DOI:10.1145/1924421.1924451

Deeply understanding the intricacies of software must always come before any considerations for modifying it.

BY GERARDO CANFORA, MASSIMILIANO DI PENTA, AND LUIGI CERULO

Achievements and Challenges in Software Reverse Engineering

THE NEED FOR changing existing software has been with us since the first programs were written. After the pioneering work of Lehman[30] we know that real-world software systems require continuous change and enhancement to satisfy new user requirements and expectations, to adapt to new and emerging business models and organizations, to adhere to changing legislation, to cope with technology innovation, and to preserve the system structure from deterioration. A major proportion of software life cycle costs is due to maintenance and support; while figures vary, there is a general agreement in the software industry that the proportion is well over 50% of software costs.

The need for modifying software induces the need to comprehend it. Software comprehension is a human-intensive process, where developers acquire sufficient knowledge about a software artifact, or an entire system, so as to be able to successfully accomplish a given task, by means of analysis, guessing, and formulation of hypotheses. In most cases, software comprehension is challenged by the lack of adequate and up-to-date software documentation, often due to limited skills or to resource and timing constraints during development activities.

Software reverse engineering is a broad term that encompasses an array of methods and tools to derive information and knowledge from existing software artifacts and leverage it into software engineering processes. In a seminal paper, Chikofsky and Cross[12] define software reverse engineering as "*the process of analyzing a subject system to identify the system's components and their inter-relationships and create representations of the system in another form or at a higher level of abstraction.*" In accordance with this definition, reverse engineering is a process of examination rather than a process of change,

» key insights

- **When maintaining a software system, often the only reliable information is embedded in its source code. Reverse engineering aims at creating high-level representations for an existing software system to support comprehension and evolution.**
- **Understanding software consumes a substantial portion of the maintenance effort; nevertheless, such a task is often performed with simple general-purpose tools, such as editors and regular-expression matchers.**
- **Available reverse engineering tools help to: extract facts from source code/ binaries, execution traces, or historical data; query the extracted facts; and build high-level views for humans.**

ILLUSTRATION BY ANDREAS KÖBERLE

as the core of reverse engineering is deriving, from available software artifacts, representations understandable by humans.

Software reverse engineering originated in software maintenance: the standard IEEE-1219[a] recommends reverse engineering as a key supporting technology to deal with systems that have the source code as the only reliable representation. Since then, it has been successfully exploited to deal with numerous software engineering problems. A non-exhaustive list includes: recovering architectures and design patterns, re-documenting programs and database, identifying reusable assets, building traceability between software artifacts, computing change impacts, re-modularizing existing systems, renewing user interfaces, migrating toward new architectures and platforms.[9]

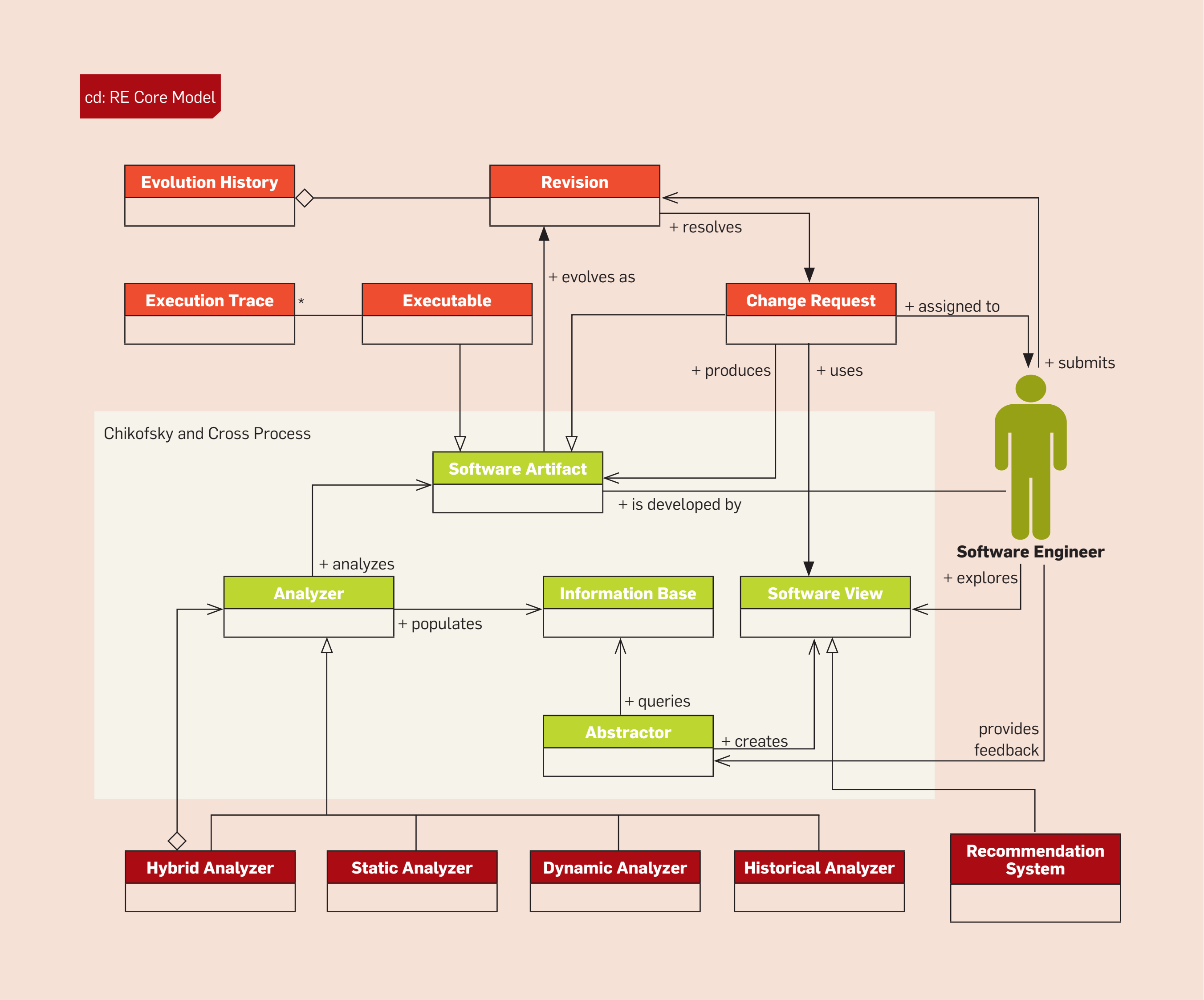
To support such a wide range of applications, many different types of tools have been developed either in academia or industry, including pretty printers, static and dynamic code analyzers, and tools for code visualization and exploration, design recovery, documents and diagrams generation, and, more recently, for mining software repositories.

During the last decade, the Y2K and the Euro problems focused the world's attention on system evolution issues, demonstrating the maturity of reverse engineering as an effective support to face difficult and critical problems. In such context, reverse engineering was used, above all, to locate features that needed modifications, and to evaluate the impact of such modifications. Success stories of reverse engineering can be seen every day, when organizations:

- Keep their development costs under control by integrating commercial-off-the-shelf;
- Components into existing software systems;
- Gain competitive advantages by

[a] The IEEE SA-1219-1988 IEEE Standard for Software Maintenance.

Reverse engineering concepts.



moving existing systems to the Web;

- Maintain software systems in line with the business evolution by preparing existing systems for fruition as a set of services within a service oriented infrastructure;
- Cope with the growing complexity of today's systems-of-systems by injecting adaptation and self-management capabilities into existing systems; and
- Maintain software investments in line with the organization strategies by thorough software portfolio management.

In this article we discuss—with no ambition to provide an exhaustive survey—developments of broad significance in the field of reverse engineering, and highlight unresolved questions and future directions. We summarize the most important reverse engineering concepts and describe the main challenges and achievements. We also provide references to relevant and available tools for the two major activities of a reverse engineering process,[12] for example, performing software analysis and building software views, respectively. Finally, we outline future trends of reverse engineering in today's' software engineering landscape.

Reverse Engineering Concepts

The accompanying figure shows a conceptual model for software reverse engineering as a UML class diagram. This model stems from the Chikofsky and Cross view of the reverse engineering process, highlighted in the gray area in the figure. As described in the model, a reverse engineering activity is performed by a *software engineer* to solve some specific problems related to a software product, consisting of *software artifacts* (for example, source code, executables, project documentation, test cases, and change requests). *Software engineers* benefit of reverse engineering by exploring software artifacts by means of *software views*, representations, sometimes visual, aimed at increasing the current comprehension of a software product, or at favoring maintenance and evolution activities. *Software views* are built by means of an *abstractor*, which in turn uses information extracted from *software artifacts* and stored in the *information base* by an *analyzer*. Often reverse engineering aims at analyzing the evolution of software artifacts across their *revisions*, which occur to satisfy *change requests*.

The *analyzers* can extract information by means of static analysis (*static analyzers*), dynamic analysis (*Dynamic Analyzers*) or a combination of the two (*hybrid analyzers*). Recently, *historical analyzers*, which extract information from the evolution repository of a software product, are gaining a growing popularity.

The *software engineer* can provide feedbacks to the reverse engineering tool with the aim of producing refined and more precise views. A particular type of *software view* that emerged recently are *recommendation systems*,[39] which provide suggestions to the software engineer and, if needed, trigger a new *change request*.

Software Analysis

Software analysis is performed by *analyzers*—tools that take software artifacts as input and extract information relevant to reverse engineering tasks. Software analysis can be: *static*, when it is performed, within a single system snapshot, on software artifacts without requiring their execution; *dynamic*, when it is performed by analyzing execution traces obtained from the execution of instrumented versions of a program, or by using an execution environment able to capture facts from program executions; and *historical*, when the aim is to gain information about the evolution of the system under analysis by considering the changes performed by developers to software artifacts, as recorded by versioning systems. David Binkley[6] has written a thorough survey on source code analysis.

Static analyzers must deal with a number of challenges:

- *Ability to parse different language variants and non-compilable code.* The diffusion of a wide number of programming languages dialects and implicit characteristics of programming languages, for example the presence of preprocessor directives and macros, makes the life of reverse engineers not easy. In some cases it is necessary to analyze partial fragments of source code violating the assumption of parsers: this might be the case, for example, of a snapshot downloaded from a versioning system.

There are tools such as the *Design Maintenance Systems* (*DMS*)[3] that perform a complete parsing of the source code, also contemplating the presence of preprocessor directives where these are foreseen by the programming language. Whenever we do not need a complete parsing, but rather to extract some specific information, we can use island and lake parsers.[35] Island parsers only parse source code fragments of interest, ignoring any other token and activating the parser only when tokens of interest are encountered. Lake parsers, instead, are able to ignore source code fragments not contemplated by the grammar, such as, related to programming language dialects. Tools such as *TXL*[15] or *SrcML*[14] allow for a robust source code island parsing. Whenever the source code is not available, decompilation tools are needed to perform the analysis.[13]

Software comprehension is a human-intensive process, where developers acquire sufficient knowledge about a software artifact or system to successfully accomplish a given task, by means of analysis, guessing, and formulation of hypotheses.

- *Ability to extract facts useful for reverse engineering tasks.* Sometimes we do not need to build a complete program Abstract Syntax Tree (AST), but rather to populate the information base with information relevant for a particular reverse engineering task.

Fact extractors such as the *Bauhaus* fact extractor[26] or *Columbus*[19] extract facts—for example, dependencies, metrics—from the source code and store them in an information base. Approaches such as the one by Kuhn et al.[28] consider the source code as text, and extract fact from it by indexing the terms it contains.

- *Ability to extract a program's semantics.* This entails the ability of analyzers to go beyond the simple AST extraction, by performing a set of analyses that are indispensable for many reverse engineering tasks.

Tools such as *CodeSurfer*[1] allow to perform a series of analyses related to a program's semantics, such as control dependence analysis, data flow analysis, or pointer analysis, and then build views on top of these analyses, for example, aimed at represent a program slice or to highlight the impact of a change.

Static analysis is reasonably fast, precise, and cheap. However, many peculiarities of programming languages, such as pointers and polymorphism, or dynamic class loading, make static analysis difficult and sometimes imprecise. In addition, the extraction of some specific information related, for instance, to the interaction of a user with the application or to the sequence of messages exchanged between objects, is difficult, if not infeasible, to perform statically.

To overcome the limitations of static analysis, reverse engineers can count on dynamic analysis, which extracts information from execution traces. For many reverse engineering tasks, it is beneficial to combine static and dynamic analysis, as static analysis can be imprecise, for example, not able to fully deal with aspects such as pointer resolution, while dynamic analysis can be incomplete, because it depends on program inputs. An extensive survey of dynamic analysis approaches for the specific purpose of program comprehension can be found in Cornelissen et al.[16]

Key challenges of dynamic analysis include:

- *Ability to compile and execute the program*, which might appear as obvious, however it is not always true if we want to analyze an intermediate version of a program. Again, this can happen when analyzing an incomplete snapshot grabbed from a versioning system.
- *Ability to choose representative inputs*, as the results of dynamic analysis strongly depend on the inputs used to execute the subject system.

For some reverse engineering tasks, it may be necessary to perform a thorough coverage of the application code or functionality, while in other cases it would be more appropriate to execute the program under a realistic execution scenarios, where the most frequently used features are exercised more than others.

- *Ability to mine relevant information from execution traces.* Execution traces tend to quickly become large and unmanageable, thus a relevant challenge is to filter them and extract information relevant for the particular understanding task being performed.

Approaches such as the one proposed by Hassan et al.[23] aim at filtering the execution traces and mining recurring patterns to analyze the application's operational profiles, and tools like Daikon[18] identify likely invariants by analyzing execution traces.

- *Ability to perform a black-box analysis*: for some kinds of systems, for example, service-oriented systems, the reverse engineer cannot access the source code, but just execute the deployed application.

Software analysis is performed by *analyzers*—tools that take software artifacts as input and extract information relevant to reverse engineering tasks.

Some approaches[31] reverse engineer properties of a deployed service relying only on the observed outputs, without the need for source code analysis.

The growing diffusion of versioning systems, bug tracking systems, and other software repositories, such as mailing lists or security advisories, poses the basis for a third dimension of software analysis, namely the *historical analysis* of data extracted from software repositories. Historical analysis is complementary to static and dynamic analysis, in that it allows us to understand *how* an artifact was modified over time, *when* it was changed, *why* it was changed, *who* changed it, and *what artifacts* changed together. (For a thorough survey on historical analysis see Kagdi et al.[24]) While mining software repositories opens challenging research directions, such sources of information must be used with particular care. Among others, in a recent roundtable of mining software repositories experts,[21] Notkin warned against the flourishing of mining software repository studies able to find relationships in and among repositories, that, although significant, are not causal, or cannot lead to improved approaches and tools to develop better software systems. Also, Mockus pointed out that information about software projects cannot be fully observed from versioning systems and problem reporting systems only.

Historical analysis poses a series of challenges related to:

- *Ability to classify and combine information from different, heterogeneous software repositories.* For example, integrating different repositories requires linking a change committed in a versioning system with an issue posted on a bug-tracking system.

A widely used heuristic aims at matching the bug tracking IDs to versioning system commit notes. Another problem historical analysis must deal with is the accuracy of the available information, for example, of the way issues posted on bug tracking systems are classified and prioritized.

- *Ability to analyze differences.* The "core" of historical analysis is to understand changes occurred in software artifacts, usually stored in versioning systems such as the Concurrent Versions

System[b] (CVS) and SubVersioN[c] (SVN).

There are approaches working on a flat representation of the program, thus ensuring language independence, and other, language-dependent, that work on ASTs. In the first category fall the Unix *diff* and *ldiff*,[11] which overcomes *diff* limitations in distinguishing likely line changes from additions and removals, and of tracking line moving. In the second category falls *ChangeDistiller*,[20] which is able to provide detailed information about the changes occurred, for example, a change to a method parameter or to control-flow construct.

▸ *Ability to group together related changes.* When performing historical analysis, it would be useful to consider changes related to the same development/maintenance activity as a whole. Grouping together related changes is particularly crucial when CVS is used, as it treats commits of different files separately, although for SVN—which intrinsically treats a commit of multiple files as a whole—it might be useful to group together related commits. Existing approaches group sequences of file revisions that share the same author, branch, and commit notes, and exhibit differences between timestamps below a certain threshold (often 200 seconds).

Table 1 provides a categorization of some relevant examples of static, dynamic, historical, and hybrid analyzers.

Building Software Views

A primary goal of reverse engineering is to produce software views by abstracting facts stored in the knowledge base. During the process of building views, there is a series of challenges that reverse engineers must face off:

▸ *Ability to query the information base containing facts populated by analyzers.* This entails the need for powerful query languages that allow building informative views from the data stored in the Information Base.

Examples of querying mechanism include: Grok, a notation for manipulating binary relations using the Tarski's relational algebra, and comprising operators for manipulating sets or relations, or *CrocoPat*,[5] which uses relational calculus to query the information base. If the information base is populated with source code ASTs, it can be queried by means of pattern matching mechanisms working over ASTs, as it happens for tools such as *DMS* and *TXL*, or even using XQuery when AST is represented in XML, as it happens for *SrcML*.

▸ *Ability to abstract low-level artifacts and reconstruct high-level information.* Very often the only reliable source of documentation for a software system is its source code. To comprehend a software system, higher-level documentation is needed, such as the system architecture, the system design, or traceability links between different software artifacts. During the years, approaches have been successful developed to build views at different levels of abstraction, namely:

▹ *Code views*: traditional code views provided by source code browsers need to be augmented with other information aimed at better supporting the tasks a software developer performs, such as, debugging, determining the impact of a change, or refactoring.

There are tools aimed at computing and visualizing slices or data-flow information,[1] extracting source code

Table 1. Software analyzers.

Name	Availability	Description
Static Analysis		
SrcML[14]	Free	Robust, island-grammar based parser. Produces a XML-based AST. URL: http://www.sdml.info/projects/srcml/
DMS®[3]	Commercial	General purpose software analysis and transformation toolkit. URL: http://www.semdesigns.com/Products/
TXL[15]	Free	Program analysis and transformation toolkit. It is based on a hybrid functional/rule-based language for program transformation. URL: http://www.txl.ca/
CodeSurfer[1]	Commercial, free for academic use	Supports a wide series of features such as data flow analysis, points-to analysis, slicing, impact analysis, dependence analysis. URL: http://www.grammatech.com/products/codesurfer/
Bauhaus fact extractor[26]	Commercial, free for academic use	Front end of the Bauhaus reverse engineering framework. The analyzer populates a knowledge base composed of the InterMediate Language (IML) representations, containing at the syntactical and semantic levels, and Resource Flow graphs (RFG) representing architectural aspects. URL: http://www.bauhaus-stuttgart.de/
Columbus/CAN[19]	Commercial, free for academic use	Robust and error-tolerant parser front end for C and C++. URL: http://www.frontendart.com/
Dynamic Analysis		
Daikon[18]	Free	Likely invariant detection tool for languages such as C, C++, Eiffel, IOA, Java, and Perl. Includes a code instrumentor. URL: http://groups.csail.mit.edu/pag/daikon/
Historical Analysis		
Ldiff[11]	Free	Line-based differencing tool, overcomes the Unix diff limitations in detecting changes. URL: http://www.rcost.unisannio.it/cerulo/tools.html
Evolizer[20]	Free (Apache 2.0 license)	Platform-realized as set of Eclipse plug ins-to analyze software project data. Includes Change Distiller, an AST-based differencing tool. URL: https://www.evolizer.org/
Hybrid Analysis		
Moose[37]	Free	Software analysis tool, relies on external fact extractors, and stores facts in different knowledge bases (e.g., FAMIX). Several tools for building views (e.g., CodeCrawler) are built on top of Moose. URL: http://moose.unibe.ch/

[b] http://ximbiot.com/cvs/

[c] http://subversion.tigris.org/

clones and visualizing them,[25] or at identifying the presence of crosscutting concerns.[33] A different case of code view is a decompilation,[13] where the objective is not to augment the source code with further information, but rather to abstract the (unavailable) source code from binaries.

▹ *Design views*: as very often design documents are not available, incomplete or outdated, design views abstract design information by relying on facts extracted from the source code. The literature reports several different approaches aimed at reconstructing software architectures;[38] among other techniques, clustering techniques[32] are used to group together modules with a particular objective in mind, for example, maximizing cohesion and minimizing coupling. Other approaches aim at reconstructing the software design—for example, UML diagrams using static analysis,[42] or a combination of static and dynamic analysis[40]—and at identifying the usage of design patterns in the software system design,[43] and, in general, at re-documenting the software structure.[45]

Table 2. Tool support for building software views.

Name	Availability	Description
Information Base Querying Tools		
Grok	Free	Programming language for manipulating binary relations. Part of the SWAG toolkit developed at the University of Waterloo. URL: http://www.swag.uwaterloo.ca/tools.html
Crocopat[5]	Free	Relational programming tool able to manipulate relations of any arity. URL: http://www.sosy-lab.org/~dbeyer/CrocoPat/
View Builders		
Bauhaus[26]	Commercial, free for academic use	Complete reverse engineering toolkit. Foresees the production of software views at different levels of details, code views (pointer analysis, dependency analysis, clone detection, dead code detection), feature location, component mining, architecture reconstruction. URL: http://www.bauhaus-stuttgart.de/
Rigi[45]	Free	Extensible reverse engineering tool for software understanding and redocumentation. Shows views at different levels of detail, from software architecture to control and data flow relations. Also includes a fact extractor. URL: http://www.rigi.csc.uvic.ca/
CCFinderX[25]	Commercial, free for academic use	Token-based clone detection tool. Works with languages such as Java, C/C++, COBOL, VB, C#. http://www.ccfinder.net/
FINT[33]	Free	Aspect mining Eclipse plugin. Detects crosscutting concerns using a combination of different techniques (fan-in analysis, grouped calls analysis and redirection finder). URL: http://swerl.tudelft.nl/bin/view/AMR/FINT
Design pattern detector[43]	Free	Graph-similarity-based design pattern detection tool. Analyzes Java bytecode. URL: http://java.uom.gr/~nikos/pattern-detection.html
Software Visualization Tools		
Codecrawler[29]	Free	Reverse engineering visualization toolkit. In particular visualizes polymetric views. URL: http://smallwiki.unibe.ch/codecrawler/
SHriMP[41]	Free	Tool (and technique) to explore software architectures and information spaces in general. Built as Protegé plugin. URL: http://www.thechiselgroup.org/shrimp/
Recommendation Systems		
HipiKat	Free	Recommends pertinent software development artifacts related to the context where the developer is operating. URL: http://www.cs.ubc.ca/labs/spl/projects/hipikat/
CloneTracker[17]	Free	Eclipse plugin for clone tracking. Developers tag clones; when clones are changed, developers are notified and supported to make consistent changes. URL: http://www.cs.mcgill.ca/~swevo/clonetracker/

▹ *Symbolic views*, highlights "what software artifacts are about." Such views can be used for program understanding, but also for other tasks such impact analysis or for recommending relevant developers for a given artifact. An example of this kind of view is the work of Kuhn et al.,[28] who build views highlighting the semantic of software packages by using an information retrieval technique—namely latent semantic indexing—to group software artifacts.

▹ *Traceability links among different views*: traceability links are crucial to support software maintenance tasks, however they may be outdated or not present at all.

Techniques using information retrieval methods have been developed to recover traceability links, hypothesizing the consistency between high-level document terms and source code identifiers and comments.[2]

▸ *Ability to properly visualize abstracted information.* Software visualization is a crucial step for reverse engineering: the way information is presented to developers could strongly affect the usefulness of the program analysis or design recovery techniques. In some cases, when reverse engineering techniques aim at extracting widely adopted and well defined diagrams from source code—for example, UML diagrams and control flow graphs—the choice of a proper visualization is quite straightforward. In other cases, visualization constitutes the essence of a reverse engineering technique, for its ability to highlight relevant information at the right level of detail. Relevant examples of visualization tools include SHriMP,[41] which allows exploring software architectures, or other pieces of knowledge related to a software system. Other visualizations have been conceived to provide developers with immediate insights about the metric profile of an entire system and zooming into detailed information about particular artifacts. For example, *CodeCrawler*[29] combines the capability of showing software entities and their relationships, with the capability of visualizing software metrics using *polymetric views*, which represent software artifacts as boxes and show different metrics using the width, the length and the color of these boxes. (An extensive survey on software visualization approaches and tools is provided by Rainer Koschke.[27])

Software visualizations are not necessarily related to a single snapshot of a software system. Historical analysis allows for building *historical views*; as an example, the *animated storyboards* proposed by Beyer and Hassan[4] use a sequence of animated panels to visualize the changes occurring in a software system.

Some relevant examples of tools supporting the creation of reverse engineering views are described in Table 2.

Future and Emerging Trends

Reverse engineering has reached great maturity, as we have examined. However, there are still a number of open problems that call for additional research to advance and improve current methods and tools, and to consider new software development scenarios where reverse engineering could play a crucial role. Moreover, radical innovations are needed to cope with new and emerging software development scenarios and new system architectures.

Current approaches to build views suffer two main limitations:

1. They are either incomplete or imprecise, especially when they are completely automatic; above all, they do not account for experts' knowledge;
2. They are semi-automatic, for example, inputs from human experts are required to complement or correct the information automatically extracted and to produce useful views for the developer.

These limitations indicate the need to close the loop between tool support and developers. Future research 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 better results.

Reverse engineering has been traditionally intended as a support for post-delivery activities. There is a growing need for full integration of reverse engineering within the development process, so that developers can benefit from on-the-fly application of reverse engineering techniques, for example, while they are writing code or a design model. Muller et al.[34] highlighted the idea of exploiting information extracted by reverse engineering as a feedback on the forward development process. This idea of continuous reverse engineering has several expected benefits:

▸ By extracting and continuously updating high-level views (such as, architectural views, or design diagrams such as class diagrams, sequence diagrams, state machines) from the source code under development, it would be possible to provide a clearer picture of the system being created;

▸ 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;

▸ A continuous analysis of the code under development and of other artifacts can be used to provide developers with useful insights, such as, suggesting the use of a particular component or revealing that duplication of code is being created; and,

▸ Feedback from developers could be automatically grabbed from the development environment and used to improve the performance of reverse engineering techniques. For example, the Rocchio Vector Space Model relevance feedback mechanism is used to improve the performances of traceability recovery approaches.

In recent years, the availability of multiple sources of data (software repositories) demands for techniques to combine them and for the need of filtering such a huge amount of information, that would otherwise cause an overload for developers.[36] Recommender systems are an emerging response to this kind of issues favored by the availability of highly customizable development environments, such as *Eclipse*[d] and *NetBeans*[e] providing ways to quickly develop new tools completely integrated in the environment the developer is using. For example,

A primary goal of reverse engineering is to produce software views by abstracting facts stored in the knowledge base.

[d] http://www.eclipse.org/
[e] http://www.netbeans.org/

CloneTracker[17] keep tracks of changes occurring on source code clones. Recommender systems integrate many reverse engineering techniques and their growing popularity suggests that next generation software development environments will most probably leverage reverse engineering into the standard outfit of software developers.

In the view of analyzing developers' work on software projects, an important challenge is to relate information about software artifacts, extracted by means of reverse engineering techniques, with information about communication/cooperation among developers or in general among project contributors. In recent years, the combination of program analysis and mining software repositories with social network analysis techniques is becoming an effective tool that helps to understand the relationships existing between developers' social networking the characteristics of the code they tend to modify.[7]

Today, many systems are multi-languages and cross-platforms, requiring reverse engineering tools able to deal with multiple languages and multiple platforms within a single conceptual framework. Finally, new kinds of software artifacts, that need to be reverse engineered, are emerging, including spreadsheets and macros deriving from the growing phenomenon of end-user programming.[8]

The advent of service oriented architectures (SOA) poses new challenges to reverse engineering. A service oriented system is composed of distributed services published by different providers and poses relevant software understanding issues.[22] In fact, each service offers, through its interface, a limited view of the features because providers "sell" the usage of a service but want to protect the technology behind it. This affects the service understandability and, being the implementation not available for reverse engineering, black box understanding techniques must be used. Upcoming work in reverse engineering can also support service providers to (semi) automatically produce service annotations—inferred using black box reverse engineering techniques—for the purpose of automatic discovery and reconfiguration.[31] From the service provider side, service annotations can also be produced using source code reverse engineering techniques, aimed at extracting models to be used for annotating the service. Of course, this kind of automatically extracted information can be different from what today is assumed by the already developed automatic service discovery and composition mechanisms, necessitating a step back and a rethinking for some of them.

Radical innovations are needed to cope with new and emerging software development scenarios and new system architectures.

Many organizations are currently using SOA as an integration platform for their systems. Thus, within an enterprise a key challenge is often to turn monolithic systems into service oriented architectures, so that they are better aligned with the business processes. Reverse engineering is a valuable option to help re-architecting these intra-enterprise applications.[10] Recasting existing systems into services is also a prerequisite to move towards cloud computing, a style of computing where not only software but also the infrastructure and the platform are seen as services.

Reverse engineering has traditionally focused on recovering functional and architectural views from existing software artifacts. However, reverse engineering approaches have also shown useful to recover performance models and workload models of an application, manly by analyzing execution traces.[23,44] These models have proven particularly useful to deal with applications running on limited-resources devices, including embedded systems and mobile and pervasive applications, and to help migrating existing applications onto these devices. Example of key problems are adapting the interface to the limited size and resolution of these devices and miniaturizing libraries to accommodate for reduced size of memory.

With the increasing diffusion of handled-devices and wireless sensor networks, power management is becoming a key issue. Consequently, an emerging goal of reverse engineering is to analyze an application and identify changes aimed at reducing power consumption. This is crucial for mobile devices such as smart phones and Personal Digital Assistants (PDAs), but, especially, in the context of sensor networks, where there is the need for increasing battery duration for geographically distributed sensors. In this context, reverse engineering has the role of understanding application power bottlenecks, which may be an excessive usage of the display or of a wireless connection, but also to the use of CPU instructions having high power requirements. Last but not least, in a long-term vision, "green computing" envisages scenarios where general-purpose computing makes a reduced utilization of power resources. This is, indeed, a software engineering problem and, for existing systems, a reverse engineering problem.

Conclusion

Reverse engineering is, at the same time, an old activity and young discipline. In fact, developers have always struggled about analyzing software components to gather information that the documentation leaves out, examining source code to reconstruct the underlying rationale and design choices, and inspecting data formats to maintain communications among applications. However, for a long time these and other reverse engineering tasks have been carried-on using ad-hoc approaches, with the help of very simple general-purposes tools, such as editors and regular-expression matchers.

Only in the past two decades has software engineering recognized the need for systematic approaches, and supporting tools, to gather information from existing software and leveraging it into engineering processes. Nevertheless, pushing the adoption of reverse engineering techniques in the development practice is still a major need, requiring appropriate education of developers both in university courses and within industry, and to support reverse engineering techniques and tools with empirical evidence about their performance and usability and with guidelines for their adoption.

References

1. Anderson, P. and Teitelbaum, T. Software inspection using CodeSurfer. In *Proc. of the 1st Workshop on Inspection in Software Engineering*, 2001, 4–11.
2. Antoniol, G., Canfora, G., Casazza, G., De Lucia, A., and Merlo, E. Recovering traceability links between code and documentation. *IEEE Transactions on Software Engineering 28*, 10 (2002), 970–983.
3. Baxter, I.D., Pidgeon, C., and Mehlich, M. DMS: Program transformations for practical scalable software evolution. In *Proc. of the 26th International Conference on Software Engineering*, 2004, 625–634.
4. Beyer, D. and Hassan, A.E. Animated visualization of software history using evolution storyboards. In *Proc. of the 13th Working Conference on Reverse Engineering*, 2006. IEEE CS Press, 199–210.
5. Beyer, D. and Lewerentz, C. CrocoPat: Efficient pattern analysis in object-oriented programs. In *Proc. of the 11th International Workshop on Program Comprehension*, 2003. IEEE CS Press, 294–295.
6. Binkley, D. Source code analysis: A road map. In *Proc. of the International Conference on Software Engineering — Future of Software Engineering Track* (FOSE). IEEE CS Press, 104–119.
7. Bird, C., Pattison, D., D'Souza, R., Filkov, V., and Devanbu, P. Latent social structure in open source projects. In *Proc. of the 16th ACM SIGSOFT international Symposium on Foundations of Software Engineering*, 2008. ACM Press, 24–35.
8. Burnett, M., Cook, C., and Rothermel, G. End-user software engineering. *Comm. ACM 47*, 9 (Sept. 2004), 53–58.
9. Canfora, G. and Di Penta, M. New frontiers of reverse engineering. In *Proc. of the International Conference on Software Engineering - Future of Software Engineering Track* (FOSE), 2007. IEEE CS Press, 326–341.
10. Canfora, G., Fasolino, A.R., Frattolillo, G., and Tramontana, P. A wrapping approach for migrating legacy system interactive functionalities to service oriented architectures. *Journal of Systems and Software 81*, 4 (2008), 463–480.
11. Canfora, G., Cerulo, L., and Di Penta, M. Tracking your changes: A language-independent approach. *IEEE Software 27*, 1 (2009), 50–57.
12. Chikofsky, E., and Cross, J.I. Reverse engineering and design recovery: A taxonomy. *IEEE Software 7*, 1 (1990), 13–17.
13. Cifuentes, C. and Gough, K.J. Decompilation of binary programs. *Software Practice and Experience 25*, 7 (1995), 811–829.
14. Collard, M.L., Kagdi, H.H., and Maletic, J.I. An XML-based lightweight C++ fact extractor. In *Proc. of the 11th International Workshop on Program Comprehension*, 2003. IEEE CS Press, 134–143.
15. Cordy, J.R., Dean, T.R., Malton, A.J., Schneider, K.A. Source transformation in software engineering using the TXL transformation system. *Information & Software Technology 44*, 13 (2002), 827–837.
16. Cornelissen, B., Zaidman, A., van Deursen, A., Moonen, L., and Koschke, R. A systematic survey of program comprehension through dynamic analysis. *IEEE Transactions on Software Engineering.* http://doi.ieeecomputersociety.org/10.1109/TSE.2009.28
17. Duala-Ekoko, E., and Robillard, M.P. Tracking code clones in evolving software. In *Proc. of the 29th International Conference on Software Engineering*, 2007. IEEE CS Press, 158–167.
18. Ernst, M.D., Cockrell, J., Griswold, W.G., Notkin, D. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering 27*, 2 (2001), 1–25.
19. Ferenc, R., Beszédes, À., Tarkiainen, and M., Gyimòthy, T. Columbus—reverse engineering tool and schema for C++. In *Proc. of the 18th International Conference on Software Maintenance*, 2002. IEEE CS Press, 172–181.
20. Fluri, B., Würsch, M., Pinzger, M., Gall, H. Change distilling: Tree differencing for finegrained source code change extraction. *IEEE Transactions on Software Engineering 33*, 11 (2007), 725–743.
21. Godfrey, M.W., Hassan, A.E., Herbsleb, J.D., Murphy, G.C., Robillard, M.P., Devanbu, P.T., Mockus, A., Perry, D.E., and Notkin, D. Future of mining software archives: A roundtable. *IEEE Software 26*, 1 (2009), 67–70.
22. Gold, N., Knight, C., Mohan, A., and Munro, M. Understanding service-oriented software. *IEEE Software 21*, 2 (2004), 71–77.
23. Hassan, A.E., Martin, D.J., Flora, P., Mansfield, P., and Dietz, D. An industrial case study of customizing operational profiles using log compression. In *Proc. of the 30th international Conference on Software Engineering* 2008. ACM Press, 713–723.
24. Kagdi, H.H., Collard, M.L., and Maletic, J.I. A survey and taxonomy of approaches for mining software repositories in the context of software evolution. *Journal of Software Maintenance 19*, 2 (2007). Wiley, 77–131.
25. Kamiya, T., Kusumoto, S., and Inoue, K. CCFinder: A multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering 28*, 7, (2002), 654–670.
26. Koschke, R. Atomic architectural component recovery for program understanding and evolution. Ph.D. thesis, Univ. of Stuttgart, Germany, 2000.
27. Koschke, R. Software visualization in software maintenance, reverse engineering, and reengineering: A research survey. *Journal of Software Maintenance 15*, 2 (2003), Wiley, 87–109.
28. Kuhn, A. Ducasse, S., and Gîrba, T. Semantic clustering: Identifying topics in source code. *Information & Software Technology 49*, 3 (2007), 230–243.
29. Lanza, M. and Ducasse, S. Polymetric views—A lightweight visual approach to reverse engineering. *IEEE Transactions on Software Engineering 29*, 9 (2003), 782–795.
30. Lehman, M.M. and Belady, L.A. *Program Evolution—Processes of Software Change.* Academic Press, London, 1985.
31. Lorenzoli, D., Mariani, L., and Pezzè, M. Automatic generation of software behavioral models. In *Proc. of the 30th International Conference on Software Engineering*, 2008. ACM Press, 501–510.
32. Maqbool, O. and Babri, H.A. Hierarchical clustering for software architecture recovery. *IEEE Transactions on Software Engineering 33*, 11 (2007), 759–780.
33. Marin, M., van Deursen, A., and Moonen, L. Identifying crosscutting concerns using fan-in analysis. *ACM Transactions on Software Engineering and Methodology 17*, 1 (2007), 1–37.
34. Muller, H.A., Jahnke, J.H., Smith, D.B., Storey, M.D., Tilley, S.R., and Wong, K. Reverse engineering: A roadmap. In *Proc. of the 22nd International Conference on Software Engineering—Future of Software Engineering Track*, 2000, 47–60.
35. Moonen, L. Generating robust parsers using island grammars. In *Proc. of the 8th Working Conference on Reverse Engineering*, 2001. IEEE CS Press, 13–22.
36. Murphy, G.C. Houston: We are in overload. In *Proc. of the 23rd IEEE International Conference on Software Maintenance*, 2007, IEEE CS Press.
37. Nierstrasz, O., Ducasse, S., and Gîrba, T. The story of Moose: An agile reengineering environment. In *Proc. of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2005, Lisbon, Portugal, (Sept. 5-9, 2005). ACM Press, 1–10.
38. Ducasse, S, and Pollet, D. Software architecture reconstruction: A process-oriented taxonomy. *IEEE Transactions on Software Engineering 35*, 4 (2009), 573–591.
39. Resnick, P. and Varian, H.R. Recommender systems. *Comm. ACM 40*, 3 (Mar. 1997), 6–89.
40. Systa, T. *Static and Dynamic Reverse Engineering Techniques for Java Software Systems.* Ph.D. thesis, Univ. of Tampere, Finland, 2000.
41. Storey, M.D., and Müller, H.A. Manipulating and documenting software structures using Shrimp views. In *Proc. of the 11th International Conference on Software Maintenance*, 1995. IEEE CS Press, 275–284.
42. Tonella, P. and Potrich, A. *Reverse Engineering of Object Oriented Code.* Springer-Verlag, Berlin, Heidelberg, New York, 2005.
43. Tsantalis, N., Chatzigeorgiou, A., Stephanides, G., and Halkidis, S. T. Design pattern detection using similarity scoring. *IEEE Transactions on Software Engineering 32*, 11 (2006), 896–909.
44. Woodside, C.M., Franks, G., and Petriu, D.C. The future of software performance engineering. In *Proc. of the International Conference on Software Engineering, Future of Software Engineering Track (FOSE)* 2007. IEEE CS Press, 171–187.
45. Wong, K., Tilley, S., Muller, H.A., and Storey, M.D. Structural redocumentation: A case study. *IEEE Software* (Jan. 1996), 46–54.

Gerardo Canfora (gerardo.canfora@unisannio.it) is a professor of software engineering in the Department of Engineering at the University of Sannio, Benevento, Italy.

Massimiliano Di Penta (dipenta@unisannio.it) is an assistant professor in the Department of Engineering, at the University of Sannio, Benevento, Italy.

Luigi Cerulo (lcerulo@unisannio.it) is an assistant professor in the Department of Biological and Environmental Sciences at the University of Sannio, Benevento, Italy.