A Small Primer on Software Reverse Engineering

Hausi A. Müller and Holger M. Kienle

Technical Report

Department of Computer Science
University of Victoria
Canada

March 2009

A Small Primer on Software Reverse Engineering

Hausi A. Müller and Holger M. Kienle
Department of Computer Science
University of Victoria, Canada
{hausi,kienle}@cs.uvic.ca

Abstract

Software reverse engineering is a subfield of software engineering that is concerned with the analysis of an existing software system—often legacy—with the goal to synthesize information about the target system so that certain aspects of it can be reasoned about. System artifacts such as requirements specification, design documents, source code, version histories, documentation, use cases, or execution traces are analyzed with the goal to reveal or synthesize abstractions and to generate visualizations that expose call graphs, subsystem structures, high-level architectures, functional decompositions, code duplications, metrics, and run-time behavior. An important aspect of reverse engineering is that it generates information about the subject system at various levels of abstraction, ranging from code-centric views such as program slices to domain knowledge such as business rules. This synthesized information includes mappings and concept assignments between abstraction layers. Thus, reverse engineering provides valuable input for evolving software systems including activities such as program comprehension, reengineering, or maintenance.

1. Introduction

Reverse engineering has a long-standing tradition in many areas, ranging from traditional manufacturing to information-based industries. The common goal of reverse engineering in all of these areas is the extraction of know-how or knowledge from human-made artifacts. The process of reverse engineering typically starts with lower levels of information that describe the artifact under study. From this low-level information higher levels of knowledge and understanding are revealed or synthesized. Thus, reverse engineering can be seen as raising the abstraction level of the information that is available about the artifact under study. In the software domain, reverse engineering deals with artifacts that are created in the process of constructing a software system, most notably the system's code base. The IEEE Standard for Software Maintenance (IEEE Std 1219-1993) defines reverse engineering as “the process of extracting software system information (including documentation) from source code”. The goal is to construct mappings between low-level implementation artifacts and high-level design and domain concepts thereby closing the gap between computer-intensive and human-intensive artifacts.

Software reverse engineering is performed for a wide variety of reasons, ranging from gaining a better understanding of system components, over impact analysis while fixing a defect, to collecting information for making informed management decisions. Goal-oriented reverse engineering can be an integral part of a broad spectrum of different tasks. Examples of such tasks are program understanding, gathering quality measures, impact localization, program slicing, clone detection, object identification, concept assignment, subsystem composition, design recovery, or business rule extraction.

The notion of reverse engineering is best understood by looking at its context and role in software construction and evolution. While the classical software development process refines a system by going through the stages of requirements, design and implementation, reverse engineering starts with the implementation's artifacts and produces higher-level abstractions such as those found in the design and requirements stages (cf. Figure 1). For example, reverse engineering may produce UML class and interaction diagrams from Java source code. However, while reverse engineering targets abstractions that can be found in the early stages of software development, it generally is infeasible to recreate the information produced by these stages completely. From the reverse engineering perspective, the classical software development process is often referred to as forward engineering regardless of the employed life cycle model (e.g., waterfall or spiral model).

Reverse engineering is closely related to program comprehension (a.k.a. program understanding), reengineering, maintenance and evolution. The fields of reverse engineering and program comprehension have a mutualistic, symbiotic relationship. Information produced by reverse engineering (e.g., visualization of call graphs) aids reverse engineers in program comprehension. Conversely, program comprehension provides theories on how programmers read and reason about code that can be applied to make reverse engineering tools and techniques more effective [20]. The theory of bottom-

up comprehension assumes that reverse engineers start the comprehension process from code statements and then successively chunk knowledge into higher-level concepts up to the system goals. Reverse engineering techniques support bottom-up comprehension by following control and data flow dependencies and by grouping program entities into higher-level concepts such as abstract data types or subsystem structures. The theory of top-down comprehension assumes that the comprehension process starts out by postulating hypotheses about the nature of the system that are successively validated/refuted and refined based on beacons in the source code. This process is supported by searching for beacons, by presenting higher-level knowledge about the system in the form of "is-a" and "part-of" hierarchies and by matching architectural hypotheses to the actual system components. Finally, there are theories that combine elements of bottom-up and top-down comprehension. One of the main challenges in comprehending software is to establish a mapping between code and requirements [20]. Reverse engineering strategies to recover and document these mappings in a manual, semi-automatic, or automatic manner include techniques such as feature location, object and design pattern identification, central and fringe component characterization, or business rule extraction.

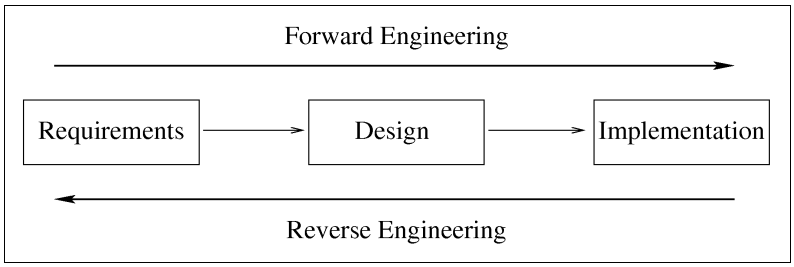


Figure 1: Relations of reverse and forward engineering to the software development process.

Reverse engineering strategies, techniques and technologies are not only crucial to maintaining and evolving the large base of installed legacy software systems, but also integral in modern integrated software development environments (e.g., support for refactoring or visualization in Eclipse). IEEE Std 1219-1993 recommends to use reverse engineering "if the documentation is not available or is insufficient and the source code is the only reliable representation". This is typically the case for legacy systems. Thus, a significant investment in understanding the system is needed before a maintenance activity can be performed. Reverse engineering can provide valuable information to assess the state of the legacy system and to estimate the effort and impact of a certain maintenance task. It is important to stress that the role of reverse engineering is restricted to synthesizing information about a subject system and not actually changing it. In particular, reverse engineering is a subtask of reengineering—which is composed of two steps: reverse engineering followed by forward engineering. Thus, a maintenance activity that affects the code involves reverse engineering followed by forward engineering.

For the past two decades the field of reverse engineering has produced powerful tools and techniques that enable us to analyze and reason about millions of lines of source code. Reverse engineering has tackled problems that require highly scalable algorithms for manipulating artifacts and synthesizing abstractions. One of the hardest problems is to summarize, organize and present the accumulated knowledge effectively to the different types of users (i.e., from novice to expert reverse engineers). Since reverse engineering has to accommodate existing artifacts, robust techniques are needed to analyze heterogeneous systems that are built using various programming models, languages, technologies, and platforms. The robustness, scalability and versatility of many of the reverse engineering techniques were aptly demonstrated during the Y2K analysis of vast amounts of deployed source code a decade ago.

Reverse engineering provides a standard process, established representations of artifacts, proven techniques, and tremendous tool support. In the following, we first characterize reverse engineering by system artifacts and subject systems. We then introduce the reverse engineering process consisting of three main activities: extraction, analysis, and visualization. In this process, abstraction plays a central role. Next, we discuss knowledge representation and manipulation, which is another issue that crosscuts the entire process. Finally, we turn to reverse engineering tools and techniques. Tool support is needed for each step in the process. We discuss important techniques for fact extraction (lexical and syntactic approaches), static and dynamic analyses, as well as canned and interactive visualizations. We close the paper with an outlook of expected developments in the reverse engineering field.

2. Reverse Engineering Domains and Subject System Artifacts

The broad range of the reverse engineering approaches is exposed by the diverse system artifacts and kinds of systems that the approaches are targeting. Historically, reverse engineering has focused on systems written in a single high-level

language and this area remains one of the major research foci. This can be explained by the fact that the source code constitutes the heart of the system and is thus typically the most important artifact that drives the reverse engineering process. Furthermore, ultimately source code is the only reliable source of information that provides an accurate description of the target system—other artifacts such as requirements and design documents may be incomplete, inaccurate, or out-of-date [17]. Determining the differences between the original design documents and the actual source code is in itself a difficult reverse engineering problem.

While the system's source code may consist of a single high-level language, such as COBOL, C++, or Java, for complex industrial and legacy systems, it is more likely that the code base includes a number of heterogeneous sources that can range from assembly language to domain-specific languages. To reverse engineer such systems, dependencies that cross language boundaries have to be uncovered and traced.

Besides artifacts that make up the system proper, there are other supporting artifacts such as natural-language requirements specifications, design documents (e.g., UML diagrams or video recordings of design meetings), bug databases (e.g., Bugzilla), source code comments, commit comments in code repositories, or messages among the developers. All of these artifacts are potential candidates as input for the reverse engineering process. However, while it is well understood how to process artificial languages (e.g., leveraging techniques from parsing theory) and how to extract useful information from it, it is much more difficult and error-prone to process artifacts based on natural languages such as source code comments or voice communication of developers. As a consequence, comparably few reverse engineering approaches make effective use of natural language artifacts.

Besides source code, there are distinct reverse engineering disciplines that target particular kinds of artifacts. Binary reverse engineering targets software that is only available in binary form, which includes binary, object, intermediate (e.g., W-code), or assembly code. Binary reverse engineering is particularly useful when the source code is lost—a surprisingly common phenomenon for legacy systems. Compared to analyzing source code, reverse engineering of binaries poses unique challenges—distinguishing program code from data, assigning meaningful variable names, or recovering structured control flow from unstructured one. Virtual machine code formats such as the Java Virtual Machine (JVM) or the Common Intermediate Language (CIL) provide a higher level of abstraction compared to binary form, making the reverse engineering of these formats closer to the reverse engineering of source code.

Data reverse engineering recognizes that many industrial and legacy systems are management information systems that handle large volumes of persistent data by integrating a database management system [17]. Thus, besides code, data is another important asset that needs to be understood. This fact became widely acknowledged with the Y2K and EUR currency conversions. While data reverse engineering often targets the persistent data structures of relational databases, this field is much broader with the goal to “help an organization determine the structure, function, and meaning of its data” [4].

The youngest discipline in the reverse engineering field targets web-based systems. For web sites the primary artifacts are HTML pages and hyperlinks. In contrast, web applications are realized with advanced technologies such as AJAX, JSP.NET, or J2EE web applications. As a result, advanced web-based systems are complex pieces of software, combining diverse functionality such as found in database, distributed, or hypermedia systems.

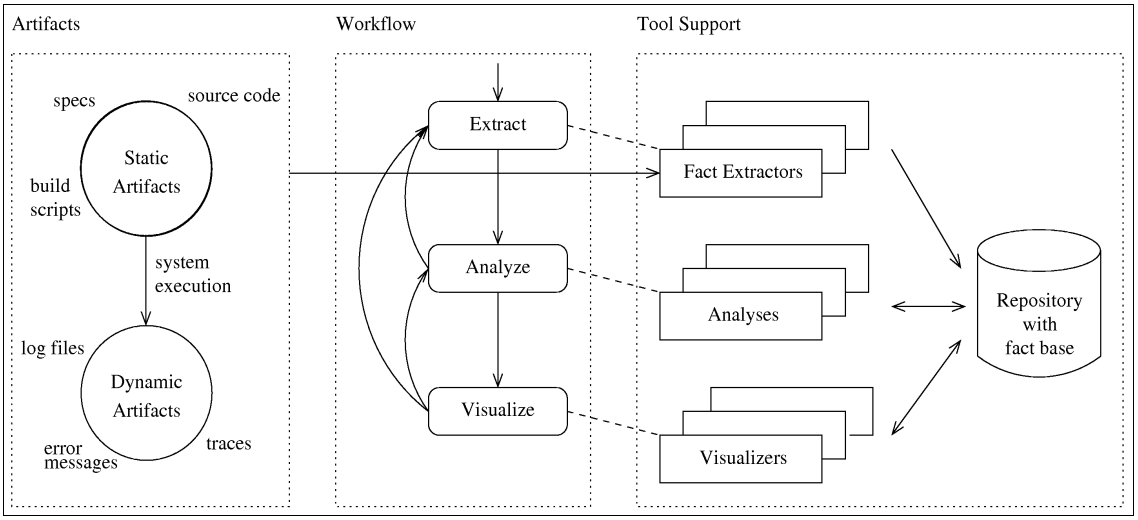


Figure 2: High-level view of the reverse engineering process workflow, its inputs, and associated tool support.

3. The Reverse Engineering Process

When conducting a reverse engineering activity, the reverse engineer follows a certain process. Chikofsky and Cross define software reverse engineering as “the process of analyzing a subject system to [...] create representations of the system in another form or at a higher level of abstraction” [1]. This definition is perhaps the most cited and most accepted in the field. It describes reverse engineering as a process of discovery that takes the system's artifacts as input and produces information about these artifacts as output. The definition does neither mandate the process's inputs nor its outputs. In fact, reverse engineering has continually broadened and adapted its inputs as well as its outputs in its quest to provide more relevant information to engineers performing reverse engineering tasks.

At a more detailed level, the workflow of the reverse engineering process can be decomposed into three subtasks: extraction, analysis, and visualization (cf. Figure 2). To illustrate this process, we describe the subtasks that a reverse engineer might perform to obtain the call graph and the subsystem structures of a system. A call graph is a popular and universal abstraction that represents call dependencies between program entities such as procedures or files [19]. The notion of a call graph mitigates complexity by reducing procedures, files, and calls in the system to atomic concepts that the reverse engineer can reason about without having to understand the underlying details. Another important reverse engineering technique is architecture reconstruction, which strives to identify subsystem structures and architectural views of the whole system from the perspective of a set of related concerns [23]. A popular view is a subsystem composition that partitions the system into hierarchical or layered subsystem structures with dependencies among these subsystems as depicted in Figure 3 [12].

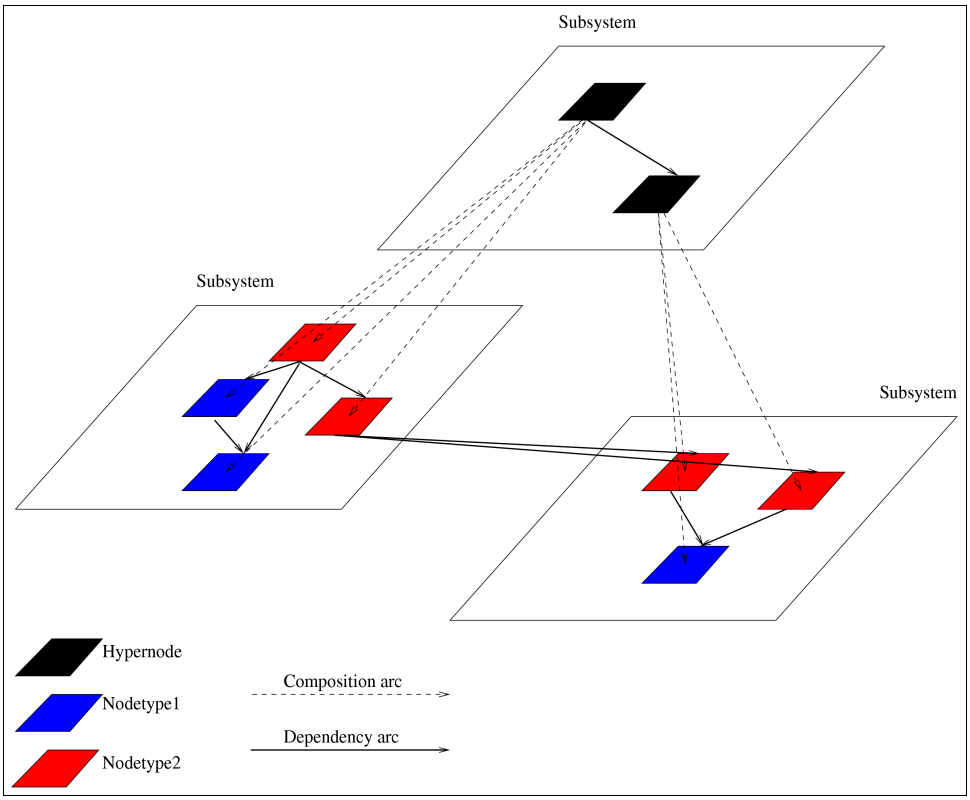


Figure 3: Graph model for subsystem composition with hypernodes and composite arcs.

3.1. First Subtask: Extraction of Facts about the System

A reverse engineering activity starts with the extraction of low-level information—so-called facts—from the system's artifacts. Most notable, this is the code base of the system, consisting of the source code along with supporting artifacts such as build scripts, database scripts, configuration files, version, variant and revision information, or test harnesses. Moreover, dynamic information about the system in the form of execution traces and log files can extend the fact base significantly.

For a call graph, only facts from the source code itself need be extracted; this is particularly useful if the system is not in an executable state. The extraction needs to identify the procedures (or methods) as well as the calls made within the

procedures. Furthermore, the artifact locations within the source code (i.e., source file name and line number) are typically of interest. While this construction of a call graph extractor is well understood, it is surprisingly difficult to build a sound and robust one [19].

Consider the (simplified) C code in Figure 4. In this case, the extractor needs to identify the C functions within the source (i.e., main, f, and g) as well as the calls within the functions. While the extraction of such information may seem trivial at first glance, there are cases that make it difficult or impossible to obtain complete and accurate information. Calling a function using a function pointer makes it difficult to recognize the call and also obscures the target of the call. C preprocessor directives can further exacerbate the call graph extraction process, since macro calls and function calls are syntactically indistinguishable and since the function names can be assembled using the preprocessor concatenation operator.

```
/* file main.c */                          /* file utils.c */

void main() { ... f(); ... g(); ... }      void f() { ... g(); ... }
                                           void g() { ... g(); ... }
```

Figure 4: Sample C program to illustrate the concept of a call graph.

For architecture reconstruction, the required facts depend on the desired views of the system. For a subsystem composition, program entities such as procedures, type definitions, or global variables need be identified along with usage dependencies such as calls to procedures or accesses to global variables. Thus, subsystem composition and call graph construction require different kinds of facts.

3.2. Second Subtask: Analysis and Knowledge Discovery of the Extracted Facts

In this task certain analyses are performed that use the facts generated in the extraction step. Typically, analyses reveal or generate additional higher-level knowledge based on the extracted, low-level facts.

To obtain a call graph, the analysis is quite simple because it only has to match procedure calls with the corresponding procedure definitions. With this information it is possible to construct a standard data structure that represents the call graph. However, this matching is not necessarily trivial. For instance, a call via a function pointer in C or a dynamic method call in C++ can have multiple potential targets. In such cases, more sophisticated analyses such as pointer analysis can reduce the number of call targets. For subsystem composition, the analysis needs to partition the program entities into sets where each set corresponds to a subsystem. The analysis is guided by the dependencies between the program entities and heuristics such as naming of program entities or software engineering principles such as high cohesion within a subsystem or low coupling among subsystems.

The potential power of an analysis and its characteristics depends on the kinds of facts that are produced by the extractor. If the extractor provides unsound facts, the analysis will also be unsound. If the extractor does not provide facts at a level of detail that enables the analysis to construct a control flow or dependency graph, pointer analysis is infeasible. A heuristic for subsystem composition can be more effective if more dependencies among program entities are provided by the fact base.

Besides analysis, the reverse engineer performs knowledge discovery and organization in this subtask. If higher-level knowledge is generated from low-level facts, this knowledge need be modeled and structured. The reverse engineer can use abstraction mechanisms such as classification, generalization/specialization, aggregation, or arbitrary grouping [22]. Extracting inheritance relationships leads to generalization or specialization hierarchies; composition of artifacts into subsystem structures produces aggregation or part-of hierarchies; and separating concerns or documenting side-effects involves arbitrary grouping.

3.3. Third Subtask: Visualization of Analysis Results

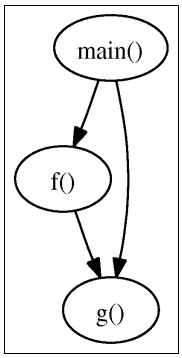
The pure analysis results are typically not fit for human consumption. The results have to be summarized and made palatable for human reverse engineers in textual and/or graphical form.

A call graph, as any graph, can be expressed in textual form. For example, the call graph of the above C code can be represented as a calls-matrix (cf. Figure 5(a)). However a graphical rendering with nodes representing procedures and arcs representing calls might be more effective for the human reverse engineer (cf. Figure 5(b)). If the call graph is depicted with a graph visualization tool, the reverse engineer can typically explore the graph interactively (e.g., by zooming in on

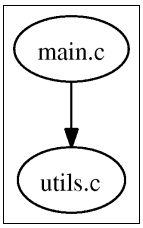
artifacts of interest, filtering arcs and nodes, applying layout algorithms, or collapsing groups of nodes into compound nodes). Furthermore, it is possible to navigate from nodes and arcs to the corresponding source code that they represent. Figure 6 shows a screenshot of the Rigi graph visualization engine [13]. The bottom left of the screenshot shows a graph window rendering the call graph of Figure 5(b).

calls	main()	f()	g()
main()	–	✓	✓
f()	–	–	✓
g()	–	–	✓

(a)



(b)



(c)

Figure 5: Visual representations of the call graph for the C program depicted in Fig. 4. The call graph can be visualized with (a) a dependency matrix, (b) a graph, (c) a graph that hoists up the dependency relationship to the file level.

The subsystem composition of a system can also be visualized naturally as a graph where nodes represent subsystems and arcs represent the dependencies among the subsystems. A subsystem composition could be also rendered in a standardized form such as a UML diagram. (A UML diagram can be seen as a graph that has standardized shapes for node and arc types.) A very simple heuristic for subsystem composition groups all entities that belong to the same file into a subsystem. The directory structure can then be used to obtain a hierarchical subsystem structure. For the C code in Figure 4, there are only two files (`main.c` and `utils.c`) and a flat directory structure. Grouping the functions into file-based subsystems and hoisting the call dependencies up to the file level results in the graph depicted in Figure 5(c).

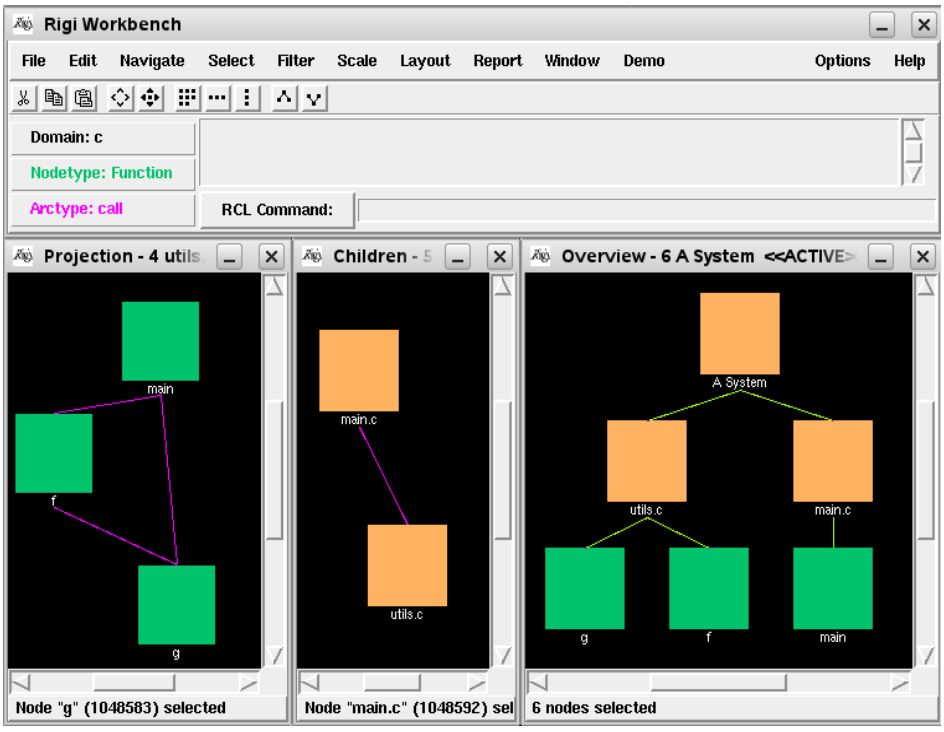


Figure 6: Screenshot of the Rigi tool depicting the call graph that corresponds to the C program in Figure 4.

In a graph viewer, the subsystem hierarchy can be visualized as a tree (i.e., part-of hierarchy) where the root represents the whole system and branches from a tree node represent subsystem refinements. The right graph window in Figure 6 shows

this hierarchy for the file-based grouping. The top-level node represent the whole system, the second-level nodes represent the two files (`main.c` and `utils.c`), and the third-level nodes represent the functions that belong to the files (`main`, `f` and `g`). Importantly, interactive exploration can be supported in a graph editor with hyper-nodes (i.e., nodes that can contain sub-graphs). In this case, a hyper-node represents a whole subsystem, which can be interactively "opened-up" to reveal its constituents (i.e., other subsystems or atomic entities, such as procedures or global variables). The graph window in the middle of Figure 6 contains two hyper-nodes; in Rigi these node can be opened-up by double-clicking on them.

3.4. Characteristics of the Reverse Engineering Process

Grouping the reverse engineering process into subtasks may give the impression that they are performed in a particular order. This, however, is an idealization because in reality the process is often ad hoc and the reverse engineer frequently iterates over the subtasks to obtain meaningful results [23]. This is illustrated in Figure 2 by the back-arcs in the workflow. For example, performing an analysis may reveal that the analysis yields unsatisfactory results because of missing facts from the extractor. In this case, the extractor need be extended and the analysis need be adapted accordingly. Similarly, the visualization of a subsystem analysis may yield a result that does not conform to the expected outcome. In this case, the analysis need be rerun with different parameters, or another analysis can be tried that is based on different clustering heuristics or similarity measures.

So far, we have described the process from a technical perspective. This is appropriate for small-scale reverse engineering activities that are typically performed by a single engineer to answer relatively well-defined questions about a subject system. With appropriate infrastructure, such questions can be answered in short order. This is the case for simple corrective maintenance scenarios where only a few artifacts are eventually modified.

In contrast, reverse engineering in-the-large may be long-running projects that involve several technical as well as non-technical stakeholders. Architecture reconstruction with the goal to migrate the whole system to a new language, platform, or technology is such an example. In this case, the technical reverse engineering process described above is embedded into an overarching maintenance or evolution process. First, a problem statement is formulated that involves all stakeholders [23]. This is then used by the process designer to define suitable architectural viewpoints (perhaps based on a library of available viewpoints) to tackle the problem, and to identify the necessary artifacts and facts that are needed to produce the viewpoints. Based on this, the extraction, analyses, and visualizations are designed and implemented. As mentioned before, this process is iterative (i.e., each of these steps is repeated and refined several times) before arriving at suitable architectural viewpoints. The resulting viewpoints of the reverse engineering process are then used as input to guide the maintenance or evolution process.

4. The Role of Abstraction

As should be apparent by now, abstraction plays a paramount role in the reverse engineering process. To characterize and contrast different abstractions we distinguish five levels: source text, structural, functional, architectural, and application [15] [9] [22]. Generally, higher levels of abstractions are more compact and easier to understand because they have a shorter "conceptual distance" to the problem domain.

The lowest abstraction level denotes the source code itself. This level is the starting point for the extraction of facts. The structural level is concerned with the representation of the source code in a form that makes it possible to reason about the source code in terms of source-level entities and relationships. Thus, at this level the abstractions still directly correspond to programming language constructs. Typical representations of facts at this level are control and data flow graphs, abstract syntax trees, and symbol tables of lexical tokes.

The functional abstraction describes the system with higher-level entities such as files, modules, procedures, or data and relationships among these entities such as calls, data accesses, or includes. The functional level represents logical and functional specifications of procedures (e.g., in the form of pre-, post-conditions, or invariants), program plans, or design patterns. At this level programming-language details of entities are abstracted away. Typical representations of facts at this level are resource flow graphs. Figure 7 shows the entities and relations used in the resource flow graph of the Bauhaus environment [11]. Entities are organized in a type hierarchy.

A call graph is an example of a basic functional abstraction. It provides an abstraction from the structural level because it omits details of the actual implementation such as the employed programming language, call semantics, passed parameters, or the number of calls. In Bauhaus, a C function is modeled as a Subprogram entity. The architectural level provides a composition of the concepts found in the functional and structural levels into architectural viewpoints and styles. Depending on the viewpoint, this level introduces entities such as subsystems, components, or processes and dependencies such as uses, compilation, instantiations, inter-process communications, or side-effects. In order to support subsystem composition, the

Bauhaus resource flow graph introduces a Component entity that allows recursive nesting of Components. In contrast, a Module does not allow recursive nesting, thus corresponding to a C file. Finally, the highest abstraction is the application level, which is concerned with domain-specific concepts including goals, business rules, policies, as well as functional and non-functional requirements.

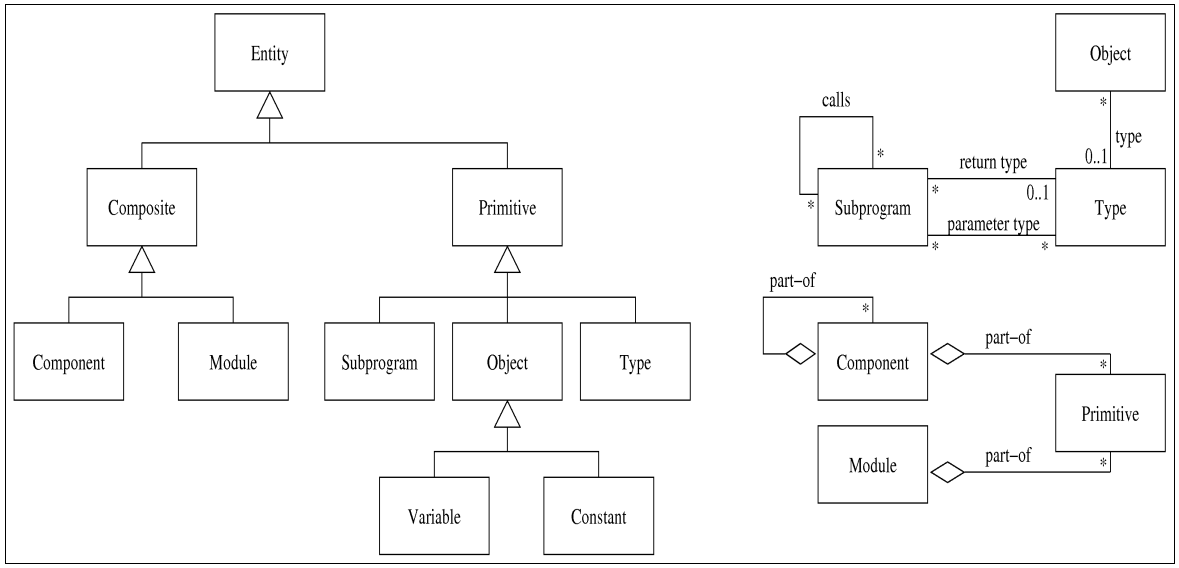


Figure 7: Data model of the Bauhaus resource flow graph with entities (left) and relationships (right).

These layers of abstractions and their relationships in the context of reverse and forward engineering are summarized in the Horseshoe model as depicted in Figure 8 [15]. The left side of the horseshoe represents reverse engineering, which starts from the source code and targets one of the abstraction levels. The right side represents forward engineering, which starts out with a system description at a certain abstraction level with the goal to produce source code. Transformations operate at the joint point of reverse and forward engineering by changing the representation of the system at a certain abstraction level. A transformation at the structural level from one representation to another while preserving the subject system's behavior is often referred to as code restructuring. In contrast, architectural reengineering operates at the architectural level, requiring architecture and design recovery, followed by architecture-driven forward engineering to reconstitute the system so that it adheres to the desired architecture. The transition from the initial to the desired architecture can be seen as an architecture-level transformation, or rearchitecting.

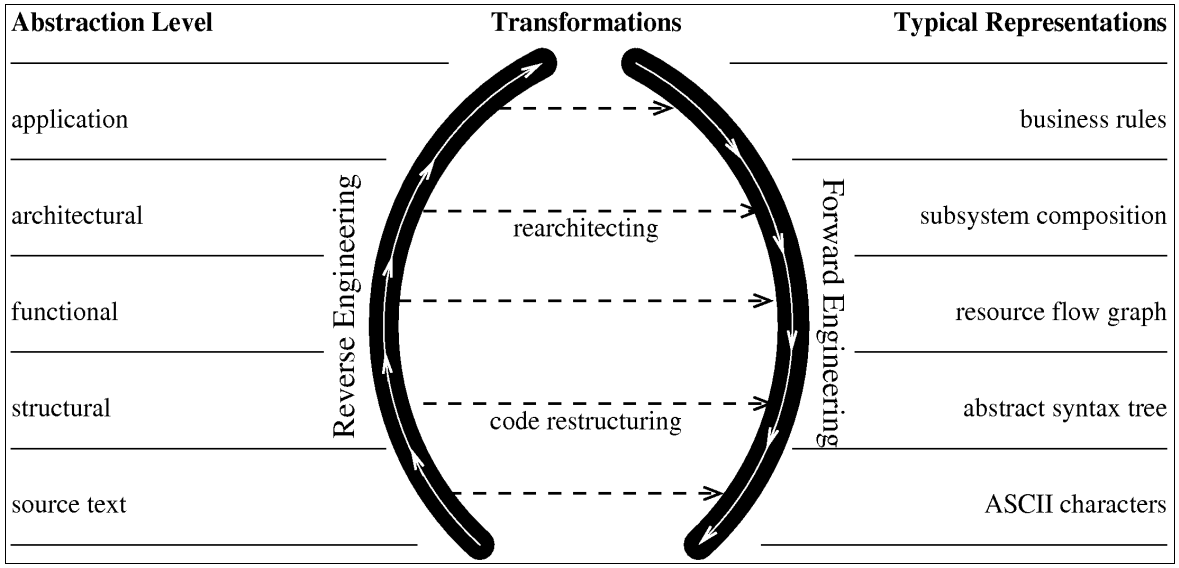


Figure 8: The Horseshoe model depicts the relationships between abstraction levels and transformations between different stages of forward and reverse engineering.

Traceability between the different abstraction levels is an important goal during the reverse engineering process. The mappings between the generated abstractions down to the lower levels should be recorded. This means for example, that dependencies between subsystems can be traced back to lower-level dependencies (e.g., dependencies between subsystems can be traced down to calls between functions). Similarly, extracted business rules should be mapped down to the source code that implements them. These mappings are typically not 1:1, but rather 1:n or even n:m. A key objective of a reverse engineering technique is to preserve the mappings even while the underlying source code evolves over time.

5. Information Representation and Manipulation

An overarching concern of reverse engineering is the representation and manipulation of information about the subject system. Over the years, researchers have proposed many data models, schemas and ontologies for different levels of abstraction to represent the information. Storing (i.e., in structured and unstructured form) and querying a huge instance of a fact schema is a challenge in its own right. These functionalities are provided by reverse engineering repositories.

A repository is the most central component that ties together the reverse engineering process (cf. Figure 2). The repository gets populated with facts extracted from the target software system. Analysis algorithms read information from the repository and possibly augment it with further information. Information stored in the repository is presented to the user with visualizers. The reverse engineer may annotate information and organize knowledge using the visualization engine, which is then mirrored in the repository.

Examples of concrete implementations of repositories range from simple text files, to domain-specific solutions, to commercial databases. Often reverse engineering information is stored in text files with the help of exchange formats [10]. Examples of exchange formats developed by the reverse engineering community include Rigi Standard Format (RSF) [13], Tuple-Attribute language (TA) [6], and Graph Exchange Language (GXL). The latter is based on a community effort and is now the de facto standard. To give an example, in TA the call graph for Figure 4 can be encoded as follows:

```
1     $INSTANCE main Function
2     $INSTANCE f Function
3     $INSTANCE g Function
4     Call main f
5     Call main g
6     Call f g
7     Call g g
```

Each relation is expressed as a triple separated by blanks (i.e., written in verb-subject-object order). Thus, the first line should be interpreted as main() is an instance of the Function type, and the fifth line should be read as main() calls f().

All of these exchange formats are graph-based in the sense that they support the encoding of information in the form of nodes, arcs, and attributes. In TA, attributes can be attached to triples as follows:

```
1     Call main f { numberOfCalls = 1 }
```

Graph-based data models are a natural fit for the reverse engineering domain—nodes represent entities and arcs represent relationships among the entities as depicted in Figure 3. Furthermore, graph-based data structures can be exploited by dedicated query languages [14]. For example, RSF can be queried with the CrocoPat language and TA comes with the Grok query language, which is based on Tarski binary relational algebra [6]. Besides domain-specific solutions, reverse engineering has leveraged existing technologies such as XML, relational databases, or the Eclipse Modeling Framework (EMF). This has the advantage that existing query facilities come with the technology (e.g., XPath, SQL, or EMF's Model Query).

Information in a repository adheres to a data model. Depending on the repository type, the data can be fixed, extensible, or general. A fixed data model is hard-wired into the repository. For a graph-based model, this means that the sets of node and arc types are fixed. As a result it is difficult to accommodate new entities or relationships. In contrast, extensible data models provide a fixed framework that can be extended if the need arises (e.g., by sub-classing of an existing node type). A general exchange format can store any information (i.e., node and arc types can be freely defined). A fixed repository is appropriate if it is tied to a particular reverse engineering approach that is not expected to change. A flexible repository is useful if the domain that the reverse engineering approach targets is well understood and can be expressed in a data model. For example, this is the case for higher-level information about object-oriented languages that can be expressed in a common core data model. Language specific information can then be provided by extension from the core model.

Importantly, this facilitates the construction of language-agnostic analyses that operate exclusively on the core model. A general repository provides full flexibility in defining the data model. The graph-based exchange formats discussed above or relational databases are general.

The data model for a call graph can be expressed in TA as follows:

```
1     $INHERIT Function $ENTITY
2     $INHERIT Call $RELATION
3     Call Function Function { numberOfCalls }
```

The first line states that there is a Function type that is a descendant of the top-level type $ENTITY. Analogously, the second line states that Call is a relation derived from the root relationship type. The third line states that the Calls relation allows only entities of the Function type as subjects and objects. (The first two lines are implied by TA and are only given here to expose the underlying type system.)

6. Techniques and Tools

Manual gathering and inspection of artifacts as part of the reverse engineering process can be effective in some limited circumstances—scanning of system documents, understanding the system's directory structure, and observing the system's behavior during execution [22] [17] [23]. However, for many reverse engineering tasks tool support is highly desirable. The dashed lines in Figure 2 show that each subtask is associated with a set of corresponding tools. Sophisticated analyses (e.g., subsystem composition based on heuristics) can provide information about a system that is difficult or practically impossible to infer manually. Another issue related to tool support is automation of repetitive tasks. This makes it possible to quickly reproduce steps in the process if the target system changes. In principle, a reverse engineer could construct manually, say, a call graph without tool support by inspecting the entire source code. However, this activity is slow, error-prone, and needs to be repeated whenever the source code changes.

Two crucial tool requirements are scalability and programmability. Since a legacy system can comprise millions source code lines, scalability in both time and space (i.e., efficient algorithms and data structures) is crucial for the entire tool chain. For smaller-scale, interactive comprehension tasks the reverse engineer needs instantaneous feedback. Also, a tool should be programmable by the end-user so that repetitive tasks can be automated and new capabilities can be added as needed. Programmability can be supported by providing a scripting language (e.g., Tcl or JavaScript), a plug-in interface (e.g., web browsers or Eclipse), an API, or a component interface (e.g., Java Beans). All of these technologies have been employed in reverse engineering tools. For example, the Rigi tool is end-user programmable and extensible using the Tcl scripting language [13]. Other crucial requirements are tool usability and adoption. An important research area within reverse engineering is tool evaluations based on comparisons, case studies, benchmarks and user studies [17].

A reverse engineering tool can be human-excluded or human-involved. Generally, the iterative nature of the reverse engineering process and the fact that program comprehension is a creative act necessitates tools that are human-involved. A human-excluded technique is fully automated, executing in batch mode. This approach is found in report-generation tools that produce information such as quality measures or static call graphs. In contrast, a human-involved tool allows the reverse engineer to provide input and guidance for analyses and visualizations upfront and during the entire reverse engineering task. An example of a human-involved analysis is semi-automatic subsystem composition that involves the reverse engineer into the identification process [11]. In this approach the engineer iterates over (1) analysis, (2) metric-based ranking, and (3) result presentation and acceptance. First, the user selects one of several analysis techniques for subsystem detection. The selected analysis is applied incrementally in the sense that it takes subsystems that have been already confirmed by the reverse engineer into account. Subsystem candidates are then presented to the reverse engineer and ranked based on user-selected metrics. The reverse engineer decides which subsystems to accept and then the cycle can start again. The rational for a semi-automatic approach is the realization that architecture recovery cannot be fully automated and thus human expertise is an integral part of the process. On the other hand, as much as possible should be automated to free the reverse engineer of boring, error-prone and repetitive tasks.

6.1. Tool Architectures

Tools for reverse engineering can be grouped into fact extractors, analyzers, and visualizers according to the functionalities that the tool provides. This grouping directly reflects the three subtasks of the reverse engineering process outlined above (cf. Figure 2). The two main approaches to tool construction are an integrated environment and a set of stand-alone tools.

An integrated environment supports the whole process within a single, coherent tool that typically has a consistent user interface and offers presentation integration. This approach has the benefit that the reverse engineer can navigate seamlessly

and interactively across the complete functionality that the tool offers. In contrast, a set of stand-alone tools has to rely on looser coupling among its tools based on control and data integration. Also, each tool has its unique, perhaps idiosyncratic, user interface. On the positive side, a set of independent tools enables the opportunistic assembly of a tool chain to match a certain reverse engineering task. For example, the Dali workbench for architecture recovery [12] is based on individual tools for fact extraction (i.e., LSME, Imagix, SNIFF+ and Perl), static analysis (i.e., Rigi, SQL and RMTool), dynamic analysis (i.e., gprof), and visualization (i.e., Rigi). The tool set implements data integration via a relational database (i.e., PostgreSQL). For this type of architecture, a main challenge is to facilitate repository synchronization and data consistency among the tools.

A popular approach to tool construction is to support presentation integration of analyses and visualizations in a common tool and to decouple the extractor from this tool by data integration with a repository (i.e., exchange format or database). Tight control and presentation integration between the analysis and visualization engines is highly desirable for semi-automatic analyses. The Rigi environment is an example of a tool set that follows this approach [13].

6.2. Extraction and Analysis Techniques

Naturally, fact extraction techniques depend on the artifacts of the subject system. Extraction of natural language text requires different techniques compared to artificial language code. Extraction of static system artifacts is different from extracting facts from the running system. Also, analyses are often tailored to match the granularity of the extracted facts. Table 1 shows examples of static reverse engineering and reengineering analyses along with the abstraction level on which they operate. Note that there are analyses, such as clone detection, that can operate on several abstraction levels depending on the analysis technique. In the following, we first discuss static extraction and analysis techniques, followed by dynamic ones.

Table 1: Analysis and transformation examples for the abstraction levels of the Horseshoe model.

Abstraction level	**Reverse engineering analyses**	**Reengineering transformations**
Source code	Lexical search, clone detection	Search-and-replace
Code structure	Coding standards conformance, identification of loop invariants, dependency analysis, impact analysis, complexity measures, program slicing, clone detection, feature location, object identification, high cohesion within components	Pretty printing, restructuring of control flow, goto elimination, redundant or dead code identification, transliteration, source code transformation (e.g., Y2K or EUR conversions)
Functional	Call graph, clustering, abstract data-type identification, central and fringe component identification, low coupling among components	Objectification, wrapping, migration from one programming language to another, migration to an object-oriented language
Architectural	Architecture recovery, subsystem composition, cliché recognition, concept assignments, separation of concerns, identifying aspects, identifying adaptive and non-adaptive components	Rearchitecting, architecture migration, wrapping, user-interface migration, database migration, middleware migration, parallelization, migration to cloud platforms
Application	Business rule extraction, domain concept identification, redocumentation	Policy and business rule evolution, migration from one domain to another

Static Techniques

For static, textual information, we can distinguish between artificial and natural languages. Extraction and analysis of natural language text is addressed by natural language processing (NLP) techniques. Reverse engineering mostly relies on NLP techniques that have been developed for information retrieval, computer linguistics and artificial intelligence including keyword extraction, statistical parsing, part-of-speech tagging, or latent semantic indexing (LSI). For example, LSI can be used to recover traceability links between natural language specifications and source code [18]. Of course NLP techniques can also be applied to source code. For example, LSI has been used to find similarities between source code fragments to identify code clones and abstract data types.

For artificial languages there is a wide variety of fact extraction techniques with different characteristics [23]. Syntactical approaches are based on parsing theory. The parser is based on a grammar and typically creates an abstract syntax tree that is either stored in a repository or kept in memory data structures—which can then be queried by analysis engines. This approach provides accurate facts at a high level of detail, but it may not scale to fact bases of multi-million line programs. To make this approach more scalable, partial abstract syntax trees can be constructed when needed. A syntactical approach to fact extraction is necessary if the reverse engineer wants to employ analyses that are based on detailed control and data flow information (e.g., pointer analysis or slicing). However, this approach has robustness problems since the extractor cannot recover from a parsing error. In practice, this is a serious problem for complex programming languages because they often have various dialects or versions, and source code may contain embedded languages such as assembly or SQL, or may rely on proprietary compiler extensions. Thus, parsers employed for reverse engineering tasks have to be more robust or forgiving than the parsers used in compilers.

On the other side of this spectrum are lexical approaches, which extract facts based on grep-style pattern matching. Text processing tools such as awk, lex, and LSME allow to associate actions based on matching patterns. As a consequence, this approach is well suited for partial fact extraction that targets a particular analysis (e.g., call graph construction) [19]. This approach is less brittle than parsing, but the provided facts may be unsound because they can contain false positives (i.e., wrongly included facts) or false negatives (i.e., missing facts). The reverse engineer must then determine whether the extractor's precision is sufficient by iteratively inspecting the extracted facts and analysis results. In practice, unsound results are still valuable to the reverse engineer for many reverse engineering tasks. For example, a call graph that is missing a small fraction of the calls is still useful for detecting organizational patterns in the code. Lastly, there are approaches such as fuzzy parsing and island grammars that can be seen as a hybrid between syntactic and lexical extractors because they are parsing selected chunks of the code while skipping over other parts.

As depicted in Table 1, there are many reverse engineering analyses that use static information. Among the more important ones are subsystem composition (as discussed above), program slicing, and clone detection. Program slicing is an analysis that simplifies the source code by removing parts that have no effect according to a certain semantic criterion [7]. The most common form of slicing removes all statements that do not affect or use a variable at a point of interest in the program. A backward slice identifies those statements—starting from the point of interest and working backwards—that affect the variable. A forward slice includes those statements—moving forward from the point of interest—that can be affected by the variable. To compute a slice the analyses needs control and data flow information. The notion of a slice can also be useful for other programming language constructs (e.g., type, parameter or interface) and be extended to other viewpoints at different levels of abstraction. Since slicing reduces the amount of code or concepts that a reverse engineer needs to reason about, it facilitates program understanding.

Duplicated, or cloned, code in legacy systems is a common phenomenon. Code clones are source code segments that are structurally or syntactically similar. Clones are often the result of copy-and-paste (-and-adapt) programming or generative programming. Identifying clones is a useful task during maintenance. Clones may cause maintenance problems because (1) errors may have been replicated in several clones and (2) a modification to one clone (e.g., bug fix) should probably be applied to all of its clones. Besides assessing the amount of duplicated code in the system, clone detection is used for change tracking and as a starting point for (object-oriented) refactoring. The precise definition of a clone depends on the actual clone analysis algorithm (e.g., some algorithms consider only whole procedures as clone candidates). Clone analyses can operate at several abstraction levels. At the textual level, clones can be discovered with simple string matching. Instead of looking for exact matches, white spaces and comments can be removed to catch slight variations (e.g., the level of indentation) in the code. This approach is simple to implement and mostly language independent. Lexical approaches can use regular expressions to group the source into tokens. For example, each identifier can be changed to an identifier marker to catch clones that have renamed variables. The syntactic level uses the structure of ASTs to identify clones based on tree matching, possibly also considering control and data-flow information. Identifying clones at the semantic level can be seen as looking for clichés.

Dynamic Techniques

In contrast to static techniques, dynamic extractors and analyses enable the reverse engineer to obtain information from the system's execution. While reverse engineering has focused historically on static information, dynamic techniques have become now a standard technique, especially to augment static results. An important characteristic of dynamic techniques is that they are unsound in the sense that trace information is based on a subset of all possible program executions. (Software testing has similar limitations.) As a consequence, a call graph based on dynamic information will show a subset of a sound, static call graph. Similar to static, unsound analyses, dynamic information can be very useful for program comprehension because they are more precise. For a dynamically dispatching call site, a static analysis may give many possible targets which are never invoked in practice, whereas a dynamic analysis may only show a few targets based on the actual calls during a certain program run.

Extraction of dynamic facts is accomplished by instrumentation of the code (both source and binary), or of the execution environment (e.g., operating system or virtual machine). This can be conveniently accomplished with debugging and profiling tools, aspect-oriented programming, or available hooks into the execution environment such as the Java Virtual Machine Tool Interface (JVMTI). The resulting low-level facts are called traces and provide information about executing sequences at various abstraction levels [8]. At the lowest level, traces can provide information about the execution of every single statement. This level is important for maintenance activities such as bug fixing. However, the more fine-grained the tracing, the more overhead is required in terms of space and time. Thus, statement-level tracing is only feasible for limited code regions. Also, tracing information needs an efficient representation such as the Compact Trace Format for storage and analysis [8]. Many tools target the tracing of procedure or method calls because scalability is less of a concern. The highest abstraction level groups procedure or classes into subsystems, tracing the interactions among the subsystems.

Important dynamic analyses techniques include trace analysis, design recovery, and feature location [3]. Trace analysis addresses scalability issues (in terms of space and time efficiency as well as reducing reverse engineers' information overload) with techniques such as pattern summarization, sampling heuristics, or metrics. Design recovery studies interaction patterns with the goal to reveal higher-level concepts. For example, there are tools that detect design patterns, construct state machines, and recover protocols from traces. Feature location is a technique that relates higher-level functionality to source code regions. To locate a feature, first test cases are executed that make use of the feature and then the resulting traces are intersected.

6.3. Visualization Techniques

Fact extraction, analyses, and knowledge organization yield huge amounts of information about the subject system. As a result, visualization techniques are needed that offer suitable views into this vast information space that facilitate the tasks of the reverse engineer. Mirroring static and dynamic analyses, visualization can either represent static structure or run-time behavior. A basic form of static visualization is code pretty-printers that provide automatic indentation and color-coding.

There are non-interactive visualizations that cannot be directly manipulated by the reverse engineer. The call graphs in Figure 5b and 5c were created with the dot graph drawing tool, which can generate non-interactive graphs in a number of vector and bitmap formats. In contrast, interactive visualizations are more common and can be manipulated by the reverse engineer. For example, the Rigi graph editor (cf. Figure 6) allows to drag-and-drop nodes of a graph, apply graph layouts, render nodes and arcs with different colors, and filter out nodes and arcs [13].

Interactivity is important to allow the navigation of the information space. In fact, most program comprehension happens during information exploration through iterative refinement of hypotheses [22]. Exploration combines navigation, analysis, and presentation. The available information about the target system can be seen as an interwoven and multidimensional information space akin to hypermedia. The visualization engine should present suitable views into the entities and relationships space that can be selected, explored, and manipulated by the reverse engineer. For example, reverse engineers should be able to select entities through structured and unstructured searching, pattern matching, and applying filters. Interactive navigation should enable to explore dependencies along organizational axes including dependency graphs, hierarchies, and levels of abstraction. Importantly, it should be possible to navigate from higher-level, graphical entities down to the corresponding source code. For instance, double-clicking on a node that represents a procedure or variable in Rigi opens up a text editor with the corresponding source code. Effective information presentation can be achieved with automatic layouts and visualization techniques. SHriMP [21] uses a nested graph view to present information that is hierarchically structured and supports the layout algorithms Grid, Spring, Radial, Tree, Sugiyama, and TreeMap. A SHriMP fisheye zoom focuses on any node of interest while maintaining pertinent properties such as orthogonality or other spatial relations.

Since most reverse engineering visualizations have to deal with large information spaces, they have to address scalability issues. The rendering speed of a visualization engine is critical for interactive program understanding tasks. On the one hand, visualizations have to cope with limited screen space and, on the other hand, with the limited perception capabilities of the reverse engineer.

Another important feature is support for customization. The CodeCrawler tool offers polymetric views, which utilize the height and width of graph nodes, and width of edges to convey additional (e.g., quality attributes or number of calls) information [16] (cf. Figure 9). A reverse engineer has the freedom to specify which (metric) information is mapped to which visual attribute. On the one hand, customizable views are highly desirable because of the diversity of reverse engineering tasks and individual user preferences. On the other hand, predefined views (e.g., call graphs or subsystem structures) are often important starting points for reverse engineers to derive customized views. For example, CodeCrawler provides a number of predefined views that address several typical comprehension tasks. If the reverse engineer is unfamiliar with the subject system, “first contact” views can provide an initial understanding of its size and complexity. There are usually views to expose aggregation and inheritance hierarchies. Candidate identification views expose

refactoring opportunities or anomalies in a subject system that should be further investigated (e.g., dead code or unused attributes). In SHriMP [21] and CodeCrawler [16], views can be conveniently customized using GUI widgets. In Rigi custom-made views can be codified as scripts to provide additional flexibility and robustness when the subject system or its abstractions evolve [13].

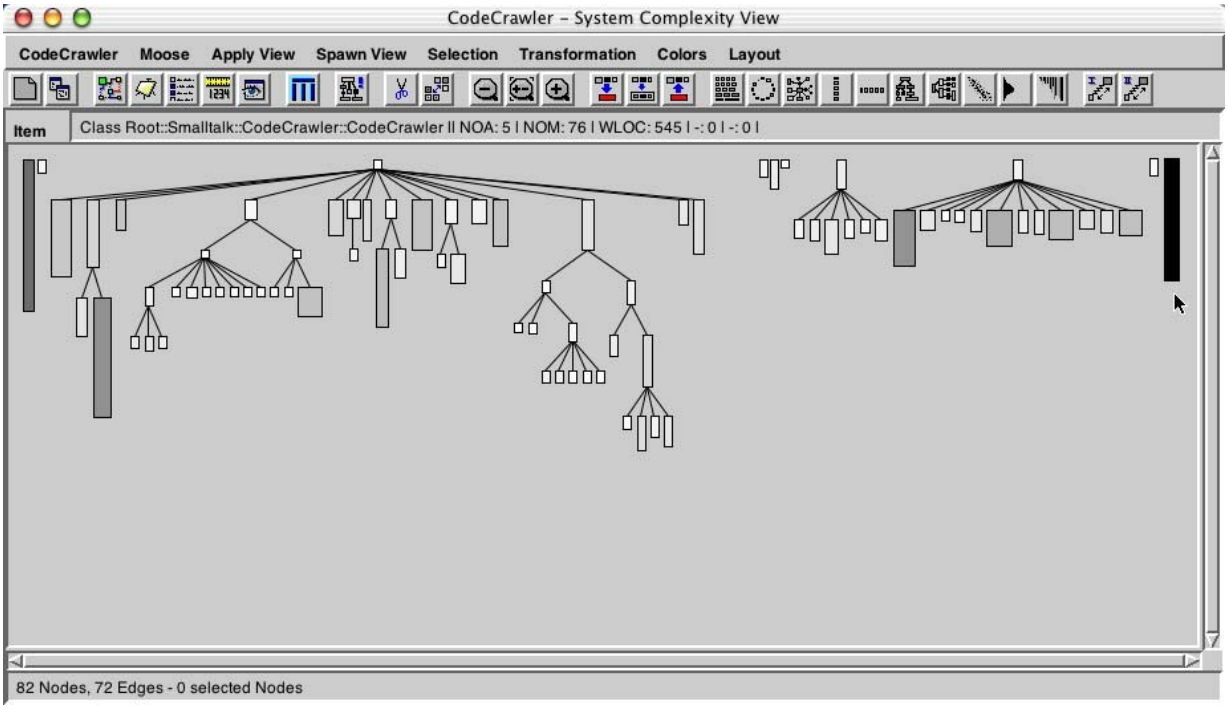


Figure 9: Screenshot of the CodeCrawler tool with a polymetric view showing class hierarchies.

Tree and graph-based visualizations are prevalent and have proven to be both useful and versatile in conveying reverse engineering results in part because they can be intuitively navigated and customized by users. For example, even clone-analysis results can be represented with graphs—nodes represent fragments of source code and edges represent duplication between two fragments. Furthermore, metrics information can be incorporated with polymetric views (e.g., to convey a similarity measure).

Besides graphs, charts and tables, there are also visualizations that employ combinations of text, graphics, audio, and video. In practice, tools often combine different visualization approaches. For example, annotations for entities and relationships typically take the form of text, but can also be in the form of audio and video. The Ciao system uses two main types of visualizations: textual database views (to provide a human-readable output of information stored in a relational repository), and graph views (to render a visual representation of the relational information). Rigi combines graphical visualization with textual reports (e.g., a list of a node's incoming or outgoing arcs) to provide information about the graphs at different levels of detail. In SHriMP, leaf nodes can be opened up to reveal, for example, a text editor (e.g., to view source) or an HTML viewer (e.g., to view Javadoc documentation). While most visualization engines use 2-D graphics, there are some engines that use 3-D graphics (e.g., CodeCity and Plum). Tools such as SHriMP and CodeCity also use animations to provide context and illustrate exploration paths, and to convey system evolution. Generally, reverse engineering visualization tools increasingly incorporate multi-media, rendering, and animation capabilities based on advanced graphics platforms (e.g., OpenGL).

7. Conclusions

The field of software reverse engineering and closely related fields, such as program comprehension or software analysis, have enjoyed many successes over the past two decades—in academia as well as industry. Many theories, methods, tools, and techniques have been developed to ease and facilitate the understanding, maintenance, and reengineering of software-intensive systems. We have become proficient in modeling and extracting artifacts and dependencies, exploring software systems with visual metaphors, and uncovering higher-level representations and knowledge. Over the past two decades reverse engineering researchers have produced over 100 Ph.D. theses—a tremendous success story and a testament to the maturity of this field.

A mature research field can easily fade away due to narrow focus, overgrazing, or lack of impact. The field of reverse engineering has resisted these threats and continues to be an active research field [17]. The success stories of reverse engineering research are reported in the premier software engineering conferences (i.e., ICSE, ICSM, CSMR, FSE, and ESEC) as well as specialized conferences (i.e., WCRE, ICPC (formerly IWPC and WPC), SCAM, PASTE, MSR, VISSOFT, ATEM, IWPSE, and WSE). The vitality of the field is amply exhibited at these more specialized events which shape and focus the field. From the beginning, industrial case studies and legacy subject systems were an essential and integral component of reverse engineering research.

We see the following trends in reverse engineering that promise to assure its continued relevance. Reverse engineering will continue to support program understanding of emerging technologies such as aspects, service-oriented systems, self-managing systems, and cloud computing—today's new technology is tomorrow's legacy in need of reverse engineering. Relatively weakly explored areas of reverse engineering are the domains of distributed systems and multi-threaded programming [3], which are gaining momentum and require novel program comprehension techniques. In recent years, the mining of software repositories (MSR) has emerged as a field of its own with its own conference. MSR is concerned with the extraction, analysis, and linking of information from software development artifacts obtained from mining source control repositories, bug repositories, and deployment logs [5]. This community adds a new dimension to our understanding (e.g., bug prediction or understanding team dynamics) of how software is constructed and evolved.

In contrast to development platforms (e.g., IBM Rational Jazz), most reverse engineering tools make little allowance for distributed and collaborative software development. Most tools assume that program comprehension is performed by a single reverse engineer. Nowadays tools have to support instantaneous collaborations among reverse engineers. Moreover, agile processes interweave forward and reverse engineering activities during software construction (e.g., continuous refactoring). The good news is that reverse engineering technologies that facilitate program understanding are increasingly incorporated into development platforms as aptly demonstrated by the Eclipse open development platform.

To learn more about the exciting research field of reverse engineering, the 2000 [17] and 2007 [2] articles presented at the ICSE Future of Software Engineering track together list over 200 references, which are good starting points to thoroughly explore and appreciate the field of reverse engineering.

Acknowledgments

We are deeply indebted to many colleagues and friends in the software reverse engineering community, who contributed tremendously to our appreciation and understanding of reverse engineering.
The work for this article was funded in part by the National Sciences and Engineering Research Council (NSERC) of Canada (CRDPJ 320529-04 and CRDPJ 356154-07) as well as IBM Corporation and CA Inc. via the CSER Consortium.

References

[1] Chikofsky, E.J., Cross, J.H.: Reverse Engineering and Design Recovery: A Taxonomy, IEEE Software 7(1):13–17 (1990)

[2] Canfora, G., Di Penta, M.: New Frontiers of Reverse Engineering, In: Proceedings Future of Software Engineering (FoSE 2007), pp. 326–341 (2007)

[3] Cornelissen, B., Zaidman, A., van Deursen, A., Moonen, L., Koschke, R.: A Systematic Survey of Program Comprehension through Dynamic Analysis, IEEE Transactions on Software Engineering (TSE), in press (2009)

[4] Davis, K.H., Aiken, P.H.: Data Reverse Engineering: A Historical Survey. In: Proceedings 7th IEEE Working Conference on Reverse Engineering (WCRE 2000), pp. 70–78 (2000)

[5] Hassan, A.E.: The Road Ahead for Mining Software Repositories. In: Müller, H.A., Tilley, S.R., Wong, K. (eds.): Frontiers of Software Maintenance (FoSM 2008), IEEE Computer Society Press, pp. 48–57 (2008)

[6] Holt, R.C.: Grokking Software Architecture. In: Proceedings 15th IEEE Working Conference on Reverse Engineering (WCRE 2008), pp. 5–14 (2008)

[7] Harman, M., Hierons, R.: An Overview of Program Slicing, Software Focus 2(3):85–92 (2001)

[8] Hamou-Lhadj, A., Lethbridge, T.C.: A Survey of Trace Exploration Tools, In: Proceedings IBM/ACM Conference of the Centre for Advanced Studies on Collaborative Research (CASCON 2004), pp. 42–55 (2004)

[9] Harandi, M.T., Ning, J.Q.: Knowledge-Based Program Analysis, IEEE Software 7(1):74 –81 (1990)

[10] Jin, D., Cordy, J.R., Dean, T.R.: Where's the schema? A Taxonomy of Patterns for Software Exchange, In: Proceedings

10th IEEE International Workshop on Program Comprehension (IWPC 2002), pp. 65–74 (2002)

[11] Koschke, R.: Atomic Architectural Component Recovery for Program Understanding and Evolution: Evaluation of Automatic Re-Modularization Techniques and Their Integration in a Semi-Automatic Method, Ph.D. Thesis, University of Stuttgart, Germany (2000)

[12] Kazman, R., Carrière, S.J.: Playing Detective: Reconstructing Software Architecture from Available Evidence, Automated Software Engineering (ASE), 6(2):107–138 (1999)

[13] Kienle, H.M., Müller, H.A.: The Rigi Reverse Engineering Environment, 1st International Workshop on Advanced Software Development Tools and Techniques (WASDeTT 2008), 17 pages (2008). http://www.iam.unibe.ch/~scg/download/wasdett/wasdett2008-paper06.pdf

[14] Kullbach, B., Winter, A.: Querying as an Enabling Technology in Software Reengineering, In: Proceedings 3rd IEEE European Conference on Software Maintenance and Reengineering (CSMR '99), pp. 42–50 (1999)

[15] Kazman, R., Woods, S.G., Carrière, S.J.: Requirements for Integrating Software Architecture and Reengineering Models: CORUM II, In: Proceedings 5th IEEE Working Conference on Reverse Engineering (WCRE '98), pp. 154–163 (1998)

[16] Lanza, M., Ducasse, S.: Polymetric Views—A Lightweight Visual Approach to Reverse Engineering, IEEE Transactions on Software Engineering (TSE), 29(9):782 –795 (2003)

[17] Müller, H.A., Jahnke, J.H., Smith, D.B., Storey, M.-A., Tilley, S.R., Wong, K.: Reverse Engineering: A Roadmap, In A. Finkelstein (ed.): In: Proceedings Conference on The Future of Software Engineering, pp. 49–60 (2000)

[18] Marcus, A., Maletic, J.I.: Recovering Documentation-to-Source-Code Traceability Links Using Latent Semantic Indexing, In: Proceedings 25th ACM/IEEE International Conference on Software Engineering (ICSE 2003), pp. 125–135 (2003)

[19] Murphy, G.C., Notkin, D., Griswold W.G., Lan, E.S.: An Empirical Study of Static Call Graph Extractors, ACM Transactions on Software Engineering and Methodology (TOSEM), 7(2):158–191 (1998)

[20] Storey, M.A.: Theories, Methods and Tools in Program Comprehension: Past, Present and Future, In: Proceedings 13th IEEE International Workshop on Program Comprehension (IWPC 2005), pp. 181–191 (2005)

[21] Storey, M.-A., C. Best, J. Michaud, D. Rayside, M. Litoiu, and M. Musen: SHriMP Views: An Interactive Environment for Information Visualization and Navigation, In: Proceedings International Conference on Human Factors in Computer Systems (CHI 2002), pp. 520–521 (2002)

[22] Tilley, S.R. The Canonical Activities of Reverse Engineering, Annals of Software Engineering, 9(1–4):249–271, Kluwer Academic Publishers (2000)

[23] van Deursen, A., Hofmeister, C., Koschke, R., Moonen, L., Riva, C.: Symphony: View-Driven Software Architecture Reconstruction, In: Proceedings 4th Working IEEE/IFIP Conference on Software Architecture (WICSE 2004), pp. 122–132 (2004)