e.g., faultiness), etc. This kind of analysis is nowadays feasible thanks to the wide use of versioning systems like the Concurrent Versions System (CVS) or the Subversion (SVN), and of problem reporting systems such as Bugzilla.

Intense research has been made during the last few years in the analysis of these data sources, and a new, important research area has appeared: mining software repositories. Relevant work has been reported in software maintenance-related conferences, at the Mining Software Repositories (MSR) workshop, and in major journals [57]. Whilst this kind of studies were conceived as an effective way for

Future of Software Engineering(FOSE'07)
0-7695-2829-5/07 $20.00 © 2007 IEEE

studying software evolution, for analyzing the developers' behavior or to correlate different software characteristics — e.g., faultiness with metric profiles [54] — recently software repository information has been used as an alternative or complementary way of software analysis. For example, Geiger *et al.* [49] related clones with software repositories co-changes, and Canfora *et al.* with crosscutting concerns [21].

During the last 10 years, software analysis moved from a single dimension (static analysis) to two dimensions (static and dynamic analysis) and, finally, today's opportunity is to add a third dimension consisting in the historical analysis of data extracted from software repositories [24]. Change diffs between file revisions, change coupling relationships between files being co-changed during the same time window, but also relationships between bug reports and code repositories, and the change rationale as documented in problem fixing reports and in CVS messages, constitute valuable sources of information complementary to static and dynamic analysis.

4.2 Future trends in design recovery

Forthcoming research related to design recovery needs to be able to deal with design paradigms which analysts and software architects are currently using. While a lot of work has been done to extract UML documentation from the source code, a lot still needs to be done in particular for what regards the extraction of dynamic diagrams and also of Object Constraint Language (OCL) pre and post-conditions. On the other hand, there is the need to develop design recovery approaches and tools for new software architectures, that have characteristics of being extremely dynamic, highly distributed, self-configurable and heterogeneous. We will discuss these issues in Section 5.2 for the analysis of Service Oriented Architectures (SOA).

In Section 3 we highlighted the important role played by WA reverse engineering in the last ten years. Web applications are now moving towards Web 2.0, and this will have non-negligible effects on reverse engineering. For example there will be the need for techniques able to support the migration of existing WAs — composed of multiple pages interacting with the users by means of HTML forms — towards Web 2.0 applications where the user interacts with a single page. Further challenges will be related to the capability of reverse engineering to deal with the number of new technologies being used into new WAs.

Reverse engineering literature reports several approaches claiming to recover design or any higher-level artifact from the source code. Based on what is discussed in Section 3, design recovery aims not only at extracting information available in the source code to produce higher-level artifacts, but also at complementing it with information coming from developers' rationale. With this in mind, design recovery approaches proposed so far have two limitations:

1. They are either incomplete or imprecise, i.e., can compromise the recall to ensure good precision or *vice versa*;
2. They are semi-automatic, i.e., their application requires inputs from the human expert to complement or correct the information automatically extracted. When an approach is completely automatic, this compromises its precision or recall, or the obtained design is only partially usable by the maintainer.

Program comprehension tasks would greatly benefit from the possibility of interactively improving the way an artifact is presented to the maintainer while the latter provides feedbacks to the reverse engineering system.

Future activities in reverse engineering should push towards a tight integration of human feedbacks into automatic reverse engineering techniques. In other words, the reverse engineering machinery should be able to learn from expert feedbacks to automatically produce results. Machine learning, meta-heuristics and artificial intelligence make available plenty of mechanisms able to exploit feedbacks provided during the computation. To mention some examples, the Rocchio feedback mechanism can be used to interactively improve results of Vector Space Models. For example, Huffman Hayes *et al.* used relevance feedbacks to improve traceability recovery [59]. Also, heuristics such as Genetic Algorithms (GAs) provide a way for building fitness functions from user feedbacks. Such a mechanism is named Interactive GAs (IGAs) (see for instance, the work by Llora *et al.* [74]) and can be potentially applied to improve heuristic-based approaches to reverse engineering, such as clustering, design pattern recovery, and refactoring identification. Further details on the use of interactive fitness functions in software engineering can be found in Harman's FoSE paper [55].

Last, but not least, reverse engineering can move steps further from recovering design artifacts: requirements are also an important output that can be produced by reverse engineering [117].

4.3 Future Trends in Software Visualization

Despite the amount of work done in software visualization, there has been a long and repetitive discussion in the reverse engineering community regarding the usefulness of visualizations. Effective visualizations should be able to:

1. show the right level of detail a particular user needs, and let the user choose to view an artifact at a deeper

Future of Software Engineering(FOSE'07)
0-7695-2829-5/07 $20.00 © 2007 IEEE

level or detail, or to have a coarse-grain, in-the-large, view;

2. show the information in a form the user is able to understand. Simpler visualizations should be favored over more complex ones, like 3D or animations, when this does not necessarily bring additional information that cannot be visualized in a simpler way.

While other reverse engineering techniques can be validated in a relatively easy manner — e.g., analyzing open source systems that, with the time, became benchmarks for the comparison of some techniques — this is more difficult for visualizations. Experimentations aiming at investigating visualization usefulness, effectiveness, and scalability are more and more needed. It is important to also note that assessing visualization techniques require the study of the mental models programmers build when understanding software [101]. Finally, experimentation for assessing software visualizations can nowadays make use of advanced monitoring systems, for example eye-tracking tools, to monitor if and how a subject, during the experimental task, uses a particular visualization.

The mechanisms behind SOA or autonomic systems are quite complex and, as will be better explained in Section 5.2, their understanding is necessary to debug these systems in case of failures, or to perform maintenance. Visualization research should work more in this area, providing, for example, support to visualize dynamic service compositions, service binding and reconfiguration.

5 Reverse Engineering in Emerging Software Development Scenarios

Different areas of reverse engineering need to deal with the new development paradigms and technologies. The following subsections highlight a series of issues and challenges for reverse engineers. In our opinion, reverse engineering of the future years must be able to cope with:

1. on the one hand, the analysis of systems having high dynamism, distribution and heterogeneity and, on the other hand, support their development by providing techniques to help developers enable mechanisms such as automatic discovery and reconfiguration;

2. the need for a full integration of reverse engineering with the development process, which will benefit from on-the-fly application of reverse engineering techniques while a developer is writing the code, working on a design model, etc.

5.1 Continuous Reverse Engineering: Using Reverse during Forward Development

Müller *et al.* [91] highlighted the idea of exploiting information extracted by reverse engineering in the forward development process, i.e., making some information or artifacts, e.g., architectural views, design diagrams, traceability links, available to the developers by using reverse engineering techniques. Today's maturity of several pieces of reverse engineering technology, and the availability of extensible development environments enable the possibility of continuously performing reverse engineering while a software system is being developed. This idea has been applied in other areas of software engineering: for example Saff and Ernst [96] proposed the idea of continuous testing, i.e., of repeatedly executing unit test cases while developing or maintaining a piece of functionality for which the developer has already specified unit test cases.

In the context of reverse engineering, this would entail different benefits. First, by extracting and continuously updating high-level views (e.g., architectural views, or design diagrams such as class diagrams, sequence diagrams, statecharts) from the source code under development, it would be possible to provide the developer with a clearer picture of the system being created. Second, when artifacts at different levels of abstraction are available, a continuous consistency check between them could help to reduce development errors, for example checking whether the code is consistent with the design or complies with pre and post conditions. Third, a continuous analysis of the code under development and of other artifacts can be used to provide developers with useful insights, e.g., suggesting the use of a particular component or to improve the source code quality, for instance by improving the level of comments, by increasing its cohesion, etc.

One piece of reverse engineering technology that can be put in this context is clone detection. The developer can continuously monitor the presence of clones when committing files via a versioning system, or even while writing / maintaining the source code. For example, while a developer maintains a cloned piece of software, the development environment could warn her/him about the presence of similar artifacts that might need to be consistently changed. Also, differences between a single code fragment under development and other clones may highlight possible programming mistakes. In the same direction, it would be possible to continuously monitor the presence of bad smells [111] and to suggest refactoring opportunities, also considering that development environments provide support for automatic refactoring.

Traceability links tend to be lost during development and maintenance and, as described in Section 3, they can be recovered using techniques that assume the consistency of identifiers/terms across different software artifacts, from requirements to the source code. The same idea can be exploited to guide developers in properly naming identifiers and commenting the source code. For example, De Lucia *et al.* [36] developed a plugin capable of indicating the traceability level between high level artifacts and the code under development. Poorly named identifiers not only make traceability recovery difficult, but also make the code difficult to be understood. In the same direction it would be possible to automatically provide developers insights to name identifiers, learning naming conventions from existing code and extracting domain terms from requirements and design. Also, keeping track of some metrics, such as the Chidamber and Kemerer metrics [27] or the conceptual cohesion [77] would improve the quality of software under development, also and possibly reducing the presence of faults [54].

Continuous reverse engineering can be used to support code documentation and to keep documentation aligned during development. Maintenance / development actions over source code, design diagrams, test cases can trigger analysis on the other artifacts and highlight the need for consistency alignment or repairing (e.g., test cases might need to be repaired because the application is being maintained [83]).

In a similar fashion, there is space for the integration of many reverse engineering approaches in an agile development or reenginering process [20] as a continuous, on-the-fly feedback. This because (i) in agile processes the documentation is scarce or does not exist at all; (ii) the quality assurance is limited to some unit and acceptance testing; and (iii) above all,software products are incrementally created through small analysis-to-code iterations, that make the continuous update of reverse engineered artifacts highly desirable.

5.2 Reverse Engineering for Service Oriented Architectures and Autonomic Computing

Modern organizations worldwide strive for agility to keep competitive in a high pressure marketplace. SOA and autonomic computing are an emerging answer to this need: the first one focuses on the development of highly dynamic, inter-organizational systems, with a clear separation of the possession and ownership of software (software as a product) from its use (software as a service) [109]; the second one [65] promotes self-adaptation and self-evolution mechanisms in software systems.

As highlighted by Gold *et al.* [51], while SOA represent the next (today we may say the new) revolution of software development, they pose relevant software understanding issues. A service oriented system is composed of distributed services provided by different organizations. Each service offers, through its interface, a limited view of the features it makes available: providers “sell” the usage of a service but want to protect the technology behind it and the implementation details. Furthermore, when some documentation/specification is available, it cannot be ensured that different service providers, belonging to different organizations or different domains, use the same terminology, the same formalism, and even provide the same amount of information. All these factors affect the service understandability and, being the implementation not available for reverse engineering, black box understanding techniques such as those used in the past by Korel [66] need to be used.

An important issue, when dealing with services, particularly in the stage of orchestrating a set of services into a business process, is to comprehend how services relate to each other and what are the differences and commonalities among the operations they publish.

Service oriented systems and, more generally, systems with autonomic capabilities, comprise the ability to autonomously perform some actions like:

- *automatic discovery:* when a service, or even an entire piece of a service composition is not available or not capable of providing the requested level of functionality, automatically discover [92] new services, bind them or even modify the way they are composed, and continue the execution;
- *self-healing* [8]: the system execution is continuously monitored until some events trigger recovery actions such as changing a composition, reconfiguring/repairing the system, interrupting the execution, etc.

These features have important consequences on the system’s understandability and as a side effect, on its maintainability especially in case the self-healing or automatic discovery and composition mechanisms are not working properly. In this context, the limited observability of services is not the only challenge for analyzability: it is necessary to cope with the extremely high dynamism and with the fact that pieces composing the system may only be known at execution time. In other words, dynamic discovery and binding would represent a problem with analogies to points-to analysis, but with much more difficulties since, in this case, it is not even possible to determine *a-priori* the set of possible end points.

As also pointed out by Müller [90], continuously evolving systems can benefit from reverse engineering and reengineering techniques, that can be used to instrument evolving software-intensive systems with autonomic elements. On the other side of the coin, reverse engineering may provide an important contribution for the realization of autonomic systems and of SOA providing features such as dynamic discovery and binding. The realization of this kind of systems is, in fact, today limited because, in order to work properly, services/components have to expose information such as their semantics, the expected level of Quality of Service (QoS), their state machine, or even part of their internal state. In many cases this simply does not happen: anyone who publishes a service just exposes the signatures of its operations (in terms of a Web Service Description Language (WSDL) interface in the case of Web services) often automatically generated from the source code. Limited effort available, short time-to-market, limited competencies are — together with the lack of well defined and standard description languages — the main factors slowing down the diffusion of SOA.

Upcoming work in reverse engineering can support service providers to (semi) automatically produce service/component annotations capable of supporting automatic discovery and reconfiguration mechanisms. Of course this kind of automatically extracted information can be different from what today is assumed by the already developed automatic discovery and composition mechanisms, necessitating a step back and a re-thinking for some of them. A viable alternative to the use of semantic annotations for pursuing discovery, composition and reconfiguration can consist in some forms of execution traces, fingerprints extracted from the source code, part of the internal state of a component, or monitoring data. Reverse engineering is surely a great support for making this information available.

Last, but not least, SOA represent an opportunity to reverse engineering researchers to put together their efforts when building new tools and creating new pieces of research. Source code analysis and reverse engineering tools can be published as services so that every time one needs to realize a new tool can build part of it by just using some service. A reverse engineering tool can be realized by composing a parsing service, a service for identifying clones, one for instrumenting the source code, one for extracting UML documentation, etc. The biggest challenge that is today limiting this interoperability is the need for common schema/ontologies to represent tool input/outputs.

6 Favoring the Adoption of Reverse Engineering and its Tools

Despite the maturity of reverse engineering research, and the fact that many pieces of reverse engineering work seem to timely solve crucial problems and to answer relevant industry needs, its adoption in industry is still limited. In our opinion, there are a few directions in which it is important to push forward:

1. *Reverse engineering education:* most software engineering and computer science curricula comprise a number of design-process and design-methods courses. These courses teach engineering design fundamentals utilizing repeatable design techniques and processes. However, very few courses use reverse engineering as a means to teach designing a software system as it happens most of the times in the real-world, that is by evolving an existing system based on new and emerging requirements and needs. Teaching reverse engineering as an integral part of the software design process — and not only as techniques to handle changes — will increase the consciousness of the role of reverse engineering, thus helping to reduce the barriers, companies have to invest on reverse engineering or either to adopt reverse engineering practices;

2. *Achieving better empirical evidence:* especially during recent years, the majority of the reverse engineering research has been empirically validated with studies aiming at measuring the performance of a technique or to compare it with existing ones. Nevertheless, the empirical maturity of this research area still needs to be improved. In his keynote at WCRE 2006, Briand [16] identified three dimensions to evaluate reverse engineering research: inputs, analysis process, and outputs. Outputs need to be evaluated in terms of correctness, completeness and usefulness to carry out a specific task. The analysis process needs to be evaluated in terms of performance and scalability. A challenging problem for the evaluation of reverse engineering research is the need for benchmarks [42], which are necessary to compare similar research approaches. Examples of benchmarks have been successfully applied for feature location (e.g., XFig) or for evaluating C++ extractors [43]. Finally, human factors play a relevant role in the use of reverse engineering tools and, above all, in software understanding tasks, raising the need to run controlled experiments;

3. *Increasing tool maturity and interoperability:* it might be argued this may or may not be the role of researchers, however the availability of mature reverse engineering tools and their interoperability, by means of sharing data format and common fact schema [39, 115] or by the ability to be composed as services (as discussed in Section 5.2) will favor their usage and, consequently, their adoption. Finally, what discussed above regarding the empirical assessment of reverse engineering techniques need to account that, in many cases, results strongly depend on the maturity, robustness and usability of tools.

Future of Software Engineering(FOSE'07)
0-7695-2829-5/07 $20.00 © 2007 IEEE

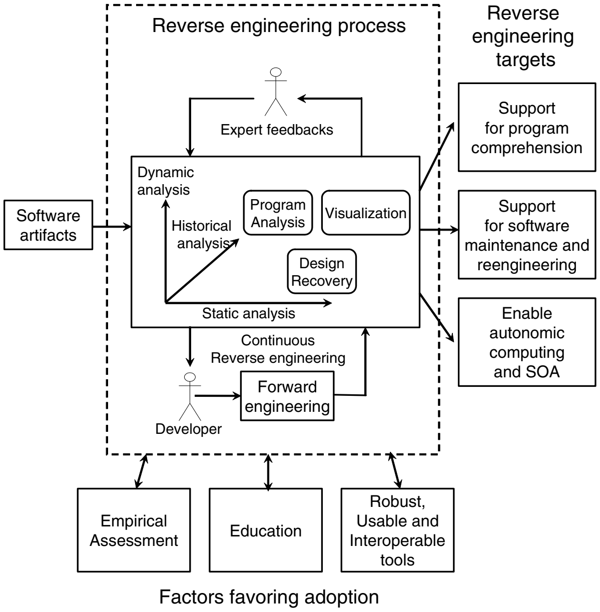
Reverse engineering process
Reverse engineering targets
Expert feedbacks
Dynamic analysis
Program Analysis
Visualization
Historical analysis
Design Recovery
Static analysis
Software artifacts
Support for program comprehension
Support for software maintenance and reengineering
Enable autonomic computing and SOA
Continuous Reverse engineering
Forward engineering
Developer
Empirical Assessment
Education
Robust, Usable and Interoperable tools
Factors favoring adoption

Figure 5. The role of reverse engineering

7 Concluding remarks

Figure 5 summarizes what could be, in our opinion, the role of reverse engineering in future. Forthcoming reverse engineering techniques will rely on information across three different and complementary dimensions: static analysis, dynamic analysis and, as a new dimension, historical analysis of artifact evolution. Today and tomorrow software will be characterized by high dynamicity, heterogeneity and distribution. This constitutes a big challenge for reverse engineering, and it would not be possible to reverse engineer such more and more complex software without properly combining information along the three dimensions.

Future reverse engineering will be part of the development process: it will be repeated over time on one and the same system while it is being developed / evolved for various purposes, from consistency checking to design rationalization. Human feedbacks will be fully integrated into the reverse engineering process, and this suggests that the traditional dichotomy between automatic and semi-automatic reverse engineering be overcame. Traditionally, automatic reverse engineering has forced to trade-off between precision and recall, whereas semi-automatic reverse engineering has used human feedback to improve the specific artifacts and views being produced, not the production process itself. We envision a new generation of reverse engineering techniques and tools with self-managing features. The basic idea is that the reverse engineer feedback is used to improve the reverse engineering process, not only a specific view; as a systems is continuously reverse engineered during its life, the engineers feedback can be used to capture and store background and implicit knowledge on both the system and the engineers. Thus, the results of a system reverse engineering could improve over time not only in terms of precision and recall, but also in terms of adequacy to the engineers style of work and task at hand. Building these features into future reverse engineering methods and tools poses many challenges to research: how could background and implicit knowledge be captured from the interaction with reverse engineers? How could engineers feedback be used to learn better ways of reverse engineering a system? What information need to be captured in the interaction with an engineer to build a profile useful to guide future reverse engineering? How could the knowledge of tasks be provided to a reverse engineering tool so that output be oriented towards the particular task at hand?

Another important aspect is related to the role reverse engineering will play in the development process. The diffusion of lightweight processes where requirement-to-code cycles represents a reality of today's software development practice. These processes however require (i) a continuous consistency check and alignment of software artifacts at different levels, and (ii) a continuous refactoring. The integration of reverse engineering tools into development environments will make this possible, enabling a tight form of continuous reverse engineering in which, during the forward phase, artifacts are continuously analyzed and, if necessary, transformed. Reverse engineering will continuosly present to the developer updated information, each time changing as the developer modifies the code, also reacting to previous feedbacks. On its own, the reverse engineering machinery improves the quality of the produced artifacts by exploiting developer feedbacks.

The adoption of reverse engineering is still limited and needs to be favored by means of three key enablers. First, better education on this discipline at different levels of scholarship is foreseen. Second, while reverse engineering research is quite solid and published research papers contain solid validation, further empirical studies will be necessary to compare techniques, to assess their usefulness and the usability of tools, and to understand in which circumstances particular techniques can be applied. The ultimate role of empirical studies would be on the one hand to assess existing reverse engineering theories, and, on the other hand, for developing new theories that, as for program comprehension [100] can be used to make sense of data and conclusions.

Finally, while industry should take the responsibility to develop solid reverse engineering tools exploiting the research that has been carried out, the production of usable prototypes and, above all, favoring the integration and the interoperability of different tools via common exchange for-

Future of Software Engineering(FOSE'07)
0-7695-2829-5/07 $20.00 © 2007 IEEE

mats and via emerging architectures like SOA, will be a key factor in promoting the diffusion of reverse engineering technology.

Reverse engineering will have three main targets. Two will be the usual ones, i.e., aiding software comprehension and providing support for maintenance and reengineering. In addition, new types of software systems are nowadays being developed: autonomic systems, or SOA allowing features such as automatic service discovery and reconfiguration. In this context, the role of reverse engineering would be to extract information needed to enable features such as automatic discovery and compositions, and to support software transformations necessary to make existing systems self healing and automatically reconfigurable.

8 Acknowledgments

The authors would like to thank Lerina Aversano, David Binkley, Luigi Cerulo, Mark Harman, Hausi Müller, and Aaron Visaggio for providing helpful comments on early drafts of this paper.

References

[1] G. Antoniol, G. Canfora, G. Casazza, A. De Lucia, and E. Merlo. Recovering traceability links between code and documentation. *IEEE Trans. Software Eng.*, 28(10):970–983, 2002.

[2] G. Antoniol, G. Casazza, M. Di Penta, and R. Fiutem. Object-oriented design patterns recovery. *Journal of Systems and Software*, 59(2):181–196, 2001.

[3] G. Antoniol, M. Di Penta, and M. Zazzara. Understanding web applications through dynamic analysis. In *12th International Workshop on Program Comprehension (IWPC 2004), 24-26 June 2004, Bari, Italy*, pages 120–131, 2004.

[4] G. Antoniol and Y.-G. Guéhéneuc. Feature identification: An epidemiological metaphor. *IEEE Trans. Software Eng.*, 32(9):627–641, 2006.

[5] G. Antoniol, E. Merlo, U. Villano, and M. Di Penta. Analyzing cloning evolution in the Linux Kernel. *Information and Software Technology*, 44:755–765, Oct 2002.

[6] R. S. Arnold and S. A. Bohner. Impact analysis - towards a framework for comparison. In *Proceedings of the Conference on Software Maintenance, ICSM 1993, Montréal, Quebec, Canada, September 1993*, pages 292–301, 1993.

[7] B. S. Baker. On finding duplication and near-duplication in large software systems. In *Proceedings of the Working Conference on Reverse Engineering*, pages 86–95, July 1995.

[8] L. Baresi, C. Ghezzi, and S. Guinea. Smart Monitors for Composed Services. In *Proc. 2nd International Conference on Service Oriented Computing (ICSOC'04)*, pages 193–202, New York, USA, November 2004. ACM.

[9] I. D. Baxter, C. Pidgeon, and M. Mehlich. DMS: program transformations for practical scalable software evolution. In *26th International Conference on Software Engineering (ICSE 2004), 23-28 May 2004, Edinburgh, United Kingdom*, pages 625–634, 2004.

[10] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. Clone detection using Abstract Syntax Trees. In *Proceedings of the International Conference on Software Maintenance*, pages 368–377, 1998.

[11] P. Benedusi, A. Cimitile, and U. de Carlini. Reverse engineering processes, design document production, and structure charts. *Journal of Systems and Software*, 19(3):225–245, 1992.

[12] T. Biggerstaff. Design recovery for maintenance and reuse. *IEEE Computer*, Jul 1989.

[13] D. Binkley. Source code analysis: A road map. In *ICSE - Future of SE Track*, 2007.

[14] M. R. Blaha. Dimensions of database reverse engineering. In *Proceedings of the Working Conference on Reverse Engineering*, pages 176–183, 1997.

[15] M. Bravenboer, K. T. Kalleberg, R. Vermaas, and E. Visser. Stratego/XT 0.16: components for transformation systems. In *Proceedings of the 2006 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-based Program Manipulation, 2006, Charleston, South Carolina, USA, January 9-10, 2006*, pages 95–99, 2006.

[16] L. C. Briand. The experimental paradigm in reverse engineering: Role, challenges, and limitations. In *Proceedings of the 13th Working Conference on Reverse Engineering (WCRE 2006), October 2006, Benevento, Italy*, pages 3–8, Los Alamitos, CA, USA, 2006. IEEE Computer Society.

[17] L. C. Briand, Y. Labiche, and J. Leduc. Toward the reverse engineering of uml sequence diagrams for distributed java software. *IEEE Trans. Software Eng.*, 32(9):642–663, 2006.

[18] M. Burnett, C. Cook, and G. Rothermel. End-user software engineering. *Commun. ACM*, 47(9):53–58, 2004.

[19] E. J. Byrne. Software reverse engineering. *Softw., Pract. Exper.*, 21(12):1349–1364, 1991.

[20] M. I. Cagnin, J. C. Maldonado, F. S. R. Germano, and R. Dellosso Penteado. PARFAIT: towards a framework-based agile reengineering process. In *2003 Agile Development Conference (ADC 2003), 25-28 June 2003, Salt Lake City, UT, USA*, pages 22–31, 2003.

[21] G. Canfora, L. Cerulo, and M. Di Penta. On the use of line co-change for identifying crosscutting concern code. In *Proceedings of the 22nd International Conference on Software Maintenance (ICSM 2006), September 2006, Philadelphia, PA, USA*, pages 213–222, Los Alamitos, CA, USA, 2006. IEEE Computer Society.

[22] G. Canfora, A. Cimitile, A. De Lucia, and G. A. Di Lucca. Decomposing legacy programs: a first step towards migrating to client-server platforms. *Journal of Systems and Software*, 54(2):99–110, 2000.

[23] G. Canfora, A. Cimitile, and M. Munro. Reverse engineering and reuse re-engineering. *Journal of Software Maintenance and Evolution - Research and Practice*, 6(2):53–72, 1994.

[24] L. Cerulo. *On the Use of Process Trails to Understand Software Development*. PhD thesis, RCOST - University of Sannio, Italy, 2006.

[25] K. Chen and V. Rajlich. Case study of feature location using dependence graph. In *Proceedings of the International Workshop on Program Comprehension*, pages 241–250. IEEE Computer Society, 2000.

[26] Y.-F. Chen, M. Y. Nishimoto, and C. V. Ramamoorthy. The C information abstraction system. *IEEE Trans. Software Eng.*, 16(3):325–334, 1990.

[27] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Trans. Software Eng.*, 20(6):476–493, 1994.

[28] E. Chikofsky and J. I. Cross. Reverse engineering and design recovery: A taxonomy. *IEEE Software*, 7(1):13–17, Jan 1990.

[29] C. Cifuentes and K. J. Gough. Decompilation of binary programs. *Softw., Pract. Exper.*, 25(7):811–829, 1995.

[30] A. Cimitile, A. De Lucia, G. A. Di Lucca, and A. R. Fasolino. Identifying objects in legacy systems using design metrics. *Journal of Systems and Software*, 44(3):199–211, 1999.

[31] M. L. Collard, H. H. Kagdi, and J. I. Maletic. An XML-based lightweight C++ fact extractor. In *11th International Workshop on Program Comprehension (IWPC 2003), May 10-11, 2003, Portland, Oregon, USA*, pages 134–143. IEEE Computer Society, 2003.

[32] J. R. Cordy. Comprehending reality - practical barriers to industrial adoption of software maintenance automation. In *11th International Workshop on Program Comprehension (IWPC 2003), May 10-11, 2003, Portland, Oregon, USA*, pages 196–206, 2003.

[33] J. R. Cordy, T. R. Dean, A. J. Malton, and K. A. Schneider. Source transformation in software engineering using the TXL transformation system. *Information & Software Technology*, 44(13):827–837, 2002.

[34] D. DaCosta, C. Dahn, S. Mancoridis, and V. Prevelakis. Characterizing the 'security vulnerability likelihood' of software functions. In *ICSM*, pages 266–275. IEEE Computer Society, 2003.

[35] A. De Lucia. Program slicing: Methods and applications. In *1st IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2001), 10 November 2001, Florence, Italy*, pages 144–151, 2001.

[36] A. De Lucia, M. Di Penta, R. Oliveto, and F. Zurolo. Improving comprehensibility of source code via traceability information: a controlled experiment. In *Proceedings of the 14th International Conference on Program Comprehension (ICPC 2006), June 2006, Athens, Greece*, pages 317–326, Los Alamitos, CA, USA, 2006. IEEE Computer Society.

[37] G. A. Di Lucca, A. R. Fasolino, and P. Tramontana. Reverse engineering Web applications: the WARE approach. *Journal of Software Maintenance*, 16(1-2):71–101, 2004.

[38] S. Ducasse and M. Lanza. The class blueprint: Visually supporting the understanding of classes. *IEEE Trans. Software Eng.*, 31(1):75–90, 2005.

[39] S. Ducasse, M. Lanza, and S. Tichelaar. Moose: an extensible language-independent environment for reengineering object-oriented systems. In *Proceedings of the Second International Symposium on Constructing Software Engineering Tools (CoSET 2000)*, 2000.

[40] S. G. Eick, J. L. Steffen, and E. E. S. Jr. Seesoft-a tool for visualizing line oriented software statistics. *IEEE Trans. Software Eng.*, 18(11):957–968, 1992.

[41] T. Eisenbarth, R. Koschke, and D. Simon. Locating features in source code. *IEEE Trans. Software Eng.*, 29(3):210–224, 2003.

[42] S. Elliott Sim, S. M. Easterbrook, and R. C. Holt. Using benchmarking to advance research: A challenge to software engineering. In *Proceedings of the 25th International Conference on Software Engineering (ICSE 2003), May 3-10, 2003, Portland, Oregon, USA*, pages 74–83, 2003.

[43] S. Elliott Sim, R. C. Holt, and S. M. Easterbrook. On using a benchmark to evaluate C++ extractors. In *Proceedings of the International Workshop on Program Comprehension*, pages 114–126. IEEE Computer Society, 2002.

[44] M. D. Ernst. Static and dynamic analysis: synergy and duality. In *ICSE Workshop on Dynamic Analysis (WODA)*, Portland, Oregon, USA, May 2003.

[45] R. Ferenc, Á. Beszédes, M. Tarkiainen, and T. Gyimóthy. Columbus - reverse engineering tool and schema for C++. In *Proceedings of the International Conference on Software Maintenance*, pages 172–181. IEEE Computer Society, 2002.

[46] R. Ferenc, S. E. Sim, R. C. Holt, R. Koschke, and T. Gyimóthy. Towards a standard schema for C/C++. In *Proceedings of the Working Conference on Reverse Engineering*, pages 49–58, 2001.

[47] R. Fiutem, P. Tonella, G. Antoniol, and E. Merlo. A cliché-based environment to support architectural reverse engineering. In *Proceedings of the International Conference on Software Maintenance*, pages 319–328. IEEE Computer Society, 1996.

[48] C. Fu, A. Milanova, B. G. Ryder, and D. Wonnacott. Robustness testing of Java server applications. *IEEE Trans. Software Eng.*, 31(4):292–311, 2005.

[49] R. Geiger, B. Fluri, H. C. Gall, and M. Pinzger. Relation of code clones and change couplings. In *Proceedings of FASE 2005*, number 3922 in Lecture Notes in Computer Science, pages 411–425, Vienna, Austria, March 2006. Springer.

[50] M. W. Godfrey and Q. Tu. Evolution in open source software: A case study. In *Proceedings of the International Conference on Software Maintenance*, San Jose California, October 2000. IEEE Press.

[51] N. Gold, C. Knight, A. Mohan, and M. Munro. Understanding service-oriented software. *IEEE Software*, 21(2):71–77, 2004.

[52] Grammatech Inc. The CodeSurfer slicing system, 2002.

[53] Y.-G. Guéhéneuc, H. A. Sahraoui, and F. Zaidi. Fingerprinting design patterns. In *11th Working Conference on Reverse Engineering (WCRE 2004), 8-12 November 2004, Delft, The Netherlands*, pages 172–181, 2004.

[54] T. Gyimóthy, R. Ferenc, and I. Siket. Empirical validation of object-oriented metrics on open source software for fault prediction. *IEEE Trans. Software Eng.*, 31(10):897–910, 2005.

[55] M. Harman. Software engineering optimization using search based techniques. In *ICSE - Future of SE Track*, 2007.

[56] A. Hassan and R. Holt. Architecture recovery of web applications. In *Proceedings of the International Conference on Software Engineering*, pages 349–359, Orlando, FL, USA, May 2002.

[57] A. E. Hassan, A. Mockus, R. C. Holt, and P. M. Johnson. Guest editor's introduction: Special issue on mining software repositories. *IEEE Trans. Software Eng.*, 31(6):426–428, 2005.

[58] D. Heuzeroth, T. Holl, G. Högström, and W. Löwe. Automatic design pattern detection. In *11th International Workshop on Program Comprehension (IWPC 2003), May 10-11, 2003, Portland, Oregon, USA*, pages 94–103, 2003.

[59] J. Huffman Hayes, A. Dekhtyar, and S. K. Sundaram. Advancing candidate link generation for requirements tracing: The study of methods. *IEEE Trans. Software Eng.*, 32(1):4–19, 2006.

[60] IEEE. *std 1219: Standard for Software maintenance*. IEEE Computer Society Press, Los Alamitos, CA, USA, 1998.

[61] G. Jayaraman, V. P. Ranganath, and J. Hatcliff. Kaveri: Delivering the Indus Java program slicer to Eclipse. In *Proceedings of FASE 2005,Edinburgh, UK, April 4-8, 2005*, pages 269–272, 2005.

[62] T. Kamiya, S. Kusumoto, and K. Inoue. CCFinder: A multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, 28(7):654–670, July 2002.

[63] C. Kapser and M. W. Godfrey. 'cloning considered harmful' considered harmful. In *Proceedings of the 2006 Working Conference on Reverse Engineering*, Benevento, Italy, October 2006.

[64] R. Kazman, S. S. Woods, and S. J. Carrière. Requirements for integrating software architecture and reengineering models: Corum II. In *Proceedings of the Working Conference on Reverse Engineering*, pages 154–163, 1998.

[65] J. O. Kephart and D. M. Chess. The vision of autonomic computing. *IEEE Computer*, 36(1):41–50, 2003.

[66] B. Korel. Black-box understanding of COTS components. In *Proceedings of the International Workshop on Program Comprehension*, pages 92–101, 1999.

[67] R. Koschke. *Atomic Architectural Component Recovery for Program Understanding and Evolution*. PhD thesis, Univ. of Stuttgart, Germany, 2000.

[68] C. Krämer and L. Prechelt. Design recovery by automated search for structural design patterns in object-oriented software. In *Proceedings of the Working Conference on Reverse Engineering*, pages 208–215, 1996.

[69] A. Kuhn, S. Ducasse, and T. Gîrba. Enriching reverse engineering with semantic clustering. In *12th Working Conference on Reverse Engineering (WCRE 2005), 7-11 November 2005, Pittsburgh, PA, USA*, pages 133–142, 2005.

[70] R. Lämmel and C. Verhoef. Cracking the 500-language problem. *IEEE Software*, 18(6):78–88, 2001.

[71] D. B. Lange and Y. Nakamura. Interactive visualization of design patterns can help in framework understanding. In *Proceedings of the Annual ACM SIGPLAN Conferences on Object-Oriented Programming, Systems, Languages, and Applications*, pages 342–357, 1995.

[72] M. Lanza and S. Ducasse. Polymetric views - a lightweight visual approach to reverse engineering. *IEEE Trans. Software Eng.*, 29(9):782–795, 2003.

[73] D. J. Lawrie, H. Feild, and D. Binkley. Leveraged Quality Assessment using Information Retrieval techniques. In *Proceedings of the 14th International Conference on Program Comprehension (ICPC 2006), June 2006, Athens, Greece*, pages 149–158, Los Alamitos, CA, USA, 2006. IEEE Computer Society.

[74] X. Llorà, K. Sastry, D. E. Goldberg, A. Gupta, and L. Lakshmi. Combating user fatigue in iGAs: partial ordering, support vector machines, and synthetic fitness. In *Genetic and Evolutionary Computation Conference, GECCO 2005, Proceedings, Washington DC, USA, June 25-29, 2005*, pages 1363–1370, 2005.

[75] J. I. Maletic, A. Marcus, and L. Feng. Source Viewer 3D (sv3D) - a framework for software visualization. In *Proceedings of the 25th International Conference on Software Engineering (ICSE 2003), May 3-10, 2003, Portland, Oregon, USA*, pages 812–813, 2003.

[76] A. Marcus and J. I. Maletic. Recovering documentation-to-source-code traceability links using Latent Semantic Indexing. In *Proceedings of the 25th International Conference on Software Engineering (ICSE 2003), May 3-10, 2003, Portland, Oregon, USA*, pages 125–137, 2003.

[77] A. Marcus and D. Poshyvanyk. The conceptual cohesion of classes. In *21st IEEE International Conference on Software Maintenance (ICSM 2005), 25-30 September 2005, Budapest, Hungary*, pages 133–142, 2005.

[78] A. Marcus, A. Sergeyev, V. Rajlich, and J. I. Maletic. An information retrieval approach to concept location in source code. In *Proceedings of the Working Conference on Reverse Engineering*, pages 214–223. IEEE Computer Society, 2004.

[79] M. Marin, A. van Deursen, and L. Moonen. Identifying aspects using fan-in analysis. In *Proceedings of the 11th Working Conference on Reverse Engineering (WCRE2004)*, pages 132–141. IEEE CS Press, 2004.

[80] L. Markosian, P. Newcomb, R. Brand, S. Burson, and T. Kitzmiller. Using an enabling technology to reengineer legacy systems. *Commun. ACM*, 37(5):58–70, 1994.

[81] J. Mayrand, C. Leblanc, and E. Merlo. Experiment on the automatic detection of function clones in a software system using metrics. In *Proceedings of the International Conference on Software Maintenance*, pages 244–253, Monterey, CA, Nov 1996.

[82] A. M. Memon, I. Banerjee, and A. Nagarajan. GUI ripping: Reverse engineering of graphical user interfaces for testing. In *10th Working Conference on Reverse Engineering (WCRE 2003), 13-16 November 2003, Victoria, Canada*, pages 260–269, 2003.

[83] A. M. Memon and M. L. Soffa. Regression testing of GUIs. In *ESEC / SIGSOFT FSE*, pages 118–127, 2003.

[84] E. Merlo, P.-Y. Gagné, J.-F. Girard, K. Kontogiannis, L. J. Hendren, P. Panangaden, and R. de Mori. Reengineering user interfaces. *IEEE Software*, 12(1):64–73, 1995.

[85] A. Milanova, A. Rountev, and B. G. Ryder. Parameterized object sensitivity for points-to analysis for java. *ACM Trans. Softw. Eng. Methodol.*, 14(1):1–41, 2005.

[86] B. S. Mitchell and S. Mancoridis. On the automatic modularization of software systems using the Bunch Tool. *IEEE Trans. Software Eng.*, 32(3):193–208, 2006.

[87] D. L. Moise, K. Wong, H. J. Hoover, and D. Hou. Reverse engineering scripting language extensions. In *Proceedings of the 14th International Conference on Program Comprehension (ICPC 2006), June 2006, Athens, Greece*, pages 295–306, Los Alamitos, CA, USA, 2006. IEEE Computer Society.

[88] L. Moonen. Generating robust parsers using island grammars. In *Proceedings of the Working Conference on Reverse Engineering*, pages 13–22, 2001.

[89] M. Moore. *User Interface Reengineering*. PhD thesis, Georgia Institute of Technology, USA, 1998.

[90] H. A. Müller. Bits of history, challenges for the future and autonomic computing technology. In *13th Working Conference on Reverse Engineering (WCRE 2006), 23-27 October 2006, Benevento, Italy*, pages 9–18, 2006.

[91] H. A. Müller, J. H. Jahnke, D. B. Smith, M.-A. D. Storey, S. R. Tilley, and K. Wong. Reverse engineering: a roadmap. In *ICSE - Future of SE Track*, pages 47–60, 2000.

[92] M. Paolucci, T. Kawamura, T. R. Payne, and K. Sycara. Semantic matching of Web services capabilities. In *Proceedings of the first International Semantic Web Conference (ISWC 2002)*, volume 2348 of *Lecture Notes on Computer Science*, pages 333–347. Springer-Verlag, June 2002.

[93] D. Poshyvanyk, A. Marcus, V. Rajlich, Y.-G. Guéheneuc, and G. Antoniol. Combining Probabilistic Ranking and Latent Semantic Indexing for feature identification. In *Proceedings of the 14th International Conference on Program Comprehension (ICPC 2006), June 2006, Athens, Greece*, pages 137–148, Los Alamitos, CA, USA, 2006. IEEE Computer Society.

[94] W. J. Premerlani and M. R. Blaha. An approach for reverse engineering of relational databases. *Commun. ACM*, 37(5):42–49, 134, 1994.

[95] F. Ricca and P. Tonella. Analysis and testing of web applications. In *Proceedings of the International Conference on Software Engineering*, pages 25–34, Toronto, ON, Canada, May 2001. IEEE Computer Society Press, Los Alamitos, CA, USA.

[96] D. Saff and M. D. Ernst. Reducing wasted development time via continuous testing. In *14th International Symposium on Software Reliability Engineering (ISSRE 2003), 17-20 November 2003, Denver, CO, USA*, pages 281–292, 2003.

[97] K. Sartipi. *Software Architecture Recovery based-on Pattern Matching*. PhD thesis, Univ. of Waterloo, Canada, 2003.

[98] M. Siff and T. W. Reps. Identifying modules via concept analysis. *IEEE Trans. Software Eng.*, 25(6):749–768, 1999.

[99] H. M. Sneed. Encapsulation of legacy software: A technique for reusing legacy software components. *Ann. Software Eng.*, 9:293–313, 2000.

[100] M.-A. D. Storey. Theories, methods and tools in program comprehension: Past, present and future. In *13th International Workshop on Program Comprehension (IWPC 2005), 15-16 May 2005, St. Louis, MO, USA*, pages 181–191, 2005.

[101] M.-A. D. Storey, F. D. Fracchia, and H. A. Müller. Cognitive design elements to support the construction of a mental model during software exploration. *Journal of Systems and Software*, 44(3):171–185, 1999.

[102] M.-A. D. Storey and H. A. Müller. Manipulating and documenting software structures using shrimp views. In *ICSM*, pages 275–284. IEEE Computer Society, 1995.

[103] D. Strein, H. Kratz, and W. Löwe. Cross-language program analysis and refactoring. In *Proceedings of the International Workshop on Source Code Analysis and Manipulation*, pages 207–216. IEEE Computer Society, 2006.

[104] T. Systä. *Static and Dynamic Reverse Engineering Techniques for Java Software Systems*. PhD thesis, Univ. of Tampere, Finland, 2000.

[105] S. R. Tilley and S. Huang. Evaluating the reverse engineering capabilities of Web tools for understanding site content and structure: A case study. In *Proceedings of the 23rd International Conference on Software Engineering, ICSE 2001, 12-19 May 2001, Toronto, Ontario, Canada*, pages 514–523, 2001.

[106] P. Tonella and M. Ceccato. Aspect mining through the formal concept analysis of execution traces. In *Proceedings of the Working Conference on Reverse Engineering*, pages 112–121, 2004.

[107] P. Tonella and A. Potrich. *Reverse Engineering of Object Oriented Code*. Springer-Verlag, Berlin, Heidelberg, New York, 2005.

[108] N. Tsantalis, A. Chatzigeorgiou, G. Stephanides, and S. T. Halkidis. Design pattern detection using similarity scoring. *IEEE Transactions on Software Engineering*, 32(11):896–909, 2006.

[109] M. Turner, D. Budgen, and P. Brereton. Turning software into a service. *IEEE Computer*, 36(10):38–44, 2003.

[110] V. Tzerpos and R. C. Holt. Software botryology: Automatic clustering of software systems. In *DEXA Workshop*, pages 811–818. IEEE Computer Society Press, Los Alamitos, CA, USA, 1998.

[111] E. van Emden and L. Moonen. Java quality assurance by detecting code smells. In *9th Working Conference on Reverse Engineering (WCRE 2002), 28 October - 1 November 2002, Richmond, VA, USA*, pages 97–107, 2002.

[112] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, July 1984.

[113] N. Wilde and M. C. Scully. Software reconnaissance: mapping program features to code. *Journal of Software Maintenance*, 7(1):49–62, 1995.

[114] L. M. Wills. *Automated program recognition by graph Parsing*. PhD thesis, 1992.

[115] A. Winter, B. Kullbach, and V. Riediger. An overview of the GXL graph exchange language. In *Proc. of the Software Visualisation International Seminar, LNCS 2269*, pages 324–336, Dagstuhl Castle, Germany, May 2002. Springer-Verlag.

[116] K. Wong, S. Tilley, H. A. Muller, and M. D. Storey. Structural redocumentation: A case study. *IEEE Software*, pages 46–54, Jan 1995.

[117] Y. Yu, J. Mylopoulos, Y. Wang, S. Liaskos, A. Lapouchnian, Y. Zou, M. Littou, and J. C. S. P. Leite. RETR: reverse engineering to requirements. In *12th Working Conference on Reverse Engineering (WCRE 2005), 7-11 November 2005, Pittsburgh, PA, USA*, page 234, 2005.