The Tools Perspective on Software Reverse Engineering: Requirements, Construction, and Evaluation

HOLGER M. KIENLE
Departments of Computer Science
University of Victoria, Canada

HAUSI A. MÜLLER
Department of Computer Science
University of Victoria, Canada

Abstract
Software reverse engineering is a subdiscipline of software engineering, striving to provide support for the comprehension of software systems by creating suitable representations of the system in another form or higher level of abstraction. In order to be effective, reverse engineering needs tool support, which provides functionality to extract low-level facts from the systems, to analyze and generate knowledge about the systems, and to visualize that knowledge so that reverse engineers are able to comprehend the aspects of the system that they are interested in effectively.

This chapter explores the issue of building tools for reverse engineering. Since tools are an important part of conducting research in reverse engineering, it is worthwhile to reflect upon the state of tool building with the goal to advance upon it—and thus to advance reverse engineering research as a whole. We tackle this goal by looking at the issue of tools through a set of lenses. The purpose of each lens is to focus on a critical topic for tool building by surveying the current state-of-the-art and identifying challenges that need to be addressed.

In this chapter we discuss three lenses, namely (1) requirements for reverse engineering tools, (2) construction of reverse engineering tools, and (3) evaluation of reverse engineering tools. The first lens identifies a number of generic quality attributes that reverse engineering tools should strive to meet.
The second lens approaches tools from the observation that since tool building is a key activity in research, it should be conducted in an effective and rather predictable manner. The third lens looks at the role that tools play in supporting the evaluation of reverse engineering research. While each lens looks at the topic from a different perspective, taken together they provide a holistic picture of tool building in the reverse engineering domain.

1. Introduction and Background .............................................. 190
   1.1. Techniques, Processes, and Tools .................................. 192
   1.2. Tool Components .................................................. 194
   1.3. Exploring Tools Through a Set of Lenses ....................... 198
2. Tool Requirements Lens .................................................. 199
   2.1. Scalability .......................................................... 201
   2.2. Interoperability .................................................... 206
   2.3. Customizability ..................................................... 212
   2.4. Usability ............................................................ 221
   2.5. Adoptability ........................................................ 227
   2.6. Requirements of Exchange Formats ............................... 236
   2.7. Discussion .......................................................... 240
3. Tool Construction Lens .................................................... 242
   3.1. Tool Architecture ................................................... 243
   3.2. Component-Based Tool Development .............................. 247
   3.3. Model-Driven Tool Development .................................. 252
   3.4. Discussion .......................................................... 254
4. Tool Evaluation Lens ....................................................... 258
   4.1. Evaluation-Driven Tool Building .................................. 262
   4.2. Theory-Grounded Tool Building ................................... 265
   4.3. Discussion .......................................................... 268
5. Conclusions ................................................................. 271
References ................................................................. 274

1. Introduction and Background

“Week by week his understanding of his world improves, the white spaces on his map filling up with trails and landmarks.”

Hari Kunzru, The Impressionist

Broadly speaking, reverse engineering is the process of extracting know-how or knowledge from a human-made artifact [1]. A human-made artifact refers to an
object that embodies knowledge or know-how that was previously discovered or applied by the artifact’s creator. An alternative definition is provided by the US Supreme court, who defines reverse engineering as “a fair and honest means of starting with the known product and working backwards to divine the process which aided in its development or manufacture” [2].

Reverse engineering has a long-standing tradition in many areas, most notably in traditional manufacturing (e.g., automotive and medical devices [3]), but also in technology-oriented and information-based industries such as semiconductors, digital media, telecommunications, electronics, and computer software. The latter area is the focus of this chapter. In the literature, the term software reverse engineering is used to emphasize that the target is some kind of software (system). Since we are only concerned with software we will simply use reverse engineering when it is clear from the context that we mean software reverse engineering.

In the software domain, Chikofsky and Cross define 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” [4]. Thus, the reverse engineering process typically starts with lower levels of abstraction (e.g., source code of a high-level programming language) to create higher levels of understanding (e.g., UML class or interaction diagrams). Note that this definition emphasizes that reverse engineering of software produces some kind of knowledge or facts about the software. In contrast, in the manufacturing area the goal of reverse engineering is often to come up with a process that allows to duplicate an existing artifact in the absence of technical drawings, computer models, or other kinds of technical documentation [3].

To thwart a possible misconception, the purpose of software reverse engineering rarely is to develop competing software [4, 5]. Reverse engineering of “foreign” software is much more likely for the purpose of security assessments, uncovering cryptographic techniques, and ensuring interoperability. In this respect, there is a significant difference between reverse engineering in the software and manufacturing domains. While it can be very challenging to duplicate a manufactured artifact (e.g., a fighter plane), it is trivial to duplicate software via copying it. The open source community has demonstrated that the functionality of huge software systems can be “duplicated” by implementing the software from scratch. In fact, many software engineers wish that they could do just that—to scrap their legacy systems and rebuild the same functionality with a cleaner design and different technologies. This leads to the most common purpose of software reverse engineering which is also the focus of this chapter, namely to enable software owners to gain a better understanding of their own software assets so that they are in a position to evolve it.

It is safe to assume that reverse engineering has been performed since the very beginning of software in the form of debugging and disassembly. Academic interest
into reverse engineering started around the mid-1980s with dispersed publications appearing in a number of conferences and journals (e.g., see reprints of articles in Ref. [6])—leading to the emergence of a dedicated community and the first Working Conference on Reverse Engineering (WCRE) in 1993. While it is fair to speak of a dedicated community of reverse engineering researchers [7], one should understand that reverse engineering blends into other research areas such as program comprehension (a.k.a. program understanding) [8], reengineering [6], software maintenance and evolution [9, 10], compiler construction (front-end) [11], software visualization [12], software metrics [13], software modeling [14, 15], and support for collaboration [16, 17].

1.1 Techniques, Processes, and Tools

Academic research in reverse engineering over at least two decades has yielded a broad portfolio of different techniques, processes/methods, and tools. Important techniques include representations of call graphs, impact analysis, identification of abstract data types, clustering, clone detection (a.k.a. code duplication), architecture recovery, redocumentation, cliché recognition, and extraction of business rules. The diversity of reverse engineering techniques is also demonstrated by the software artifacts that these techniques are targeting [17]. Traditionally, techniques have focused on code of high-level programming languages such as Cobol and C—and to a lesser degree assembly—as well as databases (so-called data reverse engineering [18]). There are also dynamic analyses that provide tracing information about the running system [19]. More recently, reverse engineering has broadened its range to include web sites, bug tracking and repository information, and higher level documentation in the form of UML diagrams and natural language texts.

A reverse engineering process gives important guidance on issues such as identifying relevant stakeholders for a reverse engineering project, setting project goals, defining workflows, and selecting appropriate tools and techniques. At the micro-level, reverse engineers follow a workflow that is characterized by three distinct activities: extraction, analysis, and synthesis/visualization. To illustrate these activities, we describe how a reverse engineer would obtain a call graph from the subject system [20].

Extraction: A reverse engineering activity starts with extracting facts from a software system’s sources (i.e., the artifacts). For a static call graph, only facts from the source code itself need to be extracted. It is necessary to know the procedures (or methods) as well as the calls made within the procedures. Furthermore, the position of the calls within the source code (e.g., source file name and line number) is often of interest.
Analysis: Certain analyses are performed that use the facts generated in the extraction step. To obtain a call graph, the analysis has to match procedure calls with the corresponding procedure definitions. This matching is not necessarily trivial, for instance, for a call via a function pointer, or a dynamic method call. With this information it is possible to construct a graph structure that represents the call graph.

Synthesis: Results of analyses are presented to the user in an appropriate form. Information typically is presented in a mixture of both textual and graphical form. A call graph is typically rendered with nodes (representing procedures), arcs (representing procedure calls), and node labels (giving the name of the procedure). A reverse engineering tool may show a static rendering of the call graph (e.g., Ciao/CIA [21, 22]) or may offer interactive functionality to explore the graph, for example, via applying layout algorithms (e.g., Rigi [23]).

In the micro-level process, reverse engineers use different comprehension strategies such as bottom-up (i.e., starting from the code and then grouping parts of the program into higher level abstractions), and top-down (i.e., starting with hypotheses driven by knowledge of the program’s domain and then mapping them down to the code) [24]. Some reverse engineers try to understand a program systematically in order to gain a global understanding, while others take an as-needed approach, restricting understanding to the parts related to a certain task [25]. The latter approach can be seen as just-in-time comprehension (JITC) [26, 27]; its concept is nicely illustrated by Holt’s law of maximal ignorance: “Don’t learn more than you need to get the job done” [28].

The micro-level process exclusively focuses on the technical perspective and assumes a single, isolated reverse engineer. This is appropriate for a simple reverse engineering project such as a well-defined technical problem or an academic exemplar. In contrast, the macro-level process aims to be more holistic, addressing not only technical, but also business and policy issues and stakeholders [29]. Thus, the activities of the micro-level are driven by the macro-level process, which sets high-level objectives for the overall reverse engineering effort.

Besides techniques and processes, tools are a crucial result of reverse engineering research. Tools are needed to support and validate novel techniques. Typically, reverse engineering techniques require tool support because performing them manually is impractical. Algorithms embodied in a technique may be too complex or cumbersome to perform manually (e.g., creating a call graph manually via code inspection). Also, whenever the target software system changes, the technique needs to be reapplied (e.g., recreating of a call graph whenever the system changes).

As a result, it is expected in the reverse engineering community that a proposed technique is accompanied by a supporting tool and validation that demonstrates the technique’s feasibility. While the supporting tool is typically a proof-of-concept
prototype rather than an industrial-strength tool, it should meet certain key requirements such as scalability and usability, or other objective evaluation criteria. The tool is then evaluated against the requirements or criteria with the help of empirical studies (Section 4). But tools are not strictly a vehicle to validate techniques—they are more than that because there is a symbiotic relationship between building of tools and exploring of research ideas [30].

On the one hand, the construction of tools is an important part of reverse engineering research. On the other hand, tool construction is neither simple nor cheap to accomplish [31]. Tool building is costly, requiring significant resources. This is especially the case if the tool has to be robust enough to be used in (industrial) user studies. Sometimes a significant part of the resources of an entire research group are devoted to building, evaluating, and improving a tool.

### 1.2 Tool Components

The reverse engineering community has developed many reverse engineering tools—prominent examples include Bauhaus [32, 33], Ciao/CIA [21, 22], Columbus [34], GUPRO [35], Moose [30], Rigi [23], PBS [36], and SHriMP [37]. Importantly, most of these reverse engineering tools have a similar software architecture, consisting of several components with standard functionalities (Fig. 1): extractor, analyzer, visualizer, and repository. The extractor, analyzer, and visualizer components reflect the reverse engineering activities of extraction, analysis, and synthesis, respectively (see Section 1.1). In the following, we give a brief overview of the four tool component types.

![Components of reverse engineering tools](image-url)
1.2.1 Repository

The most central component is the repository. It gets populated with facts extracted from the target software system. Analyses read information from the repository and possibly augment it with further information. Information stored in the repository is presented to the user with visualizers.

Examples of concrete implementations of repositories range from simple text files to commercial databases. Many reverse engineering tools store data in text files and define their own exchange format [38]. For example, the Rigi tool defines the Rigi Standard Format (RSF) [23, 39], which has been adopted by a number of other tools as well. The Moose tool defines the MSE exchange format [40]. Many tool now also support the GXL format [41].

The data in a repository conforms to a schema. The purpose of a schema is to impose certain constraints on otherwise unrestricted (data) structures. An important design consideration for a schema is its granularity—it “has to be detailed enough to provide the information needed and coarse grained enough for comfortable handling” [42].

Schemas are often discussed exclusively in the context of repositories. This is understandable because of the repository’s central role to facilitate data exchange. However, the remaining three component types also adhere to schemas, but this is often not recognized because these schemas are typically implicit and internal. For example, extractors, analyzers, and visualizers often use in-memory data structures (such as abstract syntax trees or control-flow graphs). For data export, these components then have to transform the data from their internal representations in order to conform with the repository’s schema.

1.2.2 Extractors

Extractors populate the repository with facts obtained from the artifacts that make up the subject system. The extractor has to provide all the facts that are of interest to its clients (i.e., subsequent analyses).

Most reverse engineering tools focus on the extraction of static facts from source code. Static extraction techniques use a number of different approaches. Some extractors use compiler-technology to parse the source. Many parsers are in fact based on compiler frontends. For example, Rigi’s C++ parser is built on top of IBM’s Visual Age compiler [43]. However, there are also parsers that have been built from scratch such as the Columbus C++ parser [34].

1 Schemas are also known as meta-models and domain models.
In contrast to parsing, there are lightweight approaches such as lexical extractors, which are based on pattern matching of regular expressions. Lexical approaches are not precise, that is, they can produce fact bases with false positives (i.e., facts that do not exist in the source) and false negatives (i.e., facts that should have been extracted from the source) [44]. On the other hand, they are more flexible than parsers [45]. Typically, lexical extractor tools are agnostic to the target language, and reverse engineers use them to write ad hoc patterns to extract information required for a particular task. Such lightweight approaches are a natural match for JITC.

1.2.3 Analyzers

Analyzers query the repository and use the obtained facts to synthesize useful information for reverse engineering tasks. Reverse engineering research has developed a broad spectrum of automated analyses [46, 47].

Analyses can be distinguished by the kinds of representation that they operate upon: lexical, syntactic, control flow, data flow, and semantic [48]. Call graphs, for instance, can be constructed based on lexical matching of function declarations and calls. However, depending on the target language this approach can produce a significant number of false positives and false negatives. For example, in C a lexical approach cannot distinguish between macros and function calls. To reduce such imprecisions, a syntactic approach can be used. To improve precision of the call targets, pointer analysis could be used, which requires the construction of a flow graph.

Another approach to classify analyses is with the help of dichotomies [49, 50]—important ones in the context of reverse engineering are as follows:

Static versus dynamic: Static analyses produce information that is valid for all possible executions, whereas dynamic analyses results are only valid for a specific program run. To assist the reverse engineer, both types of analyses are useful. There are tools that combine both static and dynamic information such as Shimba [51].

Sound versus unsound: Sound analyses produce results that are guaranteed to hold for all program runs. Unsound analyses make no such guarantees. Thus, dynamic analyses are always unsound. Unsound analyses are rather common in the reverse engineering domain. As explained above, many techniques that construct call graphs are unsound because they make simplifying assumptions, thus yielding a potentially wrong or else incomplete result. Still, even if the result of an analysis is unsound, it may give valuable information to a reverse engineer.

Speed versus precision: Static analyses typically trade speed for precision or vice versa. This trade-off is well exemplified by the field of pointer analysis, which has produced various different analysis techniques [52]. Speed is often a problem for analyses that rely on data-flow information, especially if the target systems has millions of lines of code.
Multilanguage versus single-language: Many analyses are tailored toward a specific programming language (e.g., pointer analysis or clustering for C). However, for a (global) analysis to be truly useful in a heterogeneous environment, language boundaries must be crossed. In fact, many industrial systems are based on multiple languages [53].

1.2.4 Visualizers

For software engineers to make the most effective use of the information that is gathered by extractors and augmented by analyzers, the information has to be presented in a suitable form by software visualizers. Mili and Steiner define software visualization as “a representation of computer programs, associated documentation and data, that enhances, simplifies and clarifies the mental representation the software engineer has of the operation of a computer system” [54]. Since the complexity of software makes it difficult to visualize, it is important that tools choose suitable techniques. Particularly, different kinds of visualizations are more or less effective, depending on the comprehension task [55]. Some information needs to be suitably condensed to give a birds-eye view of the subject system (e.g., call dependencies between files or modules rather than procedures), other information has to be presented in the most suitable representation for human consumption (e.g., call dependencies shown in tabular reports or directed graphs).

Many reverse engineering tools use graphs to convey information [56]. The Rigi environment represents software structures as typed, directed graphs and provides a graph visualizer/editor to manipulate the graphs [23]. Rigi graphs can have different colors (to distinguish node and arc types), but all nodes and arcs have the same shape. Polymetric views utilize height and width of nodes, and width of edges to convey additional (e.g., metric) information [57]. The SHriMP visualization tool has a view that represents graphs with a nested layout [58]. 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) [59].

Approaches also differ on how the visualized information can be manipulated. For example, one can distinguish between different levels of modifiability:

Static information: Static information is read-only such as a textual report or a static picture of a graph or diagram. Viewers for such information can offer navigational aids such as hypermedia to navigate between pieces of information [60].

Generation of views from static information: Even if the underlying static information cannot be modified, viewers can offer customizable views to selectively suppress and highlight information. Views can be created by applying filters or layouts. Rigi, for instance, has filters to suppress the rendering of certain nodes and arcs.
Enhancement of static information: Some approaches do not allow the user to change the underlying information, but to enhance it. For example, a viewer might allow the user to add annotations to read-only entities. Rigi allows the user to group nodes into a parent node, facilitating the construction of a superimposed hierarchical tree-structure on the static node structure.

Modification of information: An editor can allow the user to change and manipulate the underlying information. Note that changes are typically constrained in some way in order to ensure the information’s consistency. For example, a graph editor that allows the user to delete a node, typically also deletes the arcs attached to that node in order to keep the graph meaningful.

1.3 Exploring Tools Through a Set of Lenses

In a sense, when it comes to tools the “rubber” (i.e., research ideas) “hits the road” (i.e., applying the research ideas). The practicality of a proposed technique can be shown by embodying the technique in a tool and by applying the tool to real-world problems. Conversely, if a tool gets adopted and used in industry it is a strong indication of the usefulness of the tool’s techniques and/or process. A tool or prototype is a viable strategy to transition research results from academia to industry.

Given that tools are a crucial part of reverse engineering research that strongly interact with research in reverse engineering techniques and processes, we believe that more emphasis should be directed towards understanding of and improving upon tool building and understanding its overall research impact.

Thus, in this chapter, we specifically focus on reverse engineering tools—as opposed to techniques and processes—and explore their impact on research with the help of three lenses that address tool requirements, tool construction, and tool evaluation (Table I). Each lens looks at the topic from a different perspective. Taken together, the lenses provide a holistic picture. For each lens, we describe its:

—purpose and raison d’être,
—historical importance,
—current state-of-the-art,
—future directions.

The first lens, tool requirements, identifies a number of generic quality attributes that reverse engineering tools should strive to meet. The tool construction lens approaches tools from the observation that since tool building is a key activity in

---

2 The idea to approach this topic through several lenses was inspired by the metatriangulation approach as described by Jaspersom et al. [61].
The tool evaluation lens discusses how tools support the evaluation of research. In this chapter, we put an emphasis on the first lens, exploring tool requirements in greater detail with the help of an in-depth literature survey. The lenses on tool construction and evaluation are primarily guided by our own experiences and observations. Even though we address the lenses in isolation for clarity, one should be aware that there are interdependencies among them. Where needed, we point out how lenses influence each other.

The rest of this chapter is organized as follows. In Sections 2, 3, and 4, we discuss each of the three tool lenses on requirements, construction, and evaluation, respectively. Section 5 concludes and identifies key issues to advance research in reverse engineering.

### 2. Tool Requirements Lens

“Tool research should not be entirely focused on new paradigms, but should address real user needs and expectations.”

Kenny Wong [62]

In this section, we discuss requirements—mostly quality attributes—of reverse engineering tools. In order to ensure an objective coverage of the requirements, we have conducted an extensive review of the reverse engineering literature. To our knowledge, this is the first attempt of a comprehensive requirements survey in this

---

**Table I**

<table>
<thead>
<tr>
<th>Lens</th>
<th>Explanation</th>
<th>Key topics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tool requirements</td>
<td>What are the requirements for useful and usable reverse engineering tools?</td>
<td>Quality attributes (scalability, interoperability, customizability, usability, adoptability), functional requirements</td>
</tr>
<tr>
<td>Tool construction</td>
<td>How can reverse engineering tools be constructed in an effective and efficient manner?</td>
<td>Component-based tool building, model-based tool building, tool-building process</td>
</tr>
<tr>
<td>Tool evaluation</td>
<td>How to evaluate and compare reverse engineering tools? What theories should reverse engineering embrace?</td>
<td>Empirical research, evaluation-driven tool building, theory-grounded tool building</td>
</tr>
</tbody>
</table>
domain. We mainly focus on quality attributes when discussing tool requirements because they equally apply to a broad range of reverse engineering tools with varying functionalities. The identified requirements are useful to communicate assumptions about tool building in the reverse engineering domain. A requirement that has been reported (independently) by several sources is a good indication that reverse engineering tools should meet this requirement in order to be useful and fulfill the expectations of users.

Bass et al. state, “each domain has its own requirements for availability, modifiability, performance, security, or usability” [63]. This is an important observation because it shows that the particular requirements of a certain domain can be better understood by starting from generic, domain-neutral requirements. However, each domain instantiates this generic, high-level requirement differently, depending on the particular domain’s characteristics. Thus, it is insufficient to simply state the generic requirements for a domain without further elaborating on them—but that is what most reverse engineering research does. Furthermore, one cannot expect to meet all requirements equally well. In the words of Bass et al., “no quality can be maximized in a system without sacrificing some other quality or qualities” [64, p. 75]. Following this line of thought, a reverse engineering tool or development approach that is missing or lacking in some of the requirements identified in the following subsections can still be satisfactory. However, there needs to be a conscious decision on requirements that is informed by some kind of trade-off analysis. Again, currently research is lacking in this respect.

Even though we chose the term requirements, it is not meant to be interpreted in an overly restrictive sense. In the words of Carney and Wallnau, the term requirements “has connotations that are often overly restrictive” and “the terms ‘preference’ and ‘desire’ might be more accurate than ‘requirement.’” [65]. Still, since requirements is firmly established in the literature we will stick to it.

Several researchers have discussed requirements of tools in some detail. In his dissertation, Wong has distilled 23 high-level and 13 data requirements for software understanding and reverse engineering tools [62]. Tichelaar discusses six requirements for reengineering environments [66]. Some researchers discuss requirements in the context of a certain kind of tool. For instance, Hamou-Lhadj et al. discuss requirements for trace analysis tools [67], and van Deursen and Kuipers state requirements for document generators [68]. Hainaut et al. analyze requirements for database reverse engineering environments [18]. Other researchers address requirements of specific tool functionalities. Koschke et al. give 14 requirements for intermediate representations in the context of reverse engineering [69], and Ducasse and Tichelaar discuss requirements of exchange formats and schemas [70].

Tool requirements are also exposed by the criteria used in tool assessments, comparisons, and surveys. For example, there are comparisons of the features of
reverse engineering tools [71–75], fact extractors [76, 77], design recovery [78], and software visualizers for static [79, 80] and dynamic code structure [19, 81], and repository-based data [82]. Some of these comparisons are based on the experiences of the researchers assessing the tool(s), while others are based on controlled user studies, structured demonstrations, or questionnaires.

In the remainder of this section, we first discuss the identified quality attributes (i.e., scalability, interoperability, customizability, usability, and adoptability) and then give functional requirements for exchange formats; more details can be found in the first author’s dissertation [83]. When discussing quality attributes we first give examples of researchers that have postulated the requirement for reverse engineering tools as a whole. Then, where applicable, we address the implications that a certain requirement has on tool components (i.e., repository, extractor, analyzer, and visualizer), and discuss techniques that enable tools to meet these requirements.

2.1 Scalability

“Software developers frequently confront issues of bigness, a.k.a. scale. A harsh criticism of a solution to a software problem is the comment, ‘but it doesn’t scale.’”

David West [84]

It is important to realize that reverse engineering tools might be used on subject systems of significant size. For example, one academic reverse engineering tool has been used on Microsoft Excel, which was reported to have 1.2 million lines of C code at the time [85]. A survey among users of software visualization tools has found that there is equal weight on the visualization of small, medium, and large target systems [79]. One-third of the visualized systems were large (i.e., more than one million LOC), one-third were medium (i.e., between one million and 100,000 LOC), and one-third were small (i.e., less than 100,000 LOC). Bellay and Gall have evaluated four reverse engineering tools (Software Refinery, Imagix 4D, Rigi, and SNiFF+) using a 150,000 LOC embedded software system programmed in C and assembly as a case study [72]. Even though this system is well below a million lines, they conclude that “many shortcomings of reverse engineering tools are due to the size of the case study system.” Baxter et al., discussing the requirements that the Design Maintenance System (DMS) has to meet, make aware of the relationship between system size and processing time [86]:

“A fundamental problem is the sheer size of large software systems. DMS is capable of reading literally tens of thousands of source files into a single session to enable analysis
across entire systems. ... Size translates into computational costs: 1 microsecond per source line means 2.5 seconds of CPU to do anything to 2.5 million lines.”

Whereas some tool implementations handle only limited input, serving often as a proof-of-concept prototype, realistic industrial-strength tools have to handle large target systems. Favre et al. state that “very often, a large number of unexpected problems are discovered when applying good tools at a large scale. This includes not only the size of the software but also the size of the company” [87].

Scalability as a general requirement for reverse engineering tools is discussed by a number of researchers, for instance:

—Brown states for his CodeNavigator tool that it has been designed to “provide information about large-scale software systems” [88]. Furthermore, addressing program understanding tools such as CodeNavigator in general, he observes: “to be effective, they must be able to handle systems of significant size” [88].

—In the context of legacy systems (which are typically large, highly complex, and mission-critical), Wong gives the following tool requirement: “Handle the scale, complexity, and diversity of large software systems” [62].

—Based on their experiences with the Moose reverse engineering environment, Ducasse et al. state that “it should be possible to handle a legacy system of any size without incurring long latency times” [89].

—Lethbridge and Anquetil have developed a list of key requirement for software exploration tools that support JITC. Among the requirements, they state that the system should “be able to automatically process a body of source code of very large size, that is, consisting of at least several million lines of code” [90].

—In his dissertation, Tilley says “it is essential that any approach to reverse engineering be applicable to large systems. For example, effective approaches to program understanding must be applicable to huge, multimillion line software systems” [91].

2.1.1 Repositories

The amount of information that needs to be stored in the repository can affect scalability. Cox and Clark say “a repository is scalable when there are no restricting limitations on the amount of extracted information or code that is stored” [93]. The amount of extracted information can be quite large. For example, a GXL file generated with the Columbus tool to represent the Mozilla 1.6 Web browser has a

---

3 This is especially the case for trace data of dynamic analyses, because a small program can, in principle, generate an infinite amount of trace data [93].
size of about 3.5 GB and contains about 4.5 million nodes [94]. The Datrix representation of Vim 6.2 (220,000 LOC) has 3,008,664 relations and 1,852,326 entities [95]. Reverse engineering environments that use exchange formats as their repository should ensure that it “works for multimillion lines of code (e.g., 3–10 MLOC)” [41]. Similarly, Wong requires for a scalable data format to “handle multimillion line software systems” [62]. St-Denis et al. also list scalability among their requirements for model interchange formats (“can be used for real-life, e.g., large-scale applications”) [96]. They discuss the following factors that may affect scalability: “the compressibility of the model interchange object, the possibility of exchanging partial or differential models and the possibility of using references (e.g., hyperlinks) to information instead of the information itself.” Another factor is incremental construction and loading of information to achieve resource optimization [21, 70]. Hamou-Lhadj et al., discussing the requirements of trace exploration tools, state that input/output performance is critical since traces can become very large and users do not tolerate systems with a poor response time [67]. They also note that “a general XML format, such as GXL, often requires more processing than a tuned special format. Performance of parsing XML poses an obstacle, especially for large data sets.”

Analogous to repository scalability, there is a scalability requirement for the in-memory data structures of reverse engineering components. Ducasse and Tichelaar state that “the greater the level of detail, the higher the memory consumption and load time of information from databases or files and the slower the response times of tools that use the information” [70]. For example, the internal representation of Software Refinery’s C extractor is about 35 times the size of the source code [97].

2.1.2 Extractors

Since the extractor has to read in and process all relevant sources of the target system, scalability is an important concern. In the context of web service mining, Sneed says that tools “have to be very fast, since they will be scanning through several million lines of code to identify which portions of that code are potential web services” [98]. A related scalability concern is the question “how many models do you need to extract?” since each model might require a unique extractor [70].

As already mentioned in Section 1.2, there is a broad spectrum of extractor techniques with different performance trade-offs. For example, van Deursen and Kuipers observe that “lexical analysis is very efficient. This allows us to analyze large numbers of files in a short time, and also allows us to experiment with different lexical patterns: If a pattern does not yield the correct answer, the analysis can be
easily changed and rerun” [68]. Ferenc et al., discussing the Columbus tool, state “parsing speed” as a requirement for C++ extractors [34]. Bellay and Gall also give “parse speed” among their reverse engineering tool evaluation criteria [72]. They also give incremental parsing as a criterion, which “provides the ability to parse only the parts that changed, thereby reducing the parse time.” For performance reasons, extractors are often divided into scanner and parser [86].

2.1.3 Analyses

Researchers seem to discuss the scalability or performance of analyses only if there is a potential problem with the run-time of an analysis. For batch-style analyses, there is a problem if the analysis cannot be executed in a longer time-frame such as a nightly run. In contrast, an analysis’ run-time of an interactive tool should be almost instantaneous in the best case. Compared to the other tool components, researchers have stated few general requirements for analyses. An explanation might be that the scalability of analyses is mainly dependent on the time complexity of particular algorithms.

Moise and Wong discuss their experiences with the Rigi tool in an industrial case study. They have developed three specific analyses (written in Tcl) for clustering the subject system according to different criteria. They report for their analyses that “performance is becoming a serious issue with decomposition times running potentially into hours” [99]. Researchers have also discussed scalability and performance issues in the context of trace data compression [19, 67].

An important technique to make analyses more scalable is to “support incremental analyses” [62]. Nierstrasz et al. [30] state for their tool that

“one key technique that allows Moose to deal with large volumes of data is lazy evaluation combined with caching. For example, although Moose provides for an extensive set of metrics, not all of them are needed for all analyses. Even the relevant metrics are typically not needed for all entities in the system. Thus, instead of computing all metrics for all entities at once, our infrastructure allows for lazy computation of the metrics.”

Flexible analyses can trade precision for scalability. For example, Fiutem et al. state for their Architecture Recovery Tool (ART): “To achieve scalability to large industrial size systems, special attention was also devoted to the speed of convergence of the iterative fixpoint method by conceiving a flexible analyzer that allows fine tuning of the trade-off between execution time performance and flow information precision” [100].
2.1.4 Visualizers

A visualizer’s rendering speed has to scale up to large amounts of data. This is especially the case for interactive systems that allow direct manipulation of graphs or diagrams. In a user study of visualization tools “users stopped the generation of call graphs because they felt it was taking too long” [101]. Storey et al., reporting on a user study with the SHriMP visualizer, found the following: “The single most important problem with SHriMP views was the slow response of the interface. Since SHriMP views are based on direct manipulation, users expecting immediacy were disturbed by the slow response” [102]. Czeranski et al. made the experience that “the Bauhaus GUI, which is based on Rigi, has a few unpleasant properties. It is relatively slow, which can cause noticeable waiting periods for large graphs, and hence sometimes disrupts the fluent use of the tool” [103]. There is also the issue of lacking performance: “For scalability we must consider if the tool supports large software projects. If the technique does not appear to scale, it may be the implementation which does not scale rather than the technique” [82].

Armstrong and Trudeau state in their tool evaluation that “fast graph loading and drawing is essential to the usability of any visualization tool” [74]. Bellay and Gall use “speed of generation” of textual and graphical reports as an assessment criterion and experienced that “graphical views often need an unacceptable amount of time to be generated because of the layout algorithms” [72]. Contrary to Bellay and Gall’s findings, Moise and Wong note that their “case study showed that Rigi can deal with large graphs” [99]. In their survey, Bassil and Keller include “tool performance (execution speed of the tool)” as a practical aspect of software visualization tools [79, p. 5].

2.1.5 Discussion

Most researchers discuss scalability in the context of computational performance and efficiency. However, it is typically discussed without giving explicit or quantitative metrics to measure them. The following is a fairly typical example: “The overall repository software is responsive and quite usable. However, in a few places the performance is slower than we would have liked. For example, large pick lists (> 500 elements) take a few moments to load” [104].

Considering the large number of publications about reverse engineering tools, there is a surprising low number of more or less quantitative measures about tools’ performance such as the following ones:

—“CodeCrawler can visualize at this time ca. 1 million entities. Note that we keep all the entities in memory” [57].
—ISVis can analyze “program event traces numbering over 1,000,000 events” [92].
—“Mozilla compiled on our Linux system 4 in about 35 minutes, while the Acacia extraction took three and a half hours and the translation into TA took another three hours” [105].

Furthermore, as the above examples suggest, each researcher reports measurements based on different criteria and metrics; even tool evaluations tend to not use objective measurements for comparisons among tools [72, 74].

It appears that researchers are also neglecting—or purposely withholding—performance measures. For example, in the domain of trace exploration tools, Hamou-Lhadj and Lethbridge made the following observation [19]:

“In order to visualize and analyze large program executions, an efficient representation of the event space is needed. Unfortunately, most of the tools mentioned above do not even discuss this aspect, which makes us have doubts regarding their scalability. It is also important to note that most of the experiments that are conducted by the authors of these tools are based on very small execution traces.”

There is an increased awareness in the software engineering community that approaches need to scale to industrial demands [106]. As a next step, the development of accepted criteria for the evaluation of tools such as benchmarks [77, 107] would be helpful to make performance and scalability measurements more meaningful.

### 2.2 Interoperability

“Building tools is expensive, in terms of both time and resources. An infrastructure for tool interoperability allows tool designers and users to re-purpose tools, which helps to amortize this cost.”

Ferenc et al. [108]

Interoperability is the “ability of a collection of communicating entities to (a) share specified information and (b) operate on that information according to an agreed operational semantics” [109]. In other words, tools that interoperate are able “to pass information and control among themselves” [110].

In a small survey about negative aspects of reverse engineering tools, 6 software engineers out of 19 (31%) complained that tools are not integrated and/or are incompatible with each other [90]—this was also the most frequent complaint!

Reverse engineering researchers have recognized the need for interoperability. Woods et al. [111] observe that

—“many tools have been written with closed architectures that hide useful internal products.”
“many external products are not produced in a useful form for further computation.” They further conclude that “the bottom-line is that existing program understanding tools and environments have not been designed to interoperate.”

Making tools interoperable yields a number of potential benefits. Interoperability enables code reuse in general because it becomes easier for a tool to utilize the functionalities of another one. As a result, reusing of existing functionality can “prevent repetitive ‘wheel-creation’” [112]. Zelkowitz and Cuthill view interoperability as a way to improve automation in software engineering: “Tool integration . . . is crucial to effectively provide automated support for activities. In order to automate activities with a tool set, there must be a seamless way to pass information and control among the tools” [113]. Furthermore, interoperability reduces the time and effort to (opportunistically) assemble a tool set that supports a particular reverse engineering task or process, because “no one tool performs well in all tasks” [81]. For instance, the developers of the Dali tool say, that “one of our emphases has been to provide an open, lightweight environment so that tools can be integrated opportunistically” [114].

A number of researchers address interoperability as a general tool requirement; for instance:

—Interoperability is among the tool requirements given by Lethbridge and Anquetil. They require from tools to “wherever possible, be able to interoperate with other software engineering tools” [90].

—Tichelaar, discussing requirements for reengineering environments, states that “a reengineering effort is typically a cooperation of a group of specialized tools. Therefore, a reengineering environment needs to be able to integrate with external tools, either by exchanging information or ideally by supporting run-time integration” [66]. Similarly, Ducasse et al. say, “the environment should be able to operate with external tools like graph drawing tools, diagrammers (e.g., Rational Rose) and parsers” [89].

—Wong addresses interoperability in the context of tool integration [62]: “Tool integration is necessary to combine diverse techniques effectively to meet software understanding needs.”

—Interoperability is among the 12 requirements that Hainaut et al. identify for tools that aid in the reverse engineering of database systems: “A CARE tool must easily communicate with the other development tools (e.g., via integration hooks or communications with a common repository)” [18].

—The designers of the Augur tool say, “we would like Augur to be broadly usable in real engineering practice. This means that it must be interoperable with a range of existing tools and infrastructures,” and further “Augur’s design emphasizes interoperability and extensibility so that it may be incorporated into existing development efforts without significant overhead” [115].
For tools to interoperate, they have to agree on a suitable interoperability mechanism in some form or another. As a consequence, research papers often directly address the question of *how* to achieve interoperability, without explicit stating it as a requirement first.

### 2.2.1 Techniques

One ambitious approach that has been proposed to achieve interoperability among tools are the Common Object-based Reengineering Unified Model (CORUM) [111] and CORUM II [112] frameworks. These proposals strive to provide a common framework, or middleware architecture, for reverse engineering tools, encompassing standard APIs and schemas for ASTs, CFGs, call graphs, symbol tables, metrics information, execution traces, and so on.

A less ambitious approach compared to the CORUM frameworks is the community’s efforts to define a common exchange format. Exchange formats enable interoperability between tools. A particular tool can utilize an exchange format to make information available to another tool. Panas et al. say “in order to have a properly working combination of two or more tools that have been developed independently, they must be able to exchange information. For this purpose we need a common data representation that is general enough to support a variety of data” [116]. Exchange formats use simple files as a (temporary) repository for information transfer. In this case, the coupling between tools that exchange information is loose; an information producer need not to know its information consumers. Examples of exchange formats in the reverse engineering domain are RSF, TA, GraX, CDIF, and GXL; there are also a number of general-purpose data exchange and encoding formats such as ASN.1, SGML, and XML [38].

A file-based exchange format is a rather primitive form of data interoperability because there is no coordination mechanism for concurrent access. Thus, it is the responsibility of the tool user to assure data-consistency and to initiate data transfer from one tool to another. An example of a more sophisticated solution is a repository common to all tools (e.g., in the form of a database management system) such as proposed by the CORUM framework. Whereas a central repository has many benefits (e.g., data consistency), it is also a heavyweight solution. Brown cautions, “the common data repository is often a large, complex resource that must be controlled, managed, and maintained. This can occupy a great deal of time, money, and effort” [117]. Similarly, Wong concludes from the RevEngE project, which used the object-oriented Telos software repository, “there are significant difficulties in using and maintaining advanced integration technologies” [62].
2.2.2 Schema

Exchange formats and common repositories are effective at communicating the structure (or syntax) of data. However, even if tools are able to read the data, it is of little use if they do not know how to interpret it. Schemas are a vehicle to convey semantic information about the data (i.e., its meaning and use). Interoperability among tools is much more effective if they agree on a certain schema [118]. Godfrey puts it this way: “we feel that the particular syntax to define an exchange format is a small issue . . . We consider the semantic model (design of the schemas) to be the most important issue” [105]. Whereas syntax is domain-neutral, the schema models a particular domain or reflects an intended use. Thus, a single schema will not suffice. For example, there are schemas with different granularities to represent source code: fine-grained (e.g., a detailed C++ AST [108]), coarse-grained (e.g., the PBS high-level schema to model abstract architectures [119]), and in between (e.g., the Dagstuhl Middle Model [120]). Researchers try to establish standard schemas for other domains as well, for instance, execution traces: “There is also a need for a common [schema] for representing the execution traces of object-oriented systems in order to permit interoperability among tools” [19]. In practice, the diversity of tools makes it difficult to agree on schemas. Moise and Wong make the following observation:

“Often, an existing schema may not fit as-is to the particular software being analyzed or the tools being used. Consequently, schema reuse is not a simple task, and a proliferation of new schemas has resulted” [118].

All schemas have in common that they have weak semantics [121], that is, meaning is derived from the names of schema entities and possibly informal documentation. For example, an entity called line number is presumably used on a source code fragment that exists at the given line number in a particular file. However, even if this assumption is correct, it is still not clear if line numbers are counted starting from zero or one, if the line number applies to raw or preprocessed source, if the line number denotes where the fragment begins or ends, and so on.

2.2.3 API

Exchange formats specify the structure of the data and how it is stored on disk. However, how to actually read/write the exchange format and how to represent it in memory is not part of its specification [122]. Thus, tools often implement their own readers and writers. These readers and writers have their own proprietary interface, reflecting the specific needs of a particular tool. Furthermore, the features of the programming language influence the nature and functionality of the API [40, 122].
As a consequence, tools rarely can share an interface and its implementation. Interoperability can be achieved if tools agree on a standardized API to read, write, and manipulate data. A popular example of APIs that enable interoperability for relational data is Open Database Connectivity (ODBC) and Java Database Connectivity (JDBC). The GSEE software exploration tool supports JDBC for data import [123].

Dedicated frameworks for reverse engineering offer an infrastructure for tool integration via common data or control flows. Tools can implement certain interfaces to plug into the infrastructure. The CORUM frameworks are an effort to define common APIs for the reverse engineering domain. However, these efforts have not caught on in the reverse engineering community. Generally, the more sophisticated the interoperability mechanism, the more standardization and agreement among tools is necessary. Jin observes, “although the use of APIs significantly improves the speed and ease of interaction among tools, they still need to know how they can interact with each other. A tool must be aware of the requests it can make of another tool it interfaces with” [124]. This form of tool interaction can be achieved by extending the API with message passing mechanisms based on a message bus (e.g., ToolBus), or point-to-point connections.

One can attempt to draw an analogy between exchange formats (e.g., RSF) and APIs (e.g., ODBC) on how they achieve interoperability. Both abstract from the tools’ execution environments. For example, RSF is stored in ASCII files, abstracting away different file systems. Similarly, ODBC abstracts from a concrete relational database. All data adheres to the same structure, or syntax: With RSF, data is represented in tuples, in ODBC data is represented in tables. The underlying model in RSF is a typed, directed, attributed graph; in ODBC it is relational algebra. Both approaches allow schema introspection: RSF has an (optional) schema description file; ODBC has catalog functions.

2.2.4 Discussion

Interoperability and integration of tools has been extensively addressed by researchers in the context of CASE tools and software development environments. Meyers identifies a number of requirements for tool integration: (1) it should be easy to write new tools for use in a development environment, (2) a new tool should be easy to add to the environment without the need to modify other existing tools that are already part of the environment, and (3) there should be no need for different tools to perform the same task (e.g., no more than one parser for a particular language) [125]. He also discusses several approaches to system integration that have been already discussed here: shared file system (i.e., tools exchange data based
on exchange formats), selective broadcasting (i.e., message passing among tools), simple databases (i.e., tools use a common database as repository), and canonical representations (i.e., tools share a common schema).

One approach pursued by the reverse engineering community to achieve interoperability is to agree on a common exchange format. This is exemplified by the thrust to establish GXL. There are also concrete proposals for schemas in a number of areas (e.g., C++ ASTs, mid-level architectural information, and trace extraction), but no proposal has achieved widespread use yet. The community has also considered more ambitious interoperability mechanisms such as a common repository and APIs. For instance, Sim acknowledges the usefulness of a standard exchange format, but argues to move toward a common API:

“Data interchange in the form of a standard exchange format (SEF) is only a first step towards tool interoperability. Inter-tool communication using files is slow and cumbersome; a better approach would be an application program interface, or API, that allowed tools to communicate with each other directly.... Such an API is a logical next step that builds on the current drive towards a SEF” [126].

However, there is no indication that the reverse engineering community is devoting significant effort to realize this proposal. This is perhaps not surprising, considering that discussion on standard schemas has just begun. In a sense, agreement on an API is comparable to a simultaneous agreement of a standard exchange format along with a number of the most important schemas. Furthermore, exchange formats have their own benefits and it is not clear that APIs are necessarily a move in the right direction.

There is another approach to interoperability that has been mostly ignored so far by the reverse engineering community, namely service-oriented architectures (SOAs). SOA aims to integrate heterogeneous software systems with the use of middleware and services. Software systems expose their functionalities as services, communicating with each other via some kind of middleware. Reverse engineering tools could offer functionalities as web services, and thus allow other tools to discover and call their services in a standardized form. Tool services would allow the development of new reverse engineering functionality via service composition, possibly even on demand. To our knowledge work in this direction is limited so far. Ghezzi and Gall are proposing an approach based on web services [127]. They envision software analysis web services (which are wrappers of already existing tools) that are registered in an analysis catalog (which is organized with the help of a taxonomy). There is an analysis broker that enables users to manage the catalog and to compose services. To facilitate interoperability they propose to use ontologies encoded in OWL to represent inputs to and results of tools.
2.3 Customizability

“It has been repeatedly shown that no matter how much designers and programmers try to anticipate and provide for users’ needs, the effort will always fall short.”

Scott Tilley [91]

Customizability is another important requirement for reverse engineering tools. As the introductory quote by Tilley suggests, reverse engineering activities are quite diverse and depend on many factors. As a result, reverse engineers have to continuously adapt their tools to meet changing needs. Thus, it is insufficient for a reverse engineering tool to be general (i.e., it can be used without change in different contexts), it has to be flexible as well (i.e., it can be easily adapted to a new context) [128].

Michaud looks at software customization by asking what is customized, how is it customized, who performs it, and when does it occur [129]. In terms of what is customized one can distinguish between data, presentation, and control customization. In terms of how software is customized one can distinguish between source code customization and other forms that do not require to write code such as option screens, wizards, and configuration files. Customizations can be performed by the tool designers, a dedicated person who customizes the tool to suit a group of users, and the tool users themselves. Customization mechanisms are put in place during tool development by the original tool designers. These mechanisms are then used when installing the tool and during run-time.

In the context of data reverse engineering, Davis observes that “tools need [to be] customized to each project” [130]. Customizability also enables to meet needs that cannot be foreseen by tool developers, for instance, if a tool is applied in a new context or domain. Best states, “if the designer does not create an architecture that lends to extensibility, opportunities to use the tool in other domains can be missed” [131]. Tilley characterizes such customizable tools as domain-retargetable [91]. Conversely, a tool that is not customizable is probably too rigid to meet the changing needs of reverse engineers except in a few well-understood circumstances. Markosian et al. say for reengineering tools, “in our experience, lack of customizability is the single most common limiting factor in using tools for software analysis and transformation” [132].

Many tool developers, including commercial ones that produce mass-market software, see customizability as an important requirement to satisfy their customers.

---

4 Since most researchers do not distinguish between customizability and extensibility, we use both terms interchangeably. A concept related to customizability is end-user programmability, which allows users of an application to tailor it programmatically to their needs [92].
In the following, we give examples of researchers that discuss extensibility as a general tool requirement:

—Buss and Henshaw discuss their experiences with the reverse engineering of IBM’s SQL/DS system. Among the lessons learned, they state that “any reverse engineering toolkit must be extensible to meet your problem domain needs,” and “since reverse engineering is an open-ended, context-dependent activity, it is imperative that its toolkits be similarly open-ended, flexible, extensible, and versatile” [133].

—In his dissertation about domain-retargetable reverse engineering, Tilley states, “a successful reverse engineering environment should provide a mechanism through which users can extend the system’s functionality” [91].

—Bellay and Gall’s evaluation framework for reverse engineering tools contains a toolset extensibility criterion: “Tool extensibility is an important feature of many types of tools. This is also the case for reverse engineering tools, in which additional functionality often needs to be integrated to meet the specific constraints of a reverse engineering activity” [72]. The framework further distinguishes between extensibility of parsers, user interfaces, and tool functionality.

—Hainaut et al. give “functional extensibility” for CASE tools as a requirement, motivating it with “no CASE tool can satisfy the needs of all users in any possible situation” [18].

—Reiss has developed a tool, CLIME, to aid software maintenance by formulating constraints on development artifacts such as source code, UML design diagrams, comments, and test cases. Such a tool should be “adaptable to new design techniques and approaches” and as a result “must be open and extensible” [134].

Typically, tools enable customization of their functionalities via configuration files, built-in scripting support, or programmable interfaces [129].

### 2.3.1 Repositories

A repository consists of a schema and the data stored according to the schema. Each repository provides a rudimentary form of extensibility, because the data that is stored in the repository is not fixed, but customized by the applications that uses the repository. Thus, a more meaningful form of repository customizability is to look at the customizability of a repository’s schema. In fact, customizability of a reverse engineering tool is often realized with an extensible schema. The developers of the Moose reengineering tool state, “the extensibility of Moose is inherent to the extensibility of its [schema]” [89].

A number of researchers agree that an exchange format “should be extensible, allowing users to define new schemas for the facts stored in the format as needed” [119, 126]. More specifically, Ducasse et al. require that “an environment for
reverse engineering and reengineering should be extensible in many aspects: ... the [schema] should be able to represent and manipulate entities other than the ones directly extracted from the source code (e.g., measurements, associations, relationships, etc.)” [89]. Among the requirements that Ferenc et al. have identified for a C++ schema is the need for schemas to “be modular and easily extensible” [34]. Similarly, Riva states that an exchange format “should be easy to extend by the users themselves without the knowledge of complicated procedures” [135]. One of Wong’s requirements for a reverse engineering repository is to “support dynamically evolvable schemas” [62]. He further elaborates, “this flexibility to evolve schemas dynamically and incrementally is especially important in software understanding. New needs and concepts often arise over time.” Event traces are an example of data obtained with a dynamic analysis. The ISVis tools supports the extension of trace data with new event types without having to change the tool itself: “As far as ISVis is concerned, events have types, and the exact nature of the type is unimportant to the pattern matching ISVis provides” [92].

One can distinguish between the following forms of schema extensibility:

**Fixed:** Fixed schemas model a certain domain, which is not expected to change. For example, the Bauhaus Resource Graph models the design level of procedural languages [33]. A number of analyses and visualizations have been implemented based on this schema. Consequently, changes in the schema are expected to cause changes in the Bauhaus tool.

**Ad hoc:** This approach allows to add information in an unstructured way, typically in the form of annotations or extensions (which can range from free comments in natural language to formal assertions) [69]. For example, software engineering environments allow tool-specific decorations of abstract syntax trees [125], and UML allows to attach string tags to entities. Tool is then expected to ignore annotations that they do not know.

**Domain-extensible:** Domain-extensible schemas provide a core schema describing certain common domain features [136]. The core schema can then be extended. The FAMIX schema of the Moose tool provides a language-independent representation of object-oriented features, the core model, which can be extended with language-specific information [89]. Similarly to FAMIX, the Dagstuhl Middle Model allows extensibility via subclassing [120].

**Domain-retargetable:** Domain-retargetable schemas are domain-neutral per se, allowing the specification of any domain. For instance, the TA exchange format has been used to define schemas for architecture recovery at various levels of abstraction [90, 119, 137]. RSF has schemas to represent higher level software architectures, C++, Java, web sites, C, COBOL, PL/AS, LaTeX, and so on. The Rigi tool is a generic graph viewer that can visualize data that adheres to any RSF schema.
2.3.2 Extractors

There are few customizable extractors for reverse engineering. An early example of a customizable parser is Software Refinery’s DIALECT. Newcomb and Markosian report their experiences with the migration of a COBOL payroll system [138]. They give a simple customization example of DIALECT:

> “the OS/VS compiler used for the payroll system allowed some periods at the end of sentences to be omitted; this syntax had not previously been handled by REFINE/COBOL.”

TXL is a source code analysis and transformation tool that allows grammar customizations via so-called agile parsing [139]. In TXL, there is a base grammar for the input language that can be customized with grammar overrides (e.g., to support a particular dialect or analysis task). A typical idiom in TXL programs is to first include the base grammar and then to change selected nonterminals with redefine statements. For example, the left side of Fig. 2 shows a definition of a statement nonterminal for a toy imperative language. In order to introduce a block construct to this toy language, a redefine can be used that extends the nonterminal with a block_statement alternative (right side of Fig. 2).

Whereas parsers typically target only a single language and offer very limited customizations (e.g., via command-line switches), lexical analyzers do not target a particular language and can be extensively customized via pattern specifications. Cox and Clark make the following observation about their lexical extractor:

> Customizations of parsers is difficult to accomplish because it is necessary to understand the particular parsing technique (e.g., LALR or LL) as well as the grammar itself. Since DIALECT is a LALR(1) parser, it probably needs expert knowledge to actually customize it.

---

```
% base grammar
define statement
  [declaration]
  | [if_statement]
  | [while_statement]
end define

% extending a nonterminal
include 'base grammar'
redefine statement
... | [block_statement]
end redefine

define block_statement
  'begin [statement*] 'end
end define
```

Fig. 2. Example of a grammar override in TXL.
“Lexical tools are often faster to develop than parser-based tools and, when developed using hierarchical pattern sets, can be easily extended or adapted for novel situations. Extension is performed through the addition of new lexical levels or additional patterns in an existing level” [140].

2.3.3 Analyzers

Most program analyses are fixed in the sense that the reverse engineer cannot turn any knobs to influence the outcome of the analysis (e.g., one cannot trade speed for precision, or vice versa). For instance, there are many clone detection analyses, but few of them can be easily customized to control what constitutes a code clone and what does not. However, flexible analyses can be valuable because it allows the reverse engineer to instruct an analysis to focus its effort on information that is most relevant to a particular reverse engineering task [50].

Jackson and Rinard believe that software analyses should give the engineers more control, for instance, to customize the precision of an analysis: “Engineers need different degrees of precision in different situations, at different points in the program, and for different data structures. Applying a single analysis uniformly across the entire program is therefore counterproductive” [49].

The IntensiVE tools offer a specification language to express structural constraints of Java source code (e.g., to check for design patterns or coding conventions) [141]. Constraints are written in a declarative Prolog-style language, called SOUL, that has predefined predicates that match Java language concepts (e.g., MethodDeclaration and CatchClause). The unification semantics of SOUL can be customized to follow different rules. Figure 3 shows how the unification of predicates can be customized. Atkinson and Griswold have developed a whole-program analysis tool that allows the user to control the precision of its analysis as well as its termination criteria [142]. This avoids wasted resources caused by analyses that are more general than a certain reverse engineering activity actually requires.

2.3.4 Visualizers

In Bassil and Keller’s survey on software visualization tools, 45% of the respondents rated the “possibility to customize visualization” as “absolutely essential” [79]. The developers of the Sextant software comprehension tool say, “we conclude that software exploration tools should be extensible to accommodate for domain-specific navigation elements and relationships as needed” [143]. Reiss states that “since we cannot anticipate all visualizations initially, it should be easy to add new graphical objects to the system” [144]. Among the requirements that Favre states for
his GSEE software exploration environment is the need for “customizable exploration” [123].

Reiss has analyzed why software understanding tools are often not used and concludes that “the primary reason was that they failed to address the actual issues that arise in software understanding. In particular, they provided fixed views of a fixed set of information and were not flexible enough” [145]. Even though customizations seem important, Wang et al. say that “existing visualization tools typically do not allow easy extension by new visualization techniques” [146]. They have developed the EVolve software visualization framework, which “is extensible in the sense that it is very easy to integrate new data sources and new kinds of visualizations.” A new visualization is realized by extending the EVolve Java framework, which already provides abstractions for bar charts, tables, dot-plots, and so on. Storey et al.’s visualization evaluation framework addresses customizability of tool interactions:

“Effective interaction to suit particular user needs will normally require a high degree of customization. . . . Saving customizations and sharing customizations across team members may also be important” [82].

Storey et al. have evaluated 12 different visualization tools, concluding that Advizor and Xia/Creole have high customizability; VRCS, Palantir, and Jazz have low customizability; and the rest having no support for customizability. Based on this study it appears that there is a number of tools that support customizability, but this feature is not yet pervasive.
Visualization tools support a number of customization mechanisms. In the following we discuss typical ones with examples:

**Extensibility hooks:** Tools can provide hooks that make it easier to add functionality programmatically. In order to add functionality one has to write code in the tool’s implementation language. Tool provides basic hooks via subclassing, design patterns, and APIs. There are also more advanced schemes where tools are implemented as frameworks and have plug-in architectures (Section 3.1).

For example, the EDGE graph editor allows to customize nodes and edges via subclassing from base classes that support default behavior [147]. EDGE also provides callbacks that are invoked for actions such as reading a graph, drawing a node, and deleting an edge. Similarly, Graphviz has an API that allows to manipulate the in-memory representation of graphs as described in Dot [148]. EVolve is a visualization framework implemented in Java. In order to support a new visualization, the Visualization class needs to be subclassed and the class’ abstract methods must be implemented [146]. The GSEE tool is implemented as an object-oriented framework dedicated to software exploration and a set of customizable tools that instantiate the framework.

**Scripting:** Tools can offer a scripting layer to simplify end-user programming. AT&T’s graphviz provides an interactive editor, Dotty, which can be customized with a dedicated scripting language, called Lefty [148]. The Rigi graph editor can be scripted with Tcl/Tk [23]. This makes it easy to modify Rigi’s user interface to provide new user interactions and to change existing ones (e.g., via adding or modifying drop-down menus and pop-up forms). However, Rigi’s graph visualization cannot be easily changed (e.g., the shapes of nodes are fixed). The implementors of the Dali architecture reconstruction tool chose the Rigi tool for its extensibility via scripting:

“We needed to use a tool that provided both domain specific functionality—to do the necessary graph editing and manipulation—and extensibility, to integrate with the rest of the functionality of Dali. We are currently using Rigi for this purpose, since it provides both the ability to manipulate software models as graphs and extensibility via a control language based on Tcl” [114].

Soft Vision is an open visualization toolkit whose functionality is very similar to Rigi’s. The toolkit has a layered architecture consisting of a C++ core to improve performance and a Tcl/Tk layer [149]. The C++ core exposes an API to the scripting layer. Customization is accomplished either via the C++ API or Tcl scripting. The authors made the experience that “for most visualization scenarios imagined by our users, writing (or adapting) a few small Tcl scripts of under 50 lines was enough. This was definitely not the case with other reverse engineering systems we worked with” [149].

Mondrian is a visualization framework that can be scripted with Smalltalk [150]. Since the scripting provides intuitive abstractions new visualization can be defined
with a few lines of code. To give an intuition of Mondrian’s capabilities, Fig. 4 shows as sample script and the resulting graph that it generates.

Declarative specifications: Declarative specifications can take various forms, ranging from domain-specific languages to setting option interactively with the tool’s user interface. Dot has a specification language to define a graph and to control its rendering. Similarly, EDGE has a description language to specify properties of the graph [147]. The following gives an example:

Graph.x-spacing: 30
Graph.layout.algorithm: ReduceCrossings
Node.borderwidth: 2
Node.label.font: TimesRoman8
Edge.arrow.style: solid
Edge.routing: straight

Configuration files and UI-based customizations can be seen as a rudimentary form of declarative specification.

In the best case, tools offer various customization strategies. For example, the Rigi tool has a startup file to configure fonts, icons, text editor, etc. As mentioned before, more advanced customizations can be performed via Tcl/Tk scripting. However, Rigi’s architecture provides no direct support to easily customize its underlying C/C++ implementation.

Having various strategies at their disposal allows users to make trade-offs decisions. For example, programmatic customization is very powerful, but has a
steep learning curve. In contrast, scripting offers better rapid prototyping, but may lack needed performance. And so on.

2.3.5 Discussion

Whereas our survey suggests that researchers see tool customizability as an important requirement, many tools are lacking in this respect, especially extractors and analyzers. In fact, customizability of extractors and analyzers is often forced by scalability problems. For example, dynamic traces easily can become too huge to be efficiently stored, retrieved, and investigated. As a result, trace extractors can be customized for extracting only information at a certain level of granularity (e.g., functions, blocks, or statements), certain event types (e.g., function return or object creation), or parts of the execution (e.g., code of certain classes or packages) [19]. There is also the approach to have a fixed extractor or analysis with the idea to have its result then customized in a separate processing step (e.g., by a subsequent analysis and/or visualization).

There are also analyses that are generic (i.e., they can operate independently of the actual data or schemas) and thus do not need to be customized. A typical example of such an analysis is a graph layout or a textual differencing algorithm. While such generic analyses have the advantage that they are applicable across all schemas, they have the disadvantage that they cannot exploit the domain knowledge encapsulated by a particular schema.

A tool’s schema offers a key opportunity to achieve customizability. Indeed, Sim et al. offer the thesis that “tools supporting program comprehension and software maintenance require flexible conceptual models that can be modified as the user or task requires” [151]. Equally importantly, schemas should allow introspection during run-time. This has been already realized by reverse engineering repositories. Similarly to the schemas of repositories, tools should reify conceptual properties such that they can be queried and perhaps even modified during run-time. This could lead to tools that can be customized to the task at hand during run-time (rather than during a dedicated customization step that is performed off-line). A step in this direction is the Fame library, which enables run-time meta-modeling [40].

Last, tools are currently focusing exclusively on customization, disregarding personalization. In contrast to customization, which is controlled by the user,

---

6 These analyses typically exploit the fact that all schemas adhere to a common meta-schema (or meta-meta-model). For example, all RSF schemas are composed of nodes, arcs, and attributes, even though concrete schemas differ by the actual types of these entities.
personalization is initiated by the tool itself. It seems worthwhile to try to apply research results of personalization of systems such as web applications to tools.

2.4 Usability

“Nice tool, but it takes some time to get used to it.”
user feedback for the sv3D visualization tool [152]

Usability can be defined as the ease of use of a system for a particular class of users carrying out specific tasks in a specific environment [153]. This definition emphasizes that usability depends on the context of use as well as the user.

Usability encompasses a set of other quality attributes or characteristics such as [153, 154]:

Learnability: The system should be easy to learn so that the user can rapidly begin working with the system.

Efficiency: The system should be efficient to use, enabling a user who has learned the system to attain a high level of productivity.

Memorability: The system should be easy to recall, allowing the casual user to return to the system without having to relearn everything.

Satisfaction: The system should be pleasant and comfortable to use.

Among the goals of software engineering are the construction of systems that users find both useful and usable. The same is true for the construction of reverse engineering tools. Meeting the usability requirement of users has several benefits [154]. It improves the product, resulting in productive and satisfied users; increases the reputation of the product and the developer, potentially increasing the product’s use; and decreases costly redevelopment caused by user complaints.

Researchers in the reverse engineering field have pointed out the importance of usability as follows:

—In a position statement for a WCRE panel, Müller et al. state, “reverse engineering tool developers not only need to understand the technology side but also need to take the business requirements and the application usability more and more into account” [155].

—Walenstein says in his dissertation about cognitive support in software engineering tools: “The first rule of tool design is to make it useful; making it usable is necessarily second, even though it is a close second” [156].

—Discussing tool design and evaluation, Wong assures that “for program understanding tools, careful design and usability testing of the user interface is important” [62].
—In a talk entitled *Creating Software Engineering Tools That Are Usable, Useful, and Actually Used*, Singer states: “Simply put, if a tool isn’t usable it won’t be used” [157].

—Maccari and Riva have conducted an empirical evaluation of CASE tool usage at Nokia [158]. The respondents rated the modeling requirement to “be intuitive and easy to use” as highly useful (i.e., the median value of the responses was above four on a five-point scale). However, based on their experience with existing CASE tools such as Rational Rose, the respondents replied that this requirement is “insufficiently well implemented.”

—In a survey, participants rated the importance of requirements for software visualization tools [79]. The requirement “ease of using the tool (e.g., no cumbersome functionality)” was selected as the second-most important practical aspect, which 72% rated as very important (i.e., the highest value on a four-point scale) [79, p. 8]. However, the authors of the survey believe that “unfortunately, we found a disturbing gap between the high importance attached to the two aspects ease of use and quality of the user interface, and the ratings of these qualities in the software visualization tools in practical use” [79].

Tool developers often are not aware of the importance of usability or do not know how to achieve it. However, in order to systematically identify and improve a tool’s usability problems, it is necessary to change tool developers’ attitudes toward usability [159]. There are two approaches how one can design for usability [154]: product-oriented and process-oriented.

### 2.4.1 Product-Oriented Usability

Product-oriented approaches consider usability to be a product characteristic that can be captured with design knowledge embodied in interface guidelines, design heuristics, and usability patterns. For instance, Toleman and Welsh report on the evaluation of a language-based editor’s interface based on 437 guidelines [160]. This catalog covers functional areas such as data entry, data display, sequence control, and user guidance [161].

The evaluation of the usability of reverse engineering tools is often focused on the user interface. Toleman and Welsh say, “user interface design guidelines are an important resource that can and should be consulted by software tool designers” [160]. In Bassil and Keller’s survey, 69% believe that “quality of user interface (intuitive widgets)” is a very important requirement [79, p. 12]. Reiss, who has implemented many software engineering tools, believes that “having a good user interface is essential to providing a usable tool” [162]. He provides a rationalization for the user interface choices of the CLIME tool, but does not support his decisions with background literature. For the BLOOM software visualizer, Reiss emphasizes
that both usefulness and usability are important: “While it is important to provide a wide range of different analyses and to permit these to be combined in order to do realistic software understanding, it is equally important to offer users an intuitive and easily used interface that lets them select and specify what information is relevant to their particular problem” [145].

Storey’s cognitive framework emphasizes usefulness aspects for comprehension tools, but also has a design element that requires to “reduce UI cognitive overhead” [163]. Storey et al. further elaborate on this design element, stating that “poorly designed interfaces will of course induce extra overhead. Available functionality should be visible and relevant and should not impede the more cognitively challenging task of understanding a program” [164].

Design heuristics suggest properties and principles that are believed to have a positive effect on usability. Heuristics address issues such as consistency, task match, memory-load, and error handling. Bass et al. have collected 26 general usability scenarios [165]. A scenario describes an interaction that a user has with the system under consideration from a usability point of view. Examples of scenarios are Aggregating Data (i.e., systems should allow users to select and act upon arbitrary combinations of data), Aggregating Commands (i.e., systems should provide a batch or macro capability to allow users to record, aggregate, and replay commands), Providing Good Help (i.e., systems’ help procedures should be context dependent and sufficiently complete to assist users in solving problems), Supporting International Use (i.e., systems should be easily configurable for deployment in multiple cultures), Modifying Interfaces (i.e., system designers should ensure that their user interfaces can be easily modified), and Verifying Resources (i.e., systems should verify that all necessary resources are available before beginning an operation).

To our knowledge, there is no catalog of design heuristics for the reverse engineering domain. However, researchers sometimes relate their experiences, providing tidbits of ad hoc usability advice. Examples of such tidbits, grouped by usability characteristics, are:

**Learnability**: Respondents in Bassil and Keller’s survey view learnability as relatively less important; less than half see “ease of learning and installation of the tool” as a very important aspect [79, p. 7]. In contrast, Reiss emphasizes that software developers “will use new tools, languages, resources, etc., if (and this is a big if) the cost of learning that tool does not exceed its expected rewards and the tool

---

7 Unfortunately, this item in the survey combines two distinct attributes: learnability and ease of installation. The authors decided to group these together because both attributes can be considered as a necessary up-front investment to get productive with the tool (private e-mail correspondence with Rudi Keller).
has been and can easily shown to provide real benefits” [166]. For search tools that are based on pattern matching, Bull et al. require “an easy to specify pattern” [167]. This requirement trades improved query learnability (as well as simplicity and specification time) for less expressive power [168].

**Easy installation:** A tool should be easy to install. Reiss says that “it is rare to find a software visualization tool that an uninformed programmer can take off the shelf and use on their particular system immediately” [166]. Generally, the more difficult the installation, the less likely that the tool will be tried out. Thus, in the best case a tool requires zero installation. A promising approach is to make tool functionality available via the web browser. D’Ambros et al. say that “if a tool is available as a web application then there is no installation and the cost for people to ‘give it a try’ is minimal” [169]. The REportal reverse engineering tool is implemented as a Web portal. Users can upload the source code that they want to analyze and then run analyses; thus “users are not required to install any software on their computers in order to use the portal services” [170].

**Efficiency:** According to Reiss, a tool’s usage should “have low overhead and be unintrusive” [134]. Especially, it should not interfere with existing tools or work practices. Reverse engineers often manually inspect, create, and modify the data represented with an exchange format. To simplify this activity, these formats should be human readable and composable (Section 2.6).

**Memorability:** The scripting interface of a tool should not overwhelm the user with too many commands. For instance, Moise and Wong made the experience that “the Rigi command library was difficult to learn and awkward to use with the sheer number of commands” [99].

**Satisfaction:** In order to keep reverse engineers motivated, tools should be enjoyable to use. Especially, “do not automate-away enjoyable activities and leave only boring ones” [171]. Tools should be designed to “keep control of the analysis and maintenance in the hands of the [users]” [172]. Otherwise, the user may feel threatened and devalued by the tool.

### 2.4.2 Process-Oriented Usability

Process-oriented approaches such as user-centered design consider usability as a design goal that can be achieved with a collection of techniques involving the end users (e.g., task analysis, interviews, and user observations). Singer answers the question of “How do you make a tool usable?” with the recommendation to “conduct pilot tests on [the] target user group,” and to “bring it to users early and often” [157].

The SHriMP tool has a history of user studies, which also had the goal to improve the tool’s usability. For instance, a pilot study (involving 12 users who were
videotaped using think-aloud) has lead to recommendations for interface improvements to SHriMP (and Rigi) [102, 163]. These recommendations were then used to redesign SHriMP’s interface [163]. The redesign involved, for instance, a more effective navigation of the visualized graphs combining context+detail with pan+zoom, alternative methods of source code browsing, and the introduction of modes to reduce the cognitive overhead of the users during navigation. The new interface was then evaluated with another user study (which used videotaping and think-aloud, but also a questionnaire and an informal interview) [163, 173]. Besides SHriMP, the TkSee tool is also an example of a reverse engineering tool that has employed user studies (see below).

Awkward usage scenarios or work patterns can also give hints on how to improve usability. For instance, observing the work of professional software engineers, Singer and Lethbridge found that they did “jumping back and forth between tools, primarily Unix command line (performing grep) to editor and back. This jumping involved the use of cut and paste to transfer data and was frequently awkward” [174]. This scenario points toward a better integration and interoperability of tools to improve usability.

2.4.3 Usability in TkSee

The design and evolution of the TkSee search tool is an example of a tool-building effort that has combined both product-oriented and process-oriented approaches in the form of guidelines and user studies to improve the tool’s usability. A product-oriented approach was followed by evaluation of TkSee based on Nielsen’s usability guidelines [175]. Three evaluators identified 114 usability problems. The types of problems found were poor or missing feedback (e.g., what has happened following an interaction), possible confusion about tool behavior, possible misinterpretation (e.g., meaning of labels or menu items), poor labeling, lack of labeling, lack of consistency, poor graphical design, unnecessary features, lack of needed features, lack of robustness (e.g., tool crashes or hangs), incorrect behavior, and nonoptimal interaction.

The usability analysis of TkSee showed that it is important to have several evaluators with different backgrounds. One of the evaluators had a background in usability, but no background about the problem domain (i.e., program comprehension). This person tended to find general usability problems related to feedback, labeling, and graphical design, etc. In contrast, another evaluator that was already knowledgeable about TkSee and the problem domain tended to point out missing features and incorrect behavior. TkSee was also evaluated with a user analysis involving videotaping and think-aloud usability testing [175]. Eight participants found 72 problems, of which 53% had already been identified before by the
evaluators. This shows that both product-oriented and process-oriented approaches are complimentary and that both should be used to evaluate a tool’s usability. Figure 5 shows a screenshot of TkSee after its redesign, identifying some changes that were made as a result of the usability evaluation.

2.4.4 Discussion

In 1991, Grudin observed that “resistance to unfriendly systems is growing. There is growing competitive pressure for usability in the marketplace, particularly in mature application domains” [176]. Almost two decades after this statement, it is questionable whether usability of software has improved drastically. Whereas the problem of usability seems to have more visibility, it is still difficult to overcome due to other, competing pressures such as feature creep and time to market.

Usability is recognized as a problem by researchers, but it is often addressed in an ad hoc manner. Tool developers rarely discuss how they established the usability of their design. Toleman and Welsh testify, “in general, the design rationales for
software tools that are available rarely indicate the basis for the design of the user interface” [160]. There is the underlying problem that researchers in reverse engineering have neither made an attempt to define nor clarified what they mean by usability. As a result, usability is often judged subjectively by the tool developers. Toleman and Welsh criticize that “software tool designers consider themselves typical users of the tools that they build and tend to subjectively evaluate their products rather than objectively evaluate them using established usability methods” [160]. Too often, usability is only superficially addressed. Lanza and Ducasse address the usability of their CodeCrawler visualization tool by saying that “our tool has been downloaded over 2000 times and, although we have not performed a user survey yet, from personal and e-mail discussions with the users, we have learned that after a short learning time they know what they can get out of each view” [57]. Whereas these indications are encouraging for CodeCrawler indeed, they cannot replace a more formal usability assessment (such as exemplified by SHriMP and TkSee). In contrast, the developers of the Sextant software exploration tool go one step further by first stating five functional requirements (i.e., integrated comprehension, cross-artifact support, explicit representation, extensibility, and traceability), and then arguing how Sextant meets these requirements [177].

Storey’s cognitive dimensions framework is mostly focused on improving the usefulness of a program comprehension tool, but not its usability. Green and Petre introduce a framework to assess the usability of programming environments [178]. As a start, researchers should apply existing usability framework while developing usability guidelines specifically for the reverse engineering domain.

2.5 Adoptability

“Technologists tend to think that if they build a good thing, people will find their way to it and adopt it on their own, based on its inherent goodness...Wrong.”

Lauren Heinz [179]

For almost all new ideas, practices, technologies, tools, and other innovations8 in general, there is the concern of how to get them adopted. As the above quote suggests, adoption of innovations cannot be taken for granted, regardless of the perceived benefits by its proponents. This painful experience has been repeatedly made by different innovators in diverse areas. An example of a famous adoption

8 Roger defines an innovation as “an idea, practice, or object that is perceived as new by an individual or other unit of adoption” [181, p. 11].
problem in the software engineering area are CASE tools. Even though CASE tools were promoted as significantly increasing software development effectiveness in terms of productivity and quality, many developers did not use these tools or abandoned them later on—leading to questions such as “why are CASE tools not used?” [181].

The attempt to move toward a common exchange format for reverse engineering is an interesting example of the adoption of a new standard within a research community. The goal of a common exchange format is to simplify tool interoperability. However, to achieve this vision, a diverse group of stakeholders have to agree to adopt a standard first. Establishing a standard exchange format or schema is difficult because existing tools have to be modified for compliance, which may not be economical [108]. But without a critical mass, an exchange format does not make the transition to a standard exchange format. Adoption of a new format can be encouraged by addressing the functional and nonfunctional requirements of its users. In other words, the exchange format should be an “early and clear win for adopters” [182].

Perhaps the most important theory that is able to explain adoption is diffusion of innovations [180]. The theory’s roots are in sociology, but there are hundreds of publications that have applied it to study the adoption of innovations in a vast number of fields. Diffusion of innovations has identified the following characteristics as most significant [180, p. 15]:

Relative advantage (+): Relative advantage is the degree to which an innovation is perceived to be better than what it supersedes. The immediacy of the rewards of adopting an innovation is also important and explains why preventive innovations have an especially low adoption rate.

Relative advantage can mean that reverse engineers notice that a certain tool allows them to perform certain tasks with more ease, in less time, or with higher job satisfaction.

Compatibility (+): Compatibility denotes the consistency with existing values, past experiences, and needs. Generally, innovations are understood by putting them in relation with the familiar and the old-fashioned.

Favre et al. make the observation that “users are more likely to adopt a tool that works in the same environment they use on a daily basis. This means that SE tools should be integrated to the existing set of tools” [87]. For example, Buss and

9 Whereas it is clear that the users of an exchange format are researchers in the reverse engineering field, it is difficult to assess the requirements of this rather diverse community. Proposals for standard schemas face the same problem; a group of researchers proposing a schema for C/C++ note, “there is one fundamental issue that we have not yet resolved: who are the end users of this schema and what are their requirements?” [109].
Henshaw report on a platform conflict of their tool; as a result “the product maintainers are uncomfortable with the unfamiliar environment on which the analysis is run” [183]. The tool designers of sgrep try to improve adoption by making their tool compatible with a popular existing search tool, grep. They state, “sgrep is designed to be used in place of grep, so it is important that many of the design decisions found in grep, transfer over to sgrep” [167]. Since grep is a command-line tool, sgrep follows this pattern: “Although graphical user interfaces are often easier to use for novice users, we believe that familiarity is more important than ease of use, for the kinds of tasks we envision for sgrep” (emphasis added).

Complexity (−): Complexity is the difficulty of understanding and using an innovation.

Adoption of a tool can be increased by making it easier to use, or by providing training sessions and appropriate documentation. Complexity can also be reduced by identifying and eliminating unnecessary tool features.

Trialability (+): Trialability denotes the degree to which an innovation can be experimented with, without committing to it.

The authors of a work practice study involving the TkSee tool believe that an important factor in the adoption of the tool by developers was that “we allowed them to continue their existing work practices (e.g., use of grep), rather than forcing them to adopt a radical new paradigm” [174]. Also, the TkSee search tool can be easily tried out because reverse engineers can readily switch between TkSee and other search tools that they were using before. A tool is easier to try out if it is easy to install (Section 2.4).

Observability (+): Observability is the degree to which the results of an innovation are visible to others.

For instance, if a tool is visibly used for other developers, it can have a beneficial impact on adoption. Kollmann relates an experience he made in an industrial project: “It can be observed that once a certain number of people have made positive experiences with [an innovation], the propagation is often carried out considerably faster. People seem to trust the experiences others have made and are easier convinced to come aboard, resulting in a kind of snowball effect” [184].

These characteristics help to explain the rate of adoption. Adoptions that are positively related (“+”) with the above characteristics will be adopted more rapidly than other innovations. Besides these characteristics, there are other factors that determine adoption, for instance, communication channels, nature of the social system, activities of change agents, and individual/collective decision-making.

Developers of reverse engineering tools have mostly ignored the question of whether their tools are actually adopted by software developers and maintainers. For program understanding tools, Mayrhauser and Vans have observed expectations that users better adapt to a tool if they want to use it: “we still see attitudes reflected in tool builders’ minds that if we just teach programmers to understand code the way
they ought to (i.e., the way their tools work), the understanding problem will be
solved” [185]. Thus, instead of lowering adoption barriers and increasing the users’
incentives to adopt, this attitude expects users to pick up a tool in spite of the raised
adoption hurdles.

In the last few years, however, the reverse engineering community has started to
pay more attention to this question. This trend is exemplified by the following
sources:

—In 1996, Rugaber and Wills already point out that there is an adoption problem
of reverse engineering tools: “Reengineering research has had notably little effect
on actual software reengineering practice. Most of the published papers in the field
present techniques supported by prototype tools; few of which have actually been
used on real projects” [186].

—Eight years later, the organizers of the Fourth International Workshop on Adop-
tion-Centric Software Engineering come to a similar conclusion: “Research tools in
software engineering often fail to be adopted and deployed in industry” [187].

—Lethbridge makes the general observation that “one of the beliefs that moti-
vates software engineering tools builders is, ‘if we build it, they will come.’
Unfortunately, they often don’t come and we wonder why” [188]. Similarly,
Wong stresses that it is not enough to devise a new technique or tool and “simply
expect practitioners to pick it up and adopt it” [62].

—Software exploration tools use graphical presentation to visualize information
about a software system. Even though researchers perceive these tools as valuable
for reverse engineering and maintenance tasks, Storey reports that “despite the large
number of software visualization tools, few of these tools are used in practice”
[164]. Storey et al. use the adoption problem as motivation to propose a framework
of cognitive design elements to guide tool design.

—In a roadmap paper for reverse engineering research for the first decade after
the year 2000, Müller et al. state that they believe “perhaps the biggest challenge to
increase effectiveness of reverse engineering tools is wider adoption; tools can’t be
effective if they aren’t used” [46].

—In his dissertation, Wong demands from researchers to “address the practical
issues underlying reverse engineering tool adoption” [62]. Discussing lessons
learned from his Reverse Engineering Notebook, he says, “make adoption issues
an integral part of reverse engineering tool research and design” [62].

2.5.1 Tool Adoption Research

There are few researchers who see the adoption of their tools as a first-class
requirement for their research endeavor. One notable exception is the Adoption-
Centric Reverse Engineering (ACRE) project, which explores the adoption problem
of reverse engineering tools. It has initiated a series of four workshop on Adoption-Centric Software Engineering (ACSE 2001–2004) [187]. ACRE addresses the adoption problem with two lenses, cognitive support and interoperability. Since most research tools only support a few selected program understanding or maintenance tasks, reverse engineers typically have to integrate them with other tools to use them effectively. ACRE proposes to investigate the use of data, control, and presentation integration technologies such as XML protocols, the GXL exchange format, Web browsers, ECMAScript, SVG, Eclipse, and web services to make tools more interoperable and thus more adoptable. The other lens, cognitive support, refers to the means by which tools assist the cognitive work of their users (i.e., thinking and reasoning) [156]. Examples of everyday tools that provide some form of cognitive support are shopping lists, address books, and pocket calculators. Without them, certain tasks would have to be performed with an increased cognitive load (e.g., in terms of memorization and computation).

Lethbridge considers adoption using three factors: costs, benefits, and risks of tool use [188]. Potential adopters often do not perform a formal analysis of these factor, relying on their “gut feeling” instead. Examples of costs of use are (c1) purchasing of the tool, (c2) purchasing of extra hardware or support software, (c3) time to install and configure the tool, and (c4) time to learn the tool. Examples of benefits of use are (b1) time saved by the tool, and (b2) value of the increased quality of work done. Examples of risks are (r1) costs are higher than expected, (r2) benefits are less than expected, (r3) unintended negative side effects (e.g., data corruption), (r4) discontinued tool support, and (r5) difficulty to revert to previous work environment. Discussing the factors, Lethbridge says that “in addition to perceiving costs and benefits differently, adopters will more intensively perceive the risks, and the more risks they perceive, the more their perceived benefits must exceed their costs of adoption to take place” [188]. In contrast, tool researchers tend to focus on costs and benefits, ignoring or down-playing the risks. Lethbridge’s factors can be used to assess tool-adoption scenarios. For instance, adopting a reverse engineering tool that is build on top of an office suite (as envisioned by ACRE) should have low purchasing costs assuming the office suite is already used (c1 and c2), a simple installation process if the tool is provided as a plug-in (c3), and favorable learning curve resulting in saved time (c4 and b1). On the other hand, updating the office suite might render the plug-in inoperative (r4) and users might become trapped in a certain data format (r5).

Tilley et al. have looked at the adoption of research-off-the-shelf (ROTS) software [189]. They say,

“in our opinion, adoption is one of the most important, yet perhaps least appreciated, areas of interest in academic computer science circles. . . . Indeed, it can be argued that ‘transitionability’ as a quality attribute should receive more emphasis in most software projects”
In applied fields such as software engineering, “it may be a measure of success for the results of an academic project to be adopted by an industrial partner and used on a regular basis.” However, whereas adoption is a desirable (long-term) goal for a research project, it is not a necessary criterion for success. This is caused by the academic reward structure, which emphasizes publications rather than workable tools. As a result, the adoption of ROTS software is complicated by lacking “completeness (e.g., a partial solution due to an implicit focus on getting ‘just enough’ done to illustrate the feasibility of a solution, rather than going the ‘last mile’ to bring the prototype to market).” This is especially the case if the software is the result of a one-person effort produced as part of a Master’s thesis or dissertation. Additional complications for adoption are “understandability (e.g., a lack of high-quality documentation)” and “robustness (e.g., an implementation that is not quite ready for prime time).” Huang et al. also look at the relationship between academia and industry: “Both parties know that they have a symbiotic relationship with one another, yet they seem unable to truly understand what each other needs” [190]. Industry has a potential interest in research results that can mature and then be integrated into existing processes to improve software development. To encourage interest from industry, researcher have to solve relevant problems and not to work on problems that are “removed from the current needs of potential users.” Also, adoption of tools and techniques by industry could be encouraged with third-party case studies and quantitative data, but these are rarely available for ROTS software [191].

2.5.2 Adoption Factors

Researchers have suggested many potential factors that affect tool adoption, summarized as follows:

**Tool:** Researchers have mostly focused on factors that can be attributed to the tool itself. To get adopted, tools have to be both useful and usable. Storey says, “although there are many software exploration tools in existence, few of them are successful in industry. This is because many of these tools do not support the right tasks” [163]. Bull et al. state, “in any field, ease of use and adaptability to the tasks at hand are what causes a tool to be adopted” [167]. Wong believes that “lightweight tools that are specialized or adaptable to do a few things very well may be needed for easier technology insertion” [192]. He also says, “by making tools programmable, they can more easily be incorporated into other toolsets, thus easing an adoption issue of tool compatibility” [62]. Tilley et al. suggest that research tools might be more adoptable if they were more understandable, robust, and complete [189]. Devanbu points to inadequate performance of research tools, which are often not intended for large-scale applications [193]. In contrast to most research tools that
require some effort and expertise for installation, popular tools are easily installed or even already preinstalled [194].

User: For adoption, besides the characteristics of the tool, characteristics of the tool users play an important role. When starting to use a tool, users often want positive feedback very quickly. Tilley makes the point that “it is an unfortunate fact that many developers will give a tool only a short window of opportunity to succeed. If they cannot get the tool up and running in 10 min and see real results, without looking at the manual, they will often abandon the tool” [195]. Many users would not even consider a very small trial period no matter of the potential benefits that it promises—if users are happy with their existing tools they do not see the need of trying (yet) another one. Devanbu [193] says,

“developers are pressured by schedules, and keenly aware of the need to meet cost, schedule or quality requirements. This engenders a conservative bias towards simple, and/or familiar tools, even if somewhat outdated. Builders of complex tools with steep learning curves (even ones promising significant gains) face the daunting hurdle of convincing busy developers to invest training time”

The TkSee tool has collected some experiences with developers at Mitel. Less than half of the developers at Mitel used the tool for significant work over a 2-year period. For users who did not adopt, Lethbridge and Herrera found that “at some point during their learning attempts, many users had concluded that further learning was not worth additional investment of their very limited time” [175]. Interestingly, user that did not adopt tended to state reasons that revealed misconception of the TkSee tool “either because it had not proved rapidly learnable or else because they had found some aspect of it difficult to understand” [175]. TkSee was introduced to the developers without “extensive documentation or training.” Lethbridge and Herrera elaborate on this point, “we feel sure that a more proactive training program might have helped increase adoption to some extent; however, we do not feel that more extensive documentation would have helped much—we hardly ever observed anyone look at the existing documentation” [175].

Organization: Often developers do not make the decision to adopt a tool all by themselves because they are constrained by their organization.\textsuperscript{10} This is true for industry, which often mandates certain tools, as well as open source projects that assume a certain toolset (e.g., GCC, CVS, and Bugzilla).

From an organization’s perspective, “adopting a different toolset or environment discards the hard-earned employees’ experience” [196]. Furthermore, “changing

\textsuperscript{10} Rogers defines an organization as “a stable system of individuals who work together to achieve common goals through a hierarchy of ranks and a division of labor” [181, p. 375].
tools requires changing processes, which is also an expensive undertaking. As a result, the state of tool adoption in industry is rather static and conservative” [196]. Devanbu also stresses that tools have to integrate smoothly into an existing process: “Software tools work best if their operation is well tuned to existing processes. Thus, a tool that generates paper output may not be helpful in an email-based culture” [193]. Similarly, Wong says “software understanding techniques and tools need to be packaged effectively and made compatible with existing processes, users, and tools” [62].

In a social system such as an organization, the adoption of innovations can be promoted by change agents or innovation champions [180, p. 398]. If such an entity is missing the innovation probably will not get adopted. Lethbridge and Singer have rediscovered this approach for tool adoption of TkSee: “We had significant difficulty introducing new tools. One technique that seems to hold promise is to train a single individual (in our case somebody new to the organization) and have him or her act as a consultant for others” [197].

Whereas the diffusion of innovations theory has developed characteristics that hold across innovations and social systems, there might be also the need to consider domain-specific characteristics. Cordy reports his experiences with the technical, social, cultural, and economical drivers of the Canadian finance industry and financial data processing in general, which he gained during 6 years of project work with Legasys Corporation [172]. In Cordy’s experience, resistance to tool adoption is strong because of unhappy past experiences with many inadequate and premature CASE tools. As a result of the pressure of quick, low-risk enhancements of financial applications, for maintainers “only the source is real” [172]. Thus, reverse engineering tools have to present results in terms of source, and not abstract diagrams. Also, robust extractors are needed because “having no answer is completely unacceptable, and programmers will rapidly drop any analyzer that fails to yield answers due to parse errors.” The decision to adopt a tool is not made by upper management, but individually by the maintenance programmers and their group manager. In order to convince programmers to adopt a maintenance tool, it is important that they do not feel threatened by it; the workflow of the tool should be such that “all control is left in the hands of the programmer.” Cordy says, “this philosophy of assist, don’t replace, is the only one that can succeed in the social and management environment of these organizations.” Cordy believes that “by studying the maintenance culture of each industrial community, by treating their way of doing things with respect, and by working to understand how our techniques can best be fit into their existing working environment, we can both increase chances of adoption and enhance our own success” [172].

Cost: The adoption of an innovation can be explained by the cost that it causes the users and their organization. Glass [198] says that
“learning a new tool or technique actually lowers programmer productivity and product quality initially. The eventual benefit is achieved only after this learning curve is overcome. Therefore, it is worth adopting new tools and techniques, but only (a) if their value is seen realistically and (b) if patience is used in measuring benefits.”

Patterson makes the point that research software is free of charge, but this does not mean that adoption and use of it has no cost—there is a difference between cost of purchase and the cost of ownership [199]. An example of the cost of ownership is tool administration: “Once installed, some complex tools require a great amount of administration. . . . The amount of work and the skills required could be a serious barrier to the adoption of complex tools” [87].

Tilley and Distante say, “ultimately, people will only adopt a technique if they see significant benefit in its use that far outweigh the costs associated with learning and using the technique in the first place” [200]. Storey et al. state that “the adoption of any tool has a cost associated with it. Economic cost is a key concern, in addition to other costs such as the cost of installing the tool, learning how to use it, and the costs incurred during its usage” [82]. In their tool survey, Bassil and Keller did ask for the importance of the “cost of the tool” [79, p. 1]. Interestingly, 50% of the respondents from academia rated this aspect as “very important,” compared to only 32% of respondents from industry.

2.5.3 Discussion

There are many factors that influence the adoption of a tool—and many of these cannot be influenced by the tool developers directly. Rifkin reaches a similar conclusion when he says that “as designers of processes and tools that we want adopted by others, we should understand that there is only so much power in the technical content of our processes and tools” [201]. However, tool developers should make an effort by leveraging the factors that they are able to influence to increase the likelihood of tool adoption.

In order to increase the incentives of academic researchers who conduct applied research to focus more on the adoption of their proposed tools and techniques, it is necessary to change the academic reward structure. Researchers already have an incentive to raise adoption to a first-class requirement because an adopted tool has indirectly proven its usefulness [202]. However, as Storey points out, the opposite is not necessarily true: “A lack of adoption is not enough to indicate that a tool is not useful as there are many barriers to adoption (e.g., seemingly trivial usability issues can impede usage of a tool)” [203].

Researchers in the software engineering area have drawn from existing work to understand and improve adoption of tools, technologies, and methods. ACRE draws
from ideas of cognitive science to understand, measure, and evaluate the cognitive support of tools [204]. Storey et al. include cognitive support in their evaluation framework of software visualization tools [82]. Sun and Wong apply cognitive theories of human perception, especially Gestalt theory, to evaluate the SHriMP tool and to suggest improvements for it [205]. Lethbridge et al. incorporates elements of the technology acceptance model and diffusion of innovation theory to explain tool adoption. Tilley et al. look at the adoption of research tools through the lenses of Moore’s Crossing the Chasm and Christensen’s The Innovator’s Dilemma [189]. Examples of other applicable theories and models are cognitive fit, consumer behavior theory, technology transfer, SEI’s Technology Transition Practices, and technology readiness level.

Besides reverse engineering, other computer science areas have discussed adoption of their tools and techniques (e.g., product lines [206], open source [207], groupware [208], and functional programming [209]). There are many more examples, including workshops on technology transfer and adoption (e.g., [210]), and journal special issues (e.g., [211]). It is encouraging that a growing number of researchers have started to realize that adoption is an important challenge that needs to be addressed. Unfortunately, these efforts are still immature. Suggestions to improve adoption are often based on guesswork without providing an underlying theory, or apply existing theories without empirical data.

### 2.6 Requirements of Exchange Formats

The discussion, so far, has centered on quality attributes. However, the functional requirements of tools are equally important. Indeed, tool builders have to evaluate the functional requirements of their tools carefully, because a tool can be only useful to its users if it provides the functionality that the users need to fulfill their tasks. In fact, the second-largest complaint in Lethbridge’s survey was missing or wrong mix of tool features (15%) [90]. As a result, Lethbridge and Anquetil require tools to “incorporate all frequently-used facilities and advantages of tools that software engineers already commonly use” [90]. But what are these frequently used facilities?

In contrast to quality attributes which are more component-agnostic, functional requirements are typically tied to a particular component of the reverse engineering tool (Fig. 1). As an example, we will here discuss the functional requirements for exchange formats, which is an approach to realize the tool’s repository component. In other work, we have explored requirements for software visualization tools in more detail [212].

The research community has discussed extensively the functional and nonfunctional requirements for exchange formats. This discussion was started by the realization that a common exchange format would be beneficial for the whole
In the following, we briefly summarize the requirements for exchange formats reported in the literature:

**Graph model:** Wong recommends to “use a graph model with multiple node types, arc types, and attached attributes” [62]. Examples of exchange formats that adhere to this model are RSF, TA, MSE, and GXL.

**Version control:** One of Wong’s data requirements is to “provide version control over schemas and fact bases” [62]. An exchange format does not need to provide version control, but should also not unduly complicate the use of it.

**Textual form:** An exchange format should be textual. This simplifies processing and makes the format human-readable [96, 214]. Riva notes that “a human readable format allows us to navigate the data with simple text editors and eventually repair corrupted files” [135].

**File-based:** There are researchers who say data should be stored as files. Riva made the experience with FAMIX that it is convenient when “all the extracted data are contained in one single file that is easy to archive and transfer” [135].

**Formality:** An exchange format should be well defined and formally documented to “eliminate the possibility of conflicting interpretations of the specification for encoding and decoding model data” [96].

**Composability:** The fact bases of an exchange format should be composable [62]. For example, two RSF files can be simply appended to form a new, syntactically valid RSF file. A related requirement is that the exchange format “should be incremental, so that it is possible to add one subsystem at a time” [41, 119]. Flat formats such as RSF are easier to compose than nested ones such as XML [70].

**Granularity:** The exchange format should “work for several levels of abstraction” such as fine-grained AST-level and coarse-grained architectural-level [119]. Similarly, Koschke et al. state that an intermediate representation “should support different levels of granularity from fine-grained to coarse-grained” [69]. Kamp says, “the repository should support the use of the level of granularity appropriate to the current program comprehension task” [42].

**Neutrality:** The exchange format should be neutral with respect to the stored information and the platform. For example, it should “work for several source languages” and “work for static and dynamic dependencies” [41, 119]. For St-Denis et al., “the neutrality requirement ensures that the model interchange format is independent of user-specific modeling constructs in order to allow a maximum number of model users to share model information” [96].

---

11 Examples of discussion forums are WoSEF (held at ICSE 2000) [214], a WCRE 2000 working session on exchange formats, and Dagstuhl Seminar 01041 on *Interoperability of Reengineering Tools* (held in January 2001).
Incremental loading: Ducasse and Tichelaar state that “incremental loading of information is about the ability to load new entities or additional information for entities that already exist in a model. The reasons for considering incremental loading are resource optimization and the merging of information from different sources” [70].

Naming: Entities in the source model have to be represented in a suitable form in the exchange format. This mapping should be unambiguous (e.g., variables with the same name but in different scopes should be distinguishable from each other [62]). This can be accomplished with unique (but artificial) identifiers or a unique naming scheme [70]. In the FAMIX model, “all the entities have a unique name that is built using precise rules” [135].

Querying: Tichelaar et al. state that “a large portion of reengineering is devoted to the search for information. Therefore it should be easy to query the exchange format. Especially, processing by ‘standard’ file utilities (e.g., grep, sed) and scripting languages (such as Perl, Python) should be easy” [214]. Many exchange formats are associated with dedicated query languages, which have their own requirements [215].

Popularity: Even though not a technical issue, popularity and enthusiastic supporters are an important requirement since an exchange format has to facilitate data exchange between many and diverse (skeptical) stakeholders [41, 96, 119]. In this context, it can be desirable that the exchange format supports industry standards or is a standard itself [214].

Koschke and Sim point out that there are different stakeholders for an exchange format such as tool users and tool builders [213]. The requirements of different stakeholders are not necessarily the same. For instance, a tool user might favor a format that is human-readable, whereas a tool builder is primarily interested in a format that can be parsed easily and efficiently. On the one hand, formality is highly desirable to reduce ambiguity. On the other hand, according to Bosworth “simple, relaxed, sloppily extensible text formats and protocols often work better than complex and efficient binary ones. Because there are no barriers to entry, these are ideal. A bottom-up initiative can quickly form around them and reach a tipping point in terms of adoption” [216]. Perhaps not surprising, the reverse engineering community has primarily focused on technical issues of exchange formats, neglecting to discuss overarching issues such as rationale and economic impacts [217].

2.6.1 Schema

The schema is an important part of the exchange format; consequently, researchers have stated dedicated requirements for it. Several of Wong’s data requirements are targeted at schemas. According to him, a schema should “support aggregation,
inheritance, hierarchy, and constraints” [62]. Another important requirement is extensibility. Wong believes that the schema should be “dynamically extensible” [62]. Extensibility can be achieved with inheritance of schema entities, which is supported by several exchange formats (e.g., TA and FAMIX). Inheritance facilitates extensibility because it allows to add new schema entities in a defined manner [70]. Extensibility is highly desirable to support multilanguage tools. Ferenc et al. say, “the schema should be independent of any parsing technology” [108] and Riva argues for a “separation between data and presentation” [135]. A lesson learned from the Moose tool is to “make your [schema] explicit” [30]. Wong also says that the exchange format should “support introspection of schemas” [62]. This means that a tool should be able to query the schema itself and not just the data. To facilitate introspection, a meta-schema is needed (which is either formally or informally defined). Ducasse and Tichelaar introduce meta–meta-models as an axis in their tool design space [70]. They state that while an existing meta-schema has the benefit that it is already predefined and agreed upon by all tool users, it is less flexible and constrains extensibility.12

2.6.2 Other Domains

Besides reverse engineering, interchange formats are used in many different domains ranging from networking (e.g., ASN.1) and graphics (e.g., JPEG) to business processes (e.g., PSL) and biosciences (e.g., HDF5). It is an interesting question whether different domains have mostly disjoint or mostly overlapping requirements for exchange formats. If the latter is the case, then different domains can learn from each others’ requirements and experiences. There is also the question if a general exchange format such as XML could replace domain-specific ones.

GraphML is a generic XML-based format to represent graphs and as such its key goal is to “represent arbitrary graphs with arbitrary additional data” [218]. GraphML’s designers state as requirements simplicity, generality, extensibility, and robustness. Mendling has looked at exchange format requirements of different domains and found that “the challenges are quite similar across different domains” [217]. Indeed, most of the requirements that he identified apply to the reverse engineering domain: readability, ease of implementation, support of standards, platform independence, efficiency, and free availability. For the schema, he found simplicity in the sense that it is easy to understand, completeness (i.e., to be able to

12 For example, Rigi’s schema is constrained by the fact that it must contain a level arc to model hierarchical graphs. Rigi’s meta-schema is constrained because it allows only nodes, arcs, and attributes with a fixed semantics.
represent all relevant concepts of the domain), generality (i.e., to be applicable in all scenarios that are relevant in the domain), unambiguous, and extensible.

Given the overlap of requirements one may ask why the reverse engineering community has pursued the definition of a dedicated exchange format rather than using an existing one. In fact, researchers have leveraged existing formats such as CDIF and XML. However, while general exchange formats typically offer generality—because they have been designed with an eye on this particular requirements—and often also extensibility, they are suffering in terms of simplicity and readability. Also, they often lack with respect to ease of implementation, which is a concern if no standard library is available.

2.7 Discussion

Elicitation and documentation of requirements is an important activity during software development. However, requirements are often short-cut or deemphasized when developing reverse engineering tools [220]. For example, researchers often rely on “an intuitive notion of what features are beneficial” when developing a software visualizer [24]. The notion of the tool requirements are thus mostly in the minds of the researchers.

While requirements elicitation is a moving target in a research setting, there can be great value in reifying the tool requirements and their rationale, and tracking changes in the requirements as the tool evolves. One approach to ensure that requirements feature more prominently in tool construction is to employ a dedicated process for tool building, which explicitly address the elicitation and evolution of requirements (Section 3). Tool requirements can also inform tool evaluation (Section 4). For example, one can pick suitable requirements for measuring and comparing tools.

Explicitly articulated tool requirements can play an important role in driving research, especially if a community can agree on a set of desirable and/or idealized requirements whose achievement can serve as a visionary goal. For example, Wong states that addressing all of this 23 tool requirements represents “a significant research challenge,” and recommends to “summarize and distill lessons learned from reverse engineering experience to derive requirements for the next generation of tools” [62]. Similarly, Hamou-Lhadj et al. have identified requirements for trace exploration tools and in doing so “have uncovered a number of requirements that raise very interesting research challenges” [67]. The discussed quality attributes

13 Before the advent of XML, researchers have also proposed to use HTML’s meta tags to encode reverse engineering information [220].
reflect the current state of tool requirements and thus can serve as a starting point for future research directions.

In a sense, requirements can drive research from the bottom-up by informing and constraining the tool that gets developed. This approach can complement the more prominent approach of top-down research, which first constructs a tool—based on more or less vague hypotheses or notions—and then tries to find evidence that the tool is useful indeed.

However, most sources of the requirements that we found are based on personal experiences and observations that are inferred, gathered, and reported in an unsystematic manner. In our literature survey, we found that requirements are often discussed without citing related work or disclosing where else the requirement may have come from, or mentioned without giving a detailed explanation or rationalization. Also, researchers do not discuss the applicable scope of a stated requirement. For instance, is the requirement believed to apply to all software systems, to the domain of software or reverse engineering, to certain kinds of tools, or to one tool in particular? Lastly, a requirement is often discussed in isolation, without addressing dependencies or trade-offs with other requirements. Perhaps surprisingly, only Lethbridge and Anquetil explicitly separate their requirements into functional and nonfunctional ones [90]. Such a distinction makes it easier to judge the scope and applicability of requirements. Furthermore, they explicitly identify requirements that their tool does not address (yet), thus making their tool’s limitations more explicit.

In addition to the requirements gathered in an ad hoc manner, more formal techniques are needed that are grounded in studying of actual users in a realistic setting; this trend can be already observed in software visualization research [16, 221].

Whereas tool requirements seem relatively stable, they are not fixed. Changes in development and maintenance processes and in the characteristics of software need to be reflected in the requirements for reverse engineering tools. Thus, researchers should continuously reevaluate their assumptions. For example, a previously neglected requirement that is starting to receive more attention by researchers is collaboration and multiuser support [17]. This need was already articulated in 1990 for software development tools [221], and reiterated later on for the reverse engineering domain (e.g., Rugaber in 1996 [48] and Storey in 2005 [203]). In contrast, Bellay and Gall argue that multiuser support in reverse engineering tools is “not of such importance as in development tools because the application normally does not change and only one person may reverse engineer it” [72]. However, this view seems dated considering the large amount of commercial and open source software that is developed and maintained in a distributed, collaborative work-style. Indeed, Koschke states that “large maintenance and reverse engineering projects require
team-work and, hence, visualizations need to support multiple users that may work at the same system at the same time at possibly different locations” [222].

The emerging importance of this requirement is also reflected in Storey et al.’s software visualizer framework, which has a dimension to distinguish the team size that a particular tool targets [82]. Examples of tools that support teams in (near) real-time are the Jazz collaborative development environment [223] and the Churrasco collaborative software evolution analysis tool [224]. There are also commercial IDEs emerging that emphasize collaboration such as IBM’s Jazz (jazz.net).

3. Tool Construction Lens

The building of tools is an important part of many research efforts, especially in the reverse engineering domain. The tangible results of reverse engineering research is often embodied in tools, for instance, as a reference or proof-of-concept implementation.

Even though tool building is a popular technique to validate research, it is neither simple nor cheap to accomplish. Tool building is costly, requiring significant resources. This is especially the case if the tool has to be robust enough to be used in (industrial) user studies. Nierstrasz et al., who have developed the well-known Moose tool [30], say that

“in the end, the research process is not about building tools, but about exploring ideas. In the context of reengineering research, however, one must build tools to explore ideas. Crafting a tool requires engineering expertise and effort, which consumes valuable research resources.”

Sometimes a significant part of the resources of an entire research group are devoted to building, evaluating, and improving a tool. Given the significant cost associated with tool building, researchers should explore how tools can be constructed in effective and efficient manner. Wasted resources for tool building translate to less research output, slower iteration and communication of research, and reduced opportunities for adoption and transition of tools and techniques to industry.

Traditional tool building that constructs everything from scratch offers the most flexibility since almost all functionality is implemented from scratch and under the full control of the developer. On the downside, this approach is costly (e.g., in terms of longer development time) and can result in idiosyncratic tools that are difficult to learn and use. On the other hand, as for any software development project, there is the desire in tool construction to reuse. This point is articulated by Shaw as follows [225]:
“Most applications devote less than 10% of their code to the overt function of the system; the other 90% goes into system or administrative code: input and output; user interfaces, text editing, basic graphics, and standard dialogs; communication; data validation and audit trails; basic definitions for the domain such as mathematical or statistical libraries; and so on. It would be very desirable to compose the 90% from standard parts.”

There is a desire among researchers to build upon existing functionality and infrastructure. Participants of a tool building workshop for reverse engineering articulated that they “were tired of writing parsers/analyzers and wanted to avoid writing another one, in particular a C++ parser” [213]. This desire is driven by the realization that in tool building comparably little effort is spent on the research contribution, and that a significant effort is needed for the supporting infrastructure. Researchers would rather work on core activities that advance research than being tied up in lower level plumbing.

There are many different forms of reuse ranging from design and code scavenging to very high-level languages [226]. However, one can distinguish between two major techniques to achieve reuse: compositional and generative reuse. These two reuse techniques correspond to component-based (Section 3.2) and model-driven (Section 3.3) tool development, respectively. Before turning to these two techniques, we discuss an overarching issue, namely architectures for reverse and software engineering tools (Section 3.1).

3.1 Tool Architecture

The importance of architecture on software systems is firmly established [64]—tool construction is no exception in this respect. Importantly, tool architecture interacts with tool requirements (Section 2). The required quality attributes of a system often drive the decision to select a particular architecture or architectural style [63]. Conversely, the chosen architecture of a software system has a profound impact on its quality attributes [227].

When building a tool, fundamental questions of the architecture are how to separate the tool’s overall functionality into functional units, and how to interface these units with each other. For example, Fig. 1 shows a generic, high-level reverse engineering architecture, consisting of four components: extractors, analyzers, visualizers, and a repository. This architecture exposes the conceptual, or logical, structure of the software, in which the components (or units) are abstractions of the systems’ functional requirements and are related by the shares-data-with relation [64].
One can also look at the introduced conceptual architecture as a *reference model* for reverse engineering tools [64, p. 25]. A reference model emerges through increasing consensus of a research community and thus indicates a maturation of the research domain. The domain of compilers provides an example of a widely known reference model with the following functional units [11]: lexical analyzer, syntax analyzer, semantic analyzer, intermediate code generator, code optimizer, code generator, symbol-table manager, and error handler. Reference models in reverse engineering are important because they provide a frame of reference to guide researchers in understanding and implementing tools in the reverse engineering domain.

The concrete architecture of a reverse engineering tool does not necessarily coincide with the conceptual tool architecture presented above. At one extreme, one could imagine a monolithic architecture that groups extraction, analysis, and visualization into a single component (without any interfaces). In practice, one can distinguish three different architectural approaches for tool building [87, 110, 228]:

*Data-driven*: In this approach, the tool’s (functional) units are rather loosely coupled and communicate via an agreed-upon data model. Data communication can be accomplished with a repository or an exchange/document format.

A typical example of a data-driven integration framework is the Unix *pipe* mechanism, which composes new programs from existing ones (also called *filters*) by connecting the textual output of one program to the input of another. In this case, the components are executable programs and the data model are textual streams grouped into lines and lines grouped into fields via special characters such as whitespaces or colons.

Ciao/CIA is an example of a reverse engineering tool that follows this approach, which the authors call repository-based reverse engineering [22]. A typical use of Ciao is a pipeline that consists of a sequence of query commands followed by a visualization command. Figure 6 shows the typical constituents of such a pipeline [230, p. 188].

*Control-driven*: In this approach, tool components are more tightly coupled because they are based on an infrastructure that allows them to pass messages among each other (e.g., via a message server). The underlying communication infrastructure can be provided by the operating system or by more sophisticated wiring standards such as CORBA, COM, or JavaBeans.

---

14 Salus summarizes the philosophy of Unix as follows: (1) write programs that do one thing and do it well; (2) write programs to work together; and (3) write programs that handle text streams, because that is a universal interface [230, p. 53].
For example, the SHriMP tool’s core architecture now consists of a number of JavaBean components (Fig. 7) [231]. All components know the abstract concepts of the data model, consisting of entities and relationships. Most important are the Persistent Storage, Data, and Display components. The Persistent Storage Bean reads and writes data (i.e., entities and relations) to and from a repository. It passes the data to a Data Bean, which constructs the data’s in-memory representation. The Data Bean’s interface allows the data to be queried and manipulated. SHriMP already provides implementations of Persistent Storage Beans that read and write RSF and GXL. A generic Data Bean implementation is also provided. Entities and relationships in the Data Bean are visualized as nodes and arcs of a directed, hierarchical graph in the Display Bean. Examples of other components are Search Bean (to search through information associated with entities and relationships) and Filter Bean (to determine whether certain entities and relationships should be hidden). All components are independent from each other and can be replaced with other compatible beans.

A variant of the control-driven approach restricts communication patterns by having a single master component that controls a number of slave components. An example is the use of Emacs as a central component that invokes other services such as compiler, make, lint, and debugger. Many IDEs have been developed on top of Emacs; in fact, XEmacs (then called Lucid Emacs) was created to realize Lucid’s Energize C/C++ development environment.
**Presentation-driven:** This approach refers to a seamless interoperation at the user-interface level. Components are tightly integrated and have a common look-and-feel. Examples of technologies are compound documents such as OLE and OpenDoc. In order to achieve such tight integration some kind of dedicated infrastructure is necessary.

IDEs are a prime example of presentation-driven development tools. Since these IDEs are often customizable based on a plug-in mechanism (e.g., Eclipse, IBM VisualAge, and Microsoft Visual Studio), many reverse engineering tools are implemented as IDE integrations. However, there are also examples of presentation-driven reverse engineering tools that are not based on IDEs (e.g., SolidFX [232] and Columbus) or leverage Web browsers for the user interface (e.g., Churrasco and SPO [169]).

VizzAnalyzer is a plug-in-based tool framework that allows to write analyses and visualization plug-ins [116]. The plug-ins are communicating by manipulating a graph data structure. Figure 8 shows the framework’s architecture with two analysis plug-ins (Recorder and Analyzer) and two visualization plug-ins (yEd and Vizz3d). Two of these plug-ins enable to add plug-ins themselves (e.g., to support new layouts for Vizz3d).

The three approaches introduced above are usually inclusive. The presentation-driven approach needs functionality to pass control and data in order to achieve seamless integration of components; and the control-driven approach needs to pass data along.

In practice, tools combine the above approaches. Reverse engineering tools often decouple the extractor component from the rest of the tool with a data-driven approach involving an exchange format.\(^{15}\) Analyses and visualization are often tightly integrated with a control-driven and/or presentation-driven approach.

The SolidFX tool is based on presentation-driven integration and thus has the look-and-feel of an IDE [232]. The individual components for fact extraction, analysis, and visualization communicate via a central fact database. In order to accommodate third-party tools, SolidFX offers control-integration with a query API to the fact database and data-integration based on various formats (XML for ASTs, XMI for UML diagrams, SQL for metrics, Dot and VCG for graphs). Interestingly, SolidFX’s developers previously used the same components of their tool in a loosely

\(^{15}\) Historically, reverse engineering tools developed in the late 1980s and early 1990s, supported only a single programming language (e.g., MasterScope for Lisp, FAST for Fortran, and Cscope for C [22]). Since these tools consisted of a single extractor, there was often a tight coupling between the extractor and the rest of the system [22, 234]. This rather tight coupling was inflexible and made it difficult or impossible to support additional languages.
coupled manner without having a presentation-driven IDE. They relate their experiences of these two different architectures as follows:

“For the parsing phase, the [IDE] was not much more effective—a simple text makefile-like project was sufficient. However, for the exploration phase, the [IDE] and its tight tool integration were massively more productive than using the same tools standalone, connected by little scripts and data files”

Telea and Voinea [232].

### 3.2 Component-Based Tool Development

“Programs these days are like any other assemblage—films, language, music, art, architecture, writing, academic papers even—a careful collection of preexisting and new components.”

Biddle, Martin, and Noble [234]

Component-based development (CBD) is a widely applied and highly successful approach for developing software systems [235]. Consequently, researchers have started to adopt the idea of CBD for developing their research tools. In the following, we refer to this approach to tool building—which reuses existing, prepackaged functionality—as *component-based tool development* (CBTD). As opposed to
traditional tool building, which is characterized by a high degree of custom code and little reuse, CBTD leverages software components as building blocks. For example, Fig. 9 shows the reuse of components in the CIA tool [230, p. 178]. External components are shown as diamonds, while tool-internal components are shown as ovals.

A key driver for CBTD is the reuse of existing code in the form of components. If a component is carefully selected, its existing functionality can cover a significant part of the tool functionality. As a result, significantly less code needs to be written and subsequently maintained. Reiss has implemented a software development environment, Desert, based on FrameMaker. He reports that for Desert’s editor, “FrameMaker provides many of the baseline features we needed,” specifically “it displays both pictures and text” and “it can display high-quality program views” [236]. Similarly, the authors of SLEUTH say that “FrameMaker provides an effective starting point for our prototype. Many of the basic features necessary for document creation and editing are provided, allowing effort to be concentrated on more specialized features” [237].

![Diagram](image-url)

**Fig. 9.** Reuse of components in the CIA tool.
In principle, all kinds of components are candidates for CBTD. Thus, the definition of a component in the context of CBTD should be rather broad such as the one provided by Czarnecki and Eisenecker, who define (software) components as “building blocks from which different software systems can be composed” [238]. Examples of applicable components for CBTD are

—IDEs (e.g., Eclipse, Microsoft Visual Studio, IBM Visual Age, TogetherSoft Together).

—Commercial off-the-shelf products (e.g., Microsoft Office and Internet Explorer) and their open-source counterparts (e.g., OpenOffice and Firefox).\(^\text{16}\)

—Components based on Sun’s JavaBeans and Java Enterprise Beans (EJB), and Microsoft’s Component Object Model (COM) and Distributed COM (DCOM).

—Object-oriented frameworks to realize GUls (e.g., Java’s Abstract Windows Toolkit (AWT) and Swing, Eclipse’s Standard Widget Toolkit (SWT), and the Microsoft Foundation Classes (MFC)).

—Unix tools (e.g., awk, sed, and grep), text editors (e.g., Emacs), and scripting languages (e.g., Perl).

—Libraries for domains such as standard data structures (e.g., C++ Standard Template Library (STL)), graph data structures (e.g., LEDA), as well as library collections (e.g., from AT&T [230]).

The fact that CBTD is widely applied is exemplified by many tools that leverage components to implement functionalities for fact extraction and visualization [83, 235]. Table II gives examples of software and reverse engineering tools that base their visualizations on components. This list represent only a smaller sample, but illustrates the broad range of components that have been leveraged. Finding suitable components for implementing a tool is important because the characteristics of the component will determine the whole tool building effort.

The use of components fundamentally changes the development of tools and has unique benefits and drawbacks. This fact is often not realized by tool builders. Important questions for CBTD that have to be addressed by researchers are: What are good candidate components for CBTD given a tool’s application domain, its required functionality, its desired quality attributes, and its envisioned users? What impact has a certain component on the overall tool architecture, on the ramp-up time and implementation effort for the tool, and on the further maintenance and evolution of the tool? What characteristics of the components and the architecture minimize risks and maximize effects? And so on.

\(^{16}\) Both commercial and open-source products are summarized as off-the-self (OTS) products.
The use of components also has an impact on the tool’s quality attributes. In the following, we briefly give examples how CBTD can impact the five quality attributes introduced in Section 2:

**Scalability:** Scalability is especially a concern for visualizations. Generally, components such as Visio and PowerPoint are able to handle the rendering of dozens of graphical objects. This is sufficient for small to medium software graphs. However, the rendering of larger graphs can cause problems in terms of screen updating and rendering speed. For example, the developers of the Nova tool state that their tool’s “response time is largely dependent on package performance. . . . We spent significant effort understanding Visio’s drawing speed and exploring ways to use Visio that would give us better drawing performance” [243]. Also, different components and different versions of the same components can differ in their performance characteristics. For example, in Visio 4.1 the drawing speed of objects

---

**Table II**  
**EXAMPLES OF COMPONENTS TO BUILD GRAPH-BASED VISUALIZERS [235]**

<table>
<thead>
<tr>
<th>Component type</th>
<th>Host component</th>
<th>Tool-building examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>OTS products</td>
<td>Office/Visio</td>
<td>REVisio [239]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Huang et al. [240]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Nimeta [241]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>VDE [242]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Galileo/Nova [244–246]</td>
</tr>
<tr>
<td>FrameMaker</td>
<td>SLEUTH [237]</td>
<td>Desert [246]</td>
</tr>
<tr>
<td>Web browsers</td>
<td>REPortal [170]</td>
<td>Software Bookshelf [36]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>TypeExplorer [68]</td>
</tr>
<tr>
<td>IDEs</td>
<td>Eclipse</td>
<td>SHriMP [247]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MARPLE [248]</td>
</tr>
<tr>
<td>Rational Rose</td>
<td>Rose/Architect [249]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>UML/Analyzer [250]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Berenbach [251]</td>
<td></td>
</tr>
<tr>
<td>Together</td>
<td>JaVis [252]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>GoVisual [253]</td>
<td></td>
</tr>
<tr>
<td>Tools</td>
<td>AT&amp;T Graphviz</td>
<td>Reflexion model viewer [85]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ReWeb [254]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>CANTO [255]</td>
</tr>
<tr>
<td>SVG</td>
<td>SVG graph editor [256]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>SPO [257]</td>
<td></td>
</tr>
<tr>
<td>Libraries</td>
<td>OpenGL</td>
<td>Extravis [258]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>CodeCity [259]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SolidFX [232]</td>
</tr>
</tbody>
</table>
increases quadratically with the number of shapes already present on the page; in contrast, Visio 5.0’s behavior is linear [243].

Interoperability: Components can have a wide range of interoperability mechanisms and make implicit assumptions about the way they interact with other components. Thus, it is unlikely that two independently developed components that do not adhere to the same component model will interoperate seamlessly out-of-the-box. This phenomenon is known as architectural mismatch [260]. Interoperability is less of an issue if the tool is based on a single component only. Also, architectural mismatch is less of a problem for data integration compared to control and presentation integration. Wrappers and bridges can be used to integrate heterogeneous components; however, this solution can turn out to be brittle. The SHriMP tool is using an AWT-SWT bridge to integrate with Eclipse, but this approach has several undesirable effects (e.g., risk of deadlocks and lost keyboard events) [261].

Customization: Customizations of components are limited by the functionality that the API and/or scripting language provides. Based on their customization experiences with several commercial components, Egyed and Balzer say that “sadly, ‘real world’ COTS tools are often only partially accessible and customizable, greatly limiting their reuse” [262]. Reiss reports the following limitation in the FrameMaker API: “While FrameMaker notified the API that a command was over, it did not provide any information about what the command did or what was changed” [246]. As a consequence, customizations of components without the use of source code modifications always run the risk that certain desirable tool functionalities cannot be realized at all, or with less fidelity. Nova’s developers had to cope with undocumented restrictions of Microsoft Office components: “Due to [the] lack of package documentation, we discovered certain limitations of the packages only after working with them extensively. In some cases, the limitations were quite serious” [243]. For example, they found out that “the maximum number of shapes Visio 4.0 could store on a single drawing page was approximately 5400.”

Usability: Familiarity of the target users with a certain product improves usability and helps adoptability. Reiss explains the decision to use FrameMaker as a component with the fact that “we wanted an editor that programmers would actually use. This meant that the base editor must be familiar to them, preferably one they were using already” [236]. Furthermore, popular components can reduce the need for documentation: “Because such components are also popular stand-alone applications, users are often already familiar with them, and much of the application documentation applies when the application is used as a component” [244]. Tool implementors can also leverage the existing infrastructure of the help system to seamlessly provide documentation for their tool. For example, Microsoft Office’s Assistant can be customized in Visual Basic to provide tool-specific help.
Adoptability: The market share of components can improve a tool’s adoptability. The popularity of PowerPoint was an important factor for the developers of VDE: “Visio is a commercial product with many similarities to PowerPoint . . . It might provide a better technical fit to our needs, but lacks PowerPoint’s enormous user base” [242].

3.3 Model-Driven Tool Development

“Truly model-driven development uses automated transformations in a manner similar to the way a pure coding approach uses compilers.”

Kelly and Tolvanen [263]

CBTD reuses existing components, integrating and customizing them so that they fit together. In contrast, model-driven tool development (MDTD) generates code from a domain-specific, higher level specification [238, 264]. Thus, MDTD can be seen as a form of generative reuse. The reusable assets in these techniques are less intuitive and less palpable compared to compositional reuse because they consist of patterns for code generation and transformations.

The idea of MDTD derives from generative development approaches such as domain-specific modeling (DSM), model-driven development (MDD), generative programming, and Microsoft’s software factories [15]. Traditionally, software development is code-driven. As a result, the code and its models—if models do exist in the first place—are disconnected in the sense that there is limited traceability and synchronization [263]. In contrast, model-driven software development employs models as the primary vehicle to express software. From the models a code generator produces executable code. The code generator (or model transformer) embodies the reusable part that can be (re)applied to different models. Typically, the code generator will not generate the complete application’s code, but rather fill in missing pieces in an existing code base (e.g., instantiation of an object-oriented framework). As a result, models are executable in the same sense that source code is an executable specification.

Rugaber and Stirewalt propose to apply the model-driven approach to the reverse engineering process [265]. Reverse engineering of a system would produce models that can act as formal specifications of the system under study. From these models, supported by a code generator, “another version of the original system” can be produced.

Examples of generative techniques and concepts that can be applied for MDTD are
—Traditional scanner and parser generators such as lex and yacc.
—Meta-compilation systems for generating language-based tools such as LISA and JastAdd (both based on attribute grammars).
—Meta-CASE tools, which provide facilities for specifying and generating CASE and software development environments (e.g., IPSEN, MetaEdit+, and the Generic Modeling Environment (GME)).

—Application generators such as Neighbors’ Draco and Batory’s GenVoca systems [264].

—Executable meta-modeling (e.g., Kermeta) and modeling frameworks (e.g., Eclipse EMF/JET).

—Meta-programming, and source transformation and rewrite systems (e.g., DMS, Stratego, ASF+SDF, Rascal, and TXL).

—Fourth-generation languages (4GLs) such as (relational) query languages and report generators.

—Generative programming techniques such as C++ template metaprogramming and aspect-oriented programming [238].

In practice, there is an overlap between the capabilities of the above techniques and the techniques are not clearly delineated. What they have in common is a domain-specific (modeling) language. These languages allow to specify the solution using problem domain concepts. They are often declarative (allowing the user to express what is to be done rather than how it is to be done) and based on different underlying paradigms or concepts (e.g., grammars, algebraic specifications, and first-order logic).

While CBTD is widely applied, to our knowledge there are only few examples of MDTD in the reverse engineering domain. Favre emphasizes the importance that meta-models have in software construction [266]. Each software artifact (e.g., source code, test case, bug tracking entry, database record, and XML document) is, in fact, a model that conforms to a meta-model. The meta-model is not necessarily explicit, but may be implicitly encoded in the model’s operation.17 Metaware is a software that operates at the level of meta-models (e.g., compilers, IDEs, testing frameworks, databases management, and XSLT). Explicit—or reified—meta-models that are easily processable by metaware are an important step toward MDTD. Favre has implemented the GSEE, which is a meta-model-driven tool that interprets a meta-model specification and customizes the tool accordingly [123, 136].

The Moose reverse engineering environment is based on an executable meta-model (EMOF 2.0 compliant) implemented in Smalltalk [267, 268]. This approach allows it to not only specify the meta-model, but also to attach behavior to it in the form of Smalltalk code. The implementors of Moose justify this approach as

17 To give an example, an XML file may have an explicit meta-model in the form of a DTD or XML Schema. If an explicit form is missing the schema may be implicitly encoded within the tools that read the XML file and the tools’ expectation of well-formed input.
follows: “we felt the need to meta-describe our environment to enable us to be more efficient building new tools for our reengineering research. Using meta-modeling was just a means to introduce more flexibility and extensibility in our tools” [267]. The meta-model is implemented in a dedicated framework, called Fame, and leverages Smalltalk’s reflection capabilities, class extension, and pragmas. With Fame, functionality such as serialization and UI behavior can be realized in a generic manner by operating on the meta-level. Note that Fame is based on an interpretative approach (i.e., no code is generated from the model).

In contrast, the developers of the REforDI tool use “reusable frameworks, formal specifications, and generators to reduce the implementation effort” of their tools [269]. They explicitly motivate this decision with the observation that “hand-coding of re-engineering tools is a painstaking business.” The REforDI tool uses TXL and PROGRES specifications for design and code transformation as well as graphical, interactive functionality of the tool. According to the authors, the tool requires less than 1000 lines of handwritten C code. However, they caution that “the concept of graph grammars and the PROGRES language might be a bit difficult to learn for newcomers” [269].

Another example of MDTD is model-driven visualization (MDV) [270, 271]. MDV’s vision is to leverage model-driven design for creating software visualization so that “researchers and tool designers will be able to spend more time designing and evaluating their tools and less time building them” [270]. MDV’s reference architecture is depicted in Fig. 10. Both software and visualizations conform to a software meta-model and various visualization meta-models, respectively. The visualization meta-models cover domains such as graph-based, tree-based, and chart-based visualizations. At the meta-model level, transformations are applied that map from the software meta-model to visualization meta-models. A transformation encodes the abstract and visualize/synthesize activities of the reverse engineering process (Section 1.1).

The authors have instantiated the reference architecture in an Eclipse-based framework. The source meta-model adheres to the Dagstuhl Middle Model, the visualization meta-models are described with Emfatic/EMF, and the transformations are written with the Atlas Transformation Language (ATL). Figure 11 gives a toy example that shows how entities from a higher level source code meta-model (with File, Function, and Call entities) could be mapped to a graph-based visualizer meta-model (with Graph, Node, and Edge entities, respectively) [266].

3.4 Discussion

Researchers are pursing CBTD and MDTD because they are hoping to become more productive in tool development. Increased productivity in tool building frees resources for other research activities and leads to a faster innovation cycle.
Fig. 10. MDV reference architecture [270].

Fig. 11. Mappings between source and visualization meta-models [266].
These approaches to tool building are based on existing techniques—CBTD relates to CBD and MDTD relates to MDD—that are also used for the construction of software in general. This is desirable because they can leverage existing methodologies, technologies, tools, and experiences. However, since the academic tool-building domain has distinctive characteristics it appears that the existing, generic approaches should not be applied blindly. In order to establish best practices for tool building researchers have to elevate this lens to a first-order research topic.

A yardstick for success of the tool construction lens is the generation of results that make tool building in academia more predictable and effective.

This means, for example, establishing workshops about tool building and publishing issues about tool building. In fact, there are encouraging signs in that direction. For example, Coppit and Sullivan have introduced an approach to tool construction that they call package-oriented programming (POP). In a sense, POP is an instantiation of CBTD that focuses on “the use of multiple, architecturally compatible, mass-market packages as large components” [272]. The use of multiple components is motivated by the observation that many tools require functionality that needs to be drawn from several independent domains such as text editing (e.g., provided by Word) and graph editing (e.g., provided by Visio). POP proposes to use components that are architecturally compatible to simplify integration and to minimize architectural mismatch.

Furthermore, reverse engineering researchers of successful tools have published about their experiences. For example, Lanza describes his experiences with the CodeCrawler software visualizer [273], discussing CodeCrawler’s architecture (composed of three subsystems: core, meta-model, and visualization engine), the visualization engine (realized by extending the HotDraw framework), and desirable interactive mechanisms for usability. Furthermore, he distills lessons learned for all of the discussed issues. He observes that “to our knowledge there is no explicit work about the implementation and architecture of reverse engineering tools, and more specifically about software visualization tools.” Guéhéneuc describes his use of design patterns and Java language idioms when constructing the Ptidej tool suite [274]. In addition, it would be highly desirable to identify lessons learned that generalize over individual tool building experiences. Currently, there are few examples of researchers that have published such lessons learned (e.g., [169, 275, 276]).

Research is also needed in technological issues. For example, an important topic is effective tool integration. Wuys and Ducasse describe a tool integration framework based on Smalltalk and how it is used in the StarBrowser [277]. Another example is the push to establish a discipline for software that depends on grammars (so-called grammarware) by Klint et al. [278]. It is certainly the case that reverse
engineering tools are grammarware. The development of parsers for reverse engineering is characterized by *ad hoc* approaches rather than engineering. Klint et al. provide a first step toward grammarware engineering by describing a number of principles that should be followed, a life cycle for grammarware, and a list of research challenges.

Process is a neglected area with respect to tool building. It seems that few academic tool building efforts make use of an explicit process. This is hard to justify, because any software should be constructed based on a process. However, a process has to take care not to stifle unnecessarily the creative elements in research. As a first step, we have proposed a dedicated process framework for tool building in academia [279]. This framework is lightweight and makes allowance for the diversity of academic research projects (e.g., tool requirements, degree of technical uncertainty, complexity and size of the problem, and number and expertise of the development team).

Besides individual efforts, there needs to be a recognition of the tool building lens by the research community. One forum for researchers to meet and discuss tool building issues is the *International Workshop on Academic Software Development Tools and Techniques* (WASDeTT) [280, 281]. WASDeTT was held twice in 2008 and addresses topics such as

—Language-independent tools: How can we build tools that work across multiple languages?
—Tool building in an industrial context: How to build tools that get adopted by industry?
—Data interoperability among tools: How to exchange data between tools and how to process this data?
—Maturation of tools: How to grow a tool from an early prototype into a mature tool or framework?
—Tool building methodology: How—and to what degree—can we adopt established software engineering techniques for building research tools?
—Tool building in teams: How to build tools in larger—and possibly distributed—teams?
—Tool implementation language: How does the choice of a programming language impact the building of a tool, its usability, and the context in which the tool can be applied?

Some instances of WASDeTT are coupled with special issues on *Experimental Software and Toolkits* (EST) [282]. In fact, the organizers of the first WASDeTT say that “one important goal of this workshop series is to enable researchers to publish about their tools so that they can get scientific credit for their tool building efforts” [280].
4. Tool Evaluation Lens

“Evaluate the effectiveness of reverse engineering tools and techniques through empirical studies.”

Wong [62, Requirement 23]

It is not sufficient to build a reverse engineering tool for its own sake. The effectiveness of the tool has to be evaluated as well in some form. This can be accomplished with empirical research, which is a field that studies real-world phenomenon. In the context of tools this means to study how tools are used by certain subjects and how effective the tools are in accomplishing certain tasks on certain kinds of systems. In software engineering, the emphasis has been on development of new technologies rather than on evaluation and comparison of the effects of these technologies [283]. Similarly, reverse engineering tends to focus on the building of new tools, rather than evaluating these tools. However, both tool building and tool evaluation are need to establish reverse engineering as a science.

A tool evaluation has the following components (or treatment factors) [284]: (1) the tool under study, (2) a subject system that is applied on the tool, (3) tasks to be performed on the subject system with the tool, and (4) the users that operate the tool. Generally speaking, an empirical study involves the following steps [283]: (1) formulating a research question, (2) designing a study, (3) gathering data, and (4) analyzing and interpreting the data. Empirical studies of tools can be grouped into the following approaches:

Case studies: A case study investigates the tool within a real-life context. As a result, a case study often has little control over the evaluation setting. Also, the tasks that are performed with the tool are typically not well described because reverse engineering tools are typically used in an exploratory style and on an as-needed bases during software development.

According to Sjøberg et al., “case studies are particularly important for the industrial evaluation of software engineering methods and tools” [283]. Case studies are perhaps the most popular approach to evaluate reverse engineering tools. However, most case studies are rather weak in the sense that the tool is applied to a smaller subject system and that the users of the tool are its developers. Few case studies are conducted within an industrial context. In the ideal case, the “tool under

---

18 We use the term case study rather loosely guided by what researches in reverse engineering consider to be a case study in their publications. Many of these “case studies” may not actually conform to a more rigorous definition. Sjøberg et al. report on a literature study where they found that 58% of case study papers did not meet their definition of evaluative case study [284].
investigation is tried out on a real project using the standard project development procedures of the evaluating organization” [285].

**Experiments:** Experiments investigate a tool within a controlled environment with the goal to obtain certain measurements to test a hypothesis or theory. In an experiment, users of the tool can be selected and assigned to groups based on their characteristics. The tasks that are performed with the tool tend to be small, but relatively well defined. Since experiments use controlled treatment factors they are replicable and exhibit little bias.

There are very little experiments that evaluate reverse engineering tools. Storey et al. have conducted a user experiment to evaluate the effectiveness of the user interfaces of three different tools: Rigi (which uses a multiple windows approach), SHriMP views (single window), and Unix standard tools (grep and vi) [102]. The experiment measured correctness of performing a number of reverse engineering tasks and the time to complete the tasks. This experiment showed that Unix standard tools were the least effective.

Panas and Staron have conducted an experiment involving CBTD in which they used *ad hoc* composition and a framework-based approach (VizzAnalyzer) to build a reverse engineering tool [286]. They use the Goal-Question-Metric approach to answer questions about the effectiveness of tool construction and the tool quality. The experiment showed that framework customization is superior to *ad hoc* composition.

**Benchmarking:** Benchmarking is often associated with performance comparisons such as the well-known SPEC and TPC benchmarks. However, benchmarking also has a broader meaning that covers evaluation approaches that use a well-defined task sample associated with a performance measure so that a comparison among alternatives (e.g., different tools) is possible.

Benchmarking has characteristics from both case studies and experiments. They have qualitative and quantitative elements [284]. Benchmarking allows a direct comparison of results and has built-in replication. On the other hand, there is little control over the users of the tools and how they apply it. Consequently, benchmarks are preferable “if the tool undertakes automatic transformation of data with little or no interaction with the tool user (e.g., a speech synthesizer, or a compiler)” [285].

Sim has developed the CppETS benchmark to measure the performance of C++ fact extractors [77]. The task sample is structured into accuracy and robustness tests. Accuracy checks whether an extractor is able to produce correct facts for preprocessing directives and C++ entities such as variables, functions, and exceptions. Robustness tests issues such as missing header files, C++ dialects, and embedded languages. The benchmark was applied to four extractors (Ccia, cppx, Rigi’s C++ parser, and TkSee/SN). Figure 12 shows the scores of the extractors for both kinds of tests. A drawback of CppETS is that scoring involves a significant amount of manual
labor. In the best case, a benchmark is automated so that it can be easily rerun or applied to a different tool.

Perhaps as a result of Sim’s efforts [287], the reverse engineering community has proposed a number of benchmarks (e.g., for software evolution [288], clone detection [289, 290], design pattern detection [291], and recommender systems [292]). Bellon et al. have evaluated clone detectors with the help of a benchmark [290]. All necessary information is made available so that the benchmark can be replicated. They hope that “benchmark evaluation becomes a standard procedure for every new clone detector” [290].

Feature analysis: A feature analysis is based on a number of criteria that users have for a particular reverse engineering activity and mapping those criteria to features that a tool should have in order to support the activity [285]. The criteria and features could be based on personal opinion or synthesized by means such as questionnaires or (systematic) literature reviews. Often the first step is omitted or not explicitly identified. As a result, a feature analysis is then focused on how well a tool meets a given number of desired features.

Researchers often use feature analysis to evaluate a given tool. For example, the developers of Sextant evaluate their tool with five functional requirements. Feature analysis is also popular to compare different tools among each other. For example, Bellay and Gall have compared the capabilities of four reverse engineering tools (i.e., Refine/C, Imagix 4D, Rigi, and SNIFF+) in terms of their general capabilities,

![Graph showing the score of C++ fact extractors in the CppETS benchmark](image-url)

**Fig. 12.** Score of C++ fact extractors in the CppETS benchmark [77].
There are also a number of other tool comparisons and comparative frameworks as already described in Section 2. For example, Guéhéneuc et al. introduce a comparison framework for design recovery and apply it to two tools [78]. The framework addresses eight concerns: context, intent, users, input, technique, output, implementation, and tool. Some tool comparisons incorporate benchmarks, but feature analysis is mostly qualitative.

**Structured tool demonstration and other challenges:** A structured tool demonstration is “a hybrid evaluation technique that combines elements from experiments, case studies, and technology demonstrations” [293]. Such a demonstration was held as part of a CASCON workshop where different teams had to perform certain reverse engineering tasks with their tool on the same system in a live setting. Thus, a demonstration covers the entire reverse engineering experience in a day. Each tool is assigned an impartial observer who records how the tools are used to solve the tasks and who acts as “apprentice,” trying to become proficient in the tool. The organizers of the event emphasize that the purpose of such a demonstration is not to establish a ranking but rather to give the participants insights into their own tools. They pose the thesis that

*Fig. 13. Feature analysis of the analyzer capabilities of four tools [73].*
“a structured demo provides a lot of insight for tool designers into their own tools and allows them to directly compare their tool capabilities with other tools and learn about future tool extensions”

Sim et al. [151].

In a follow-up study, two additional tools (GUPRO and Bauhaus) performed the same tasks on the same subject system, but not live [151].

Other more informal tool challenges have been conducted as well. These challenges have less control over the treatment factors which means that they rely on personal interpretation, and do not allow comparability or reproducibility. The perhaps first event of this kind was a call to test different tools on an industrial legacy system [294]. The organizer expected 20–30 participants and results were presented in a session at CSMR 1998. In another event, eight tools did participate in the analysis of the SORTIE legacy system (30,000 lines of Borland C++) [295]. The teams had the task to rearchitect the system and to submit a report. Results were presented at WCRE 2001.

The working conference on Mining Software Repositories (MSR) has established the Mining Challenge since 2006. The challenge focuses on a particular software system (e.g., Eclipse) but has no fixed task sample nor does it prescribe a task. Similarly, VISSOFT 2007 did feature a Tool Demo Challenge for visualization tools to perform a number of suggested tasks (e.g., architecture, source code, or evolution analysis) on Azureus or GCC. Three tools did participate (CGA Call Graph Analyzer, Rigi, and VERSO) and submitted short reports.

Empirical research plays a key role for two interrelated activities: the evaluation of an existing tool and the generation of theories that inform the construction of tools. We call the first activity evaluation-driven tool building and the latter one theory-grounded tool building, discussing them in Sections 4.1 and 4.2, respectively.

4.1 Evaluation-Driven Tool Building

“In general, the testing of the effectiveness of many tools has been seriously lacking. The value of many research ideas have not been adequately substantiated through empirical studies.”

Storey et al. [24]

We define the term evaluation-driven tool building to emphasize that tool evaluation should be an integral part of a tool building effort. Evaluation should be considered not only as an afterthought but also during the whole tool building project.

Sensalire et al. describe a tool evaluation cycle that consists of four steps [296] (Fig. 14). This cycle emphasizes that tool building is an iterative activity that
receives feedback by conducting evaluations. In the reverse engineering community, there is increasing realization that tool evaluations are needed to advance the field (e.g., [156, 203, 296]). Concerns that proper tool evaluations are needed have been voiced in the community at least for a decade (e.g., [24]). However, tool evaluation is not yet a standard practice. In order to establish tool evaluations more firmly, there needs to be a consensus in the community what constitute a “proper” evaluation. Typical examples of tool evaluations that can be found in the literature are

—Personal impression of the tool developers.
—Anecdotes of tool usage reported back to the tool developers.
—Checking the tool’s features against a list of requirements (Section 2).
—Observational user studies.

Evaluations of tools involving users are mostly conducted with a smaller number of subjects. Furthermore, these subjects are often from a purely academic background. Less than 10% of the empirical studies in software engineering use professional software developers [299]. For reverse engineering, this number is probably even lower, leading to the question—perhaps voiced in exasperation—of “by the way, did anyone study any real programmers?” [27].

Many user studies are controlled experiments (i.e., they are conducted in an artificial environment, \textit{in vitro}). In addition, more studies that evaluate a tool’s adoption and use in industry (i.e., natural work environment, \textit{in vivo}) would be highly desirable. Such studies are less controlled by their nature, pose logistical challenges, and “may result in vast amounts of data that is difficult to analyze”
Controlled experiments with their quantitative nature should be complemented with qualitative studies that ideally observe industrial programmers [300].

Another concern is that most user studies have a small sample size. Huang and Tilley observe that most software engineering studies have this problem: “In most software engineering studies, researchers are often pleased if they can attract a dozen students to participate in their experiment” [202]. Sjöberg et al. report in a survey that on average studies have 49 subjects (with a wide range from 4 to 266) [299]. Studies with a small sample size suffer from the problem that their statistical significance is questionable. There are large studies in software engineering such as an evaluation of pair programming that involved 295 professional software developers [301]. Unfortunately, to our knowledge there are no studies for reverse engineering that are even remotely comparable in terms of size and effort.

Empirical studies can be distinguished by their distance from human contact [27]: first-degree contact exhibits a direct involvement of study participants, second-degree contact exhibits indirect involvement of participants by collecting data about them, and third-degree contact relies on theories that have been informed by lower degrees of contact. User studies are first-degree with respect to the human contact. While first-degree of contact has many benefits, it also poses potential problems. Storey cautions that “observations can also be disruptive and could be subject to the Hawthorne effect (e.g., a programmer may change her behavior because she is observed)” [203]. If students are involved there is the added problem that “they will often feel pressured to participate in their professor’s research even if they are assured that participation is voluntary” [302]. Also, Berry and Tichy state that “students in particular have a strong desire to please their instructors” [303]. These problems are mitigated by evaluation approaches that are based on second-degree (e.g., instrumentation of tools to collect usage information [304]) and third-degree (e.g., application of feature analysis to evaluate the tool).

Importantly, evaluation is not necessarily a “big-bang” activity that happens after the tool has been “finished.” Instead, evaluation issues should be considered as early as possible in the development effort. For example, user interface evaluations can be already conducted before functionality is implemented with the help of paper-based prototypes. When designing the tool it can be useful to consider already how the tool can be instrumented to record user interactions during an experiment. Evaluations can be designed based on the tool’s expected research contributions. Such evaluations should be planned during tool development since experiments involving users can take a long time to plan and to get approved by ethics committees. Tool prototypes can be used to evaluate certain aspects of the tool in a lightweight, informal manner. For example, Google uses agile techniques such as “guerilla usability testing (e.g., limited numbers of users hijacked from the Google cafeteria at short notice)” [305]. A promising approach that enables the evaluation
of tools while they are being developed is a process based on (evolutionary) prototyping (e.g., SEI’s Evolutionary Development method [64]).

### 4.2 Theory-Grounded Tool Building

“The best theory is inspired by practice and the best practice is inspired by theory.”

Knuth [306]

The purpose of a theory is to explain a set of empirical observations. In tool building, a theory can be used to guide an empirical study and to explain its outcome. Also, a theory can be useful to drive tool requirements. Storey argues that theories are needed as a solid foundation for research [203]:

“Irrespective of the evaluation technique used, theoretical underpinnings will benefit the evaluations as the results will be easier to interpret. Although our long term goal may be to build better tools, we need to understand why they are better than other approaches.”

In the following, we use the term theory rather loosely—incorporating (working) hypotheses, scientific laws, recurring experiences, lessons learned, etc.—including it to mean a proposed explanation for a phenomenon that is able to improve upon tool building.

An experiment without an underlying theory is mostly only able to establish that the tool has a characteristic based on some measure (e.g., superiority of a tool in terms of maintenance speed). Thus, the experiment is able to support a claim, but lacks a reasoning why this is so. Theories provide a framework for experiments by guiding the setup, focusing the observations, and providing reasoning for the outcomes [156]. There is a large number of potential theories from various scientific fields that can be applied to reason about tools. For example, program comprehension can be investigated at least with cognitive, psychological, and socio-cultural theories [300]. Exton proposes to utilize constructivist learning theories for tool building [307]. He notes, for instance, that “constructivist learning environments provide multiple representations of reality.” An implication for program comprehension tools is that they should have “the ability to present the same information with the same level of abstraction but with a different emphasis.”

While cognitive theories are suitable to study individuals, theories from social sciences are needed if tool is used in a collaborative work style [16]. In this context, O’Brien et al. recommend to incorporate socio-cultural theory, believing that “socio-cultural psychologists would consider that research methods focusing solely on the individual in a purely experimental environment are deficient and incapable of gaining a true insight into how understanding occurs. In fact, most empirical work
carried out to date does not take into consideration external validity in terms of the programmers’ environment, the source code used in these studies, or the tasks required of participants” [300]. Theories about adoption and technology transfer then can be applied to reason about tool adoption in an organizational setting (Section 2.5). This has been advocated by the Adoption-Centric Software Engineering project, which treats tool adoption as a first-class goal.

Instead of applying theories with little or no modification, general theories can be leveraged as a foundation to formulate dedicated theories for tool building and the reverse engineering domain. There are a number of different program understanding strategies that derive from cognitive theories (i.e., bottom-up, top-down, knowledge-based, opportunistic vs. systematic, and the integrated model) [203]. For example, Solloway and Ehrlich’s theory of top-down understanding “borrows” from two sources: text comprehension and problem solving theories [308]. Program comprehension theories can be used to come up with tool requirements. In fact, Storey observes that “many of the researchers that developed the traditional cognitive theories for program comprehension discuss the implications of the developed theories on tool design” [203]. However, she also concludes that “in many cases, the connection to tools and how they could be improved or evaluated according to the theories could be stronger.”

Formation of theories is often grounded in (empirical) observations rather than controlled experiments; the latter are more suitable for testing theories. Theories for program comprehension are often based on observation of programmers and their work practices. Examples of observational techniques are work diaries, think-aloud, shadowing, and participant observation via joining the team [309]. Lethbridge and Singer have developed a method called Work Analysis with Synchronized Shadowing (WASS) to study and record the work of software engineers [27]. Synchronized shadowing uses two note-takers during the observation that place different emphasis on the kinds of data that they are collecting. The records are later merged and encoded in use case maps, which show the flow of tasks including tools such as grep and emacs, contexts such as documents and other people, and actions. Based on the use case maps, work patterns can be synthesized whose interpretation can lead to theories and tool requirements. Lethbridge and Singer used WASS to study eight industrial software engineers and were able to derive a number of requirements for software exploration tools. For example, they found that a “tool should have a simple command to automatically locate occurrences of whatever is in the copy buffer” [27]. This is based on their observation that users frequently used the copy buffer as the argument when performing a search with the editor.

Mayrhauser and Vans use their integrated model of program comprehension to study industrial programmers with think-aloud doing maintenance tasks. Based on their observations, they map maintenance tasks to information needs and tool
capabilities [310]. For example, they found that programmers want to come back to a code segment that they looked at previously. Thus, they have the information need of “browsed locations” which a reverse engineering tool can satisfy by offering a “history of browsed locations.” More recently, Ko et al. have conducted an experiment with students performing maintenance tasks in Eclipse [311]. They say that “the central goal of our study was to elicit design requirements for tools to help with maintenance tasks” and consequently propose six such requirements and how a tool could realize them. For example, their study suggests that part of a task entails “to collect a working set of task-relevant code fragments.” In Eclipse, part of the working set is represented by the open file tabs and the state of the package explorer. As a consequence, switching of tasks results in a loss of the working set that needs to be manually recovered. They propose that a tool should “provide a working set interface that supports the quick addition and removal of task-relevant code fragments.” In a sense, these findings refine Mayrhofer and Vans’ observation that a history of browsed code locations is important during maintenance.

For tool building, theories constitute existing knowledge that can be used to build better tools. In this context, it is important to “package” theories in such a form that they can be more easily applied by researchers that are primarily interested in tool building as opposed to advancing and developing theories. Walenstein has done extensive research in how to apply cognitive theories to tool building, distilling his findings in a cognitive modeling framework (HASTI) that can be used to reason about cognitive support [312]. He explicitly states that “HASTI is tailored specifically to the needs of application-oriented researchers. They need abstractions and simplifications such that the important issues of cognitive support can be efficiently raised and addressed. They also need prebuilt models that can be rapidly and widely applied to yield insight” [156]. Examples of other cognitive theories that can be applied by tool builders are Green and Petre’s cognitive dimensions framework [178], Storey’s cognitive design elements for software exploration [163, 164], and Murray’s cognitive patterns (or microtheories) for program comprehension [313, 314]. Storey has applied her design elements to guide and assess the SHriMP visualization tool; similarly, Farah has applied Murray’s patterns for her Temporal Model Explorer tool [315]. The patterns enabled the generation of a number of (candidate) tool features.

A promising approach for generating theories for program comprehension is grounded theory, which is commonly applied in areas such as cognitive science and psychology. With grounded theory, new theories are synthesized bottom-up from experimental data rather than being informed—and constrained—by existing theories. Additional data leads to a refinement of the theory so that it continues to support all available data. Researchers in the reverse engineering domain have both advocated (e.g., [16]) and applied (e.g., [27, 316]) grounded theory. However, it is
crucial to not abuse grounded theory as a fig leaf for unsystematic theory generation. Based on their experiences, Adolph et al. caution that “like many other researchers who have claimed to follow grounded theory methods and even produce a grounded theory, many of us have only borrow[ed] a few grounded theory practices and have not followed grounded theory as a comprehensive method” [317].

4.3 Discussion

“Scientific rivalry between experimenters and between tool builders can thereby lead to an exponential growth in the capabilities of the tools and their fitness to purpose.”

Hoare [318]

The key elements of empirical research in the reverse engineering domain—theories, experiments, and tools—are mutually dependent on each other. This is illustrated in Fig. 15. Theories can guide tool experiments (e.g., Walenstein’s HASTI [312]) and to a lesser degree provide requirements for tool building (e.g., Mayrhoauser and Vans [310]). Experiments can be used for tool evaluation and to confirm or refute theories. Same as theories, experiments can provide requirements for tool building (e.g., Lethbridge and Singer [27]).

An approach to tool building that is evaluation-driven and theory-grounded has to follow a suitable process that incorporates theories, experiments, and tool building. An example of such a process that incorporates these elements is described by Storey [163]. The process consists of “several iterative phases of design, development, and evaluation” and has been used on the SHriMP tool. Figure 16 depicts a rendering of the process iterations with SHriMP as the subject tool. There is an “iterative cycle of design and test” that aims at improving a tool [164]. The (initial) design of the tool is guided by Storey’s cognitive design elements framework, which is a theory that provides a catalog of issues that should be addressed by software exploration tools. Evaluation of the tool’s design and implementation can be accomplished by empirical studies based on observation of tool users. This cycle is

![Fig. 15. Cycles of tool construction, experimentation, and theory creation.](image-url)
embedded in a larger iterative cycle for improving the underlying theory. When adopting Storey’s process, the cognitive design elements framework could be replaced with a different theory.

Compared to the other two lenses, tool evaluation is still in its infancy. However, the evaluation of tools has made rapid advances within the last few years, which is a source for optimism that some form of empirical evaluation will become an expected component when reporting a research result. However, empirical studies are costly and thus should be conducted strategically (e.g., based on cost-benefit concerns). For example, the goal is not to maximize the quality, but to find the right level of quality [283]. Having a large number of lower quality studies that are only conducted as a token effort to get a tool or method published may turn out to be detrimental to the long-term impact that empirical studies can have in synthesizing broadly accepted theories. In fact, if empirical studies are not following a rigorous procedure, the resulting data “may fail to support even a true hypothesis, or, conversely, false hypotheses may be believed to be true” [319]. It is now expected that an empirical study discusses threads to validity. However, instead of repeating (boilerplate) discussions for each evaluation, the effort should go toward reducing the threads. Also, not everything needs to be empirically investigated (e.g., it may be sufficient to rely on qualified opinions or long-term anecdotal evidence). For example, Parnas points out that the effectiveness of Kirchhoff’s circuit law has not been empirically evaluated—its effectiveness is simply accepted by electrical engineers [320].

Key concerns for empirical research in the reverse engineering domain are the development of agreed-up standards that make it easier to conduct, compare, and reason about empirical studies. In his WCRE 2006 keynote, Briand makes the point...
that “it is important that the reverse engineering research community strives to provide guidelines and develop specific empirical procedures and benchmarks that would eventually converge to become standard practices” [321]. An example of work in this direction is the proposal of a framework for defining and performing experiments for program comprehension, and of collaborations among researchers where each participant contributes to a study in a different way [298, 322]. Researchers have also proposed a common coding scheme for observational studies [323]. Reverse engineering can benefit from software engineering, which is facing similar challenges [324]. Zelkowitz and other researchers advocate the sharing of information related to empirical studies in software engineering among research groups [325]. Examples of information that should be shared are experimental results in the form of quantitative data; artifacts that support an experiment such as code, design documents, and test plans; tools for data collection; and procedures and guidelines such as reporting forms.

A straightforward approach to advance empirical studies is to work toward a benchmark, or at least a common set of subject systems. A benchmark is relatively easily applied and promises replication. Without a standard set of subject systems and evaluation principles encoded in a benchmark, there is the risk that tool evaluations are biased by applying a new tool or technique to favorable software systems. In addition, defining a benchmark can have a community-building effect, advancing the discipline: “Throughout the benchmarking process, there is greater communication and collaboration among different researchers leading to a stronger consensus on the community’s research goals” [287]. Communication of lessons learned are another example that can help to advance empirical research. Sensalire et al. describe lessons learned based on five different user studies with software visualization tools [296]. Adolph et al. discuss lessons learned in applying grounded theory in the software engineering domain [317].

A key problem of empirical research is lack of knowledge and interest. Reverse engineering researchers that are willing to advance the state of empirical research (e.g., by applying grounded theory) are rather ill equipped because their background has not prepared them to apply social science research methodology [317]. Huang and Tilley provide an example that affirms this point [202]:

“We are novice empiricists who are interested in using empirical studies as a measurement instrument; we are not particularly interested in becoming experts concerning empirical methods per se. As novices, we struggled with learning how to properly execute an empirical study, while at the same time remaining focused on the underlying problem that we were interested in exploring.”

Also, researchers that have chosen an engineering discipline are often not particularly keen on becoming experts in empirical research. One avenue to raise the
quality of empirical studies and the construction and application of theories is that the reverse engineering community works toward establishing a research environment that encourages researchers to specialize in empirical research involving tools. Such a research environment would need to give adequate scientific credit for empirical work (e.g., synthesizing of tool requirements and replication of tool evaluations) on a par with nonempirical work. This would lead to a mutual fertilization of researchers that focus on tool building and tool evaluations. Walenstein describes a grand vision of theory-based research in which multidisciplinary collaboration would lead to theories that serve as a foundation for software engineering research. In this vision, “software engineering researchers will be portrayed as consumers of theoretical advances from other disciplines, the focus of the software engineering thread of research will be set on applications of theories derived elsewhere” [156, p. 56].

5. Conclusions

In this chapter, we have addressed the issue of academic tool building in the reverse engineering domain. We have explored this issue with the help of three lenses: tool requirements, tool construction, and tool evaluation. Each lens represents an opportunity to reflect upon the current research practice and how to improve upon it.

Each lens has the potential to positively influence other lenses, which in the best case may lead to a positive feedback cycle. Articulating tool requirements represents an opportunity to make research goals more explicit. For example, what are the desirable quality attributes that the tool has to meet? It is expected that this set of requirements will be changed and refined as research advances and the tool gets constructed. Thus, tool requirements and tool construction are mutually influencing each other. Initial tool requirements can serve as early guidance for tool construction. For example, quality attributes can inform tool architecture. Conversely, experiences gained by building and test-driving a tool prototype can refine the tool’s requirements.

The tool’s quality attributes also depends on the evaluation. For example, what are the minimum requirements so that the tool can be used for the envisioned tool evaluation? If the tool is evaluated by the tool developers themselves (e.g., benchmarking its scalability) then usability and adoptability are less of a concern. Conversely, if the tool is evaluated by a user study then a certain level of usability has to be met. Furthermore, the tool has to scale up to accommodate at least the evaluation environment, which may be a subject system with millions of lines of
code in an industrial setting. A key motivation of empirical studies is to show that a new tool or technique provides a measurable benefit compared to the state-of-the-practice. Finkelstein and Kramer point out that researchers “cannot expect industry to make very large big-bang changes to processes, methods and tools, at any rate without substantial evidence of the value derived from those changes” [106]. Thus, convincing empirical studies can support tool adoption by industry. While there is a considerable attention on process in software engineering, the process of developing tools in academia is still neglected. Oivo proposes to make the development work in a research project the subject of empirical study, treating it as an experiment or quasi-experiment [326]. This approach provides a more realistic environment for experimentation compared to small-scale user studies involving students. It has the potential to strengthen empirical results and to introduce more rigor into academic tool building.

Besides the discussed lenses, reverse engineering research is experiencing a number of new challenges due to novel perspectives on how to reverse engineer systems and due to the shifting nature of the systems themselves. These challenges reflect back on the lenses in terms of new emerging tool requirements, new tool architectures, and new avenues for empirical evaluations.

Research in software evolution has led to the realization that there is no strict separation between the development and maintenance phase of a software system [10]. Similarly, reverse engineering activities are not removed from forward engineering activities—both are rather intertwined. This understanding has consequences on reverse engineering tools. First, it leads to a tighter integration of the reverse engineering tools with the software development tools, meaning in practice that reverse engineering functionality is integrated seamlessly within the developers’ IDE. Second, reverse engineering functionality has to be accessible not only instantaneously but also produce results instantaneously. Thus, batch-processing of tasks that run for a longer time period are out-of-step with a highly interactive, IDE-driven working style. In a sense, each additional second that an analysis takes rapidly decreases its attractiveness for the tool user. Thus, the idea of JITC is complemented by real-time comprehension and real-time reverse engineering. Third, following the lead of IDEs to become more collaborative, reverse engineering tools are following suit. Incorporating collaborate features enables novel reverse engineering functionality, not just augmenting existing reverse engineering approaches with a collaborative touch. However, this also leads to new challenges for tool evaluation because studies now have to cope with multiple tool instances that operate in a collaborative environment. Fourth, tighter integration of reverse and forward engineering in a single environment means that reverse engineering tools have access to and can leverage data sources that were previously not available. If data about every single keystroke of the user becomes available for data
mining, powerful recommender systems that operate in real-time become feasible. Thus, we expect that reverse engineering will increasingly incorporate and leverage functionality such as can be found in recommender systems. Furthermore, these new data sources can be used in empirical studies of tool usage.

In a user study involving software visualization tools for performing corrective maintenance, the participants questioned the need for dedicated tool support, voicing their preference for the Eclipse IDE [75]. This may be explained with the fact that Eclipse is offering functionality that was previously only provided by reverse engineering tools. Either way, it shows that reverse engineering tools are now competing with vanilla IDEs. If research in reverse engineering wants to stay relevant it has to show that its tools are useful beyond the state-of-the-art of industrial software development environments. A study of doing maintenance with Eclipse showed that only 20% of the time is spent editing code besides other activities such as reading code (20%), searching for names (13%), and navigating dependencies (16%) [311]. The study also exposed the potential of new tool support that could save up to 35% of the time spent on code navigation during maintenance tasks. Thus, dedicated tools for program comprehension and reverse engineering have the potential to make a huge impact if they support the right functionality (i.e., usefulness) together with the right interactions (i.e., usability). However, the increasing complexity of the underlying reverse engineering infrastructure and the pressure to compete with polished industrial tools may necessitate a community effort to collaborate on a common infrastructure (e.g., in the form of an application framework) for doing joint reverse engineering research [327, 328]. Such an infrastructure should allow “the state-of-the-art to be readily applied” and thus to advance upon it more easily [327].

The constantly changing nature of software means that reverse engineering approaches have to adapt accordingly. Traditionally, reverse engineering is concerned with large monolithic systems that are coded in a statically typed high-level language. Over the years, established programming languages have been getting more dynamic in nature (e.g., C++’s RTTI and C#’s Reflection.Emit) and new languages already have more dynamic features (e.g., Ruby). Domains such as web applications often rely on scripting languages on the client side (e.g., JavaScript) and server side (e.g., PHP). This trend has an impact on reverse engineering tools because static analyses are becoming less precise and whole program analyses are becoming impractical [297]. Thus, dynamic analyses are playing a more prominent role in dynamic languages to complement statically derived information.

Besides programming languages, the targeted systems are changing their characteristics. Systems are becoming less monolithic, more dynamic, and more adaptive in nature. This trend is exhibited by approaches such as SOA (where services are loosely coupled and can be discovered during run-time) and self-adaptive systems (where the system reconfigures itself during run-time) [329]. These kinds of
software require analyses that monitor the system at all levels of granularity and corresponding visualizations [330]. Such systems also often have components that are black boxes in the sense that their inner workings and source code are not accessible for reverse engineering. Thus, reverse engineering techniques may have to rely primarily on analyzing the interfaces and observing the interactions of black-box components. The same characteristic holds for emerging approaches such as Software as a Service (SaaS) and cloud computing, which hide most parts of the software behind the service provider’s server. While such restrictions are not entirely new and can also be encountered in COTS-based and distributed systems, the increasing impact of SOA, SaaS, clouds, and self-adaptivity may necessitate radically different reverse engineering techniques.

There is also the realization that systems do not exist in isolation and have to be studied and understood in a broader context. The concept of Systems of Systems (SoS) acknowledges that the boundary of a software system is often blurred and shifting because it communicates with other systems. Also, a SoS includes systems over which the integrator has little or no control [331]. As a consequence, the subject system that reverse engineering targets may not be readily identifiable and easily bounded if that system is in fact a SoS. This problem becomes more pronounced if the reverse engineering target broadens toward ultralarge-scale (ULS) systems [332].

Looking at the challenges ahead, research in reverse engineering promises to remain exciting, despite—or perhaps because of—its growing maturation in all three lenses. While tool building is a means to an end, it does no harm that it has a very attractive characteristic—it is fun!

REFERENCES


J. Singer, Creating software engineering tools that are useable, useful, and actually used, 2004, talk given at the University of Victoria, December.


