The Rigi Reverse Engineering Environment

Holger M. Kienle   Hausi A. Müller
University of Victoria, Canada

Abstract

The Rigi environment is a mature research tool that provides functionality to reverse engineer software systems. With Rigi large systems can be analyzed, summarized, and documented. This is supported with the extraction of information from source code, an exchange format to store extracted information, analyses to transform and abstract information, and visualization of information in the form of typed, directed graphs.

In this paper we describe Rigi's main components and functionalities, and assess its impact on reverse engineering research and practice. Furthermore, we discuss Rigi's architecture and motivate design decision that resulted in a decoupling of major functionalities and achieved tool extensibility, interoperability, and end-user programmability.

Key words: Reverse engineering, Program comprehension, Tool building, Tool requirements

1. Introduction

> "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*

Reverse engineering of a software system is performed for a broad variety of reasons ranging from gaining a better understanding of parts of a program, over fixing a bug, to collecting data as input for making informed management decisions. Depending on the reasons, reverse engineering can involve a broad spectrum of different tasks. Examples of such tasks are slicing, clustering, database or user interface migration, objectification, architecture recovery, metrics gathering, and business rule extraction.

Email addresses: `kienle@cs.uvic.ca` (Holger M. Kienle), `hausi@cs.uvic.ca` (Hausi A. Müller).
URLs: `holgerkienle.wikispaces.com/` (Holger M. Kienle), `webhome.cs.uvic.ca/~hausi/` (Hausi A. Müller).

Software systems that are targets for reverse engineering, such as legacy applications, are often large, with hundreds of thousands or even millions of lines of code. As a result, it is almost always highly desirable to automate reverse engineering activities. The Rigi environment [1] provides such tool support. The reverse engineering community has developed many reverse engineering environments. Prominent examples of other tools besides Rigi include SHriMP [68], Moose [8], Ciao/CIA [2], and Columbus [11]. Reasoning System's Software Refinery is an example of a commercial tool that has influenced and enabled reverse engineering research [1].

The rest of the paper is organized as follows. Section 2 summarizes Rigi's impact on reverse engineering research and tools in academia and industry. Section 3 describes the functionality of the environment, covering the repository and data model, the fact extractors, and the graph-based, interactive editor. Section 4 addresses Rigi's availability and user support. Section 5 distills tool-building experiences and lessons that we learned in the design and implementation of Rigi. Section 6 closes the paper with our conclusions.

2. Rigi's Impact in Academia and Industry

Rigi is a mature research tool that is in its current form now more than a decade old. In the following, we briefly identify Rigi's contributions on academia and industry. Some of these contributions are explained in more detail in the rest of the paper.

tool features: Rigi has pioneered the approach to reify software entities and their relationships as graphical entities and to manipulate them interactively to enhance program comprehension for software development and maintenance [39]. At the time Rigi was conceived, there were other approaches to leverage graphical visualizations for software, but they were primarily geared to aid in the design of software systems. For example, Software through Pictures used graphical editors to represent data-flow diagrams, entity-relationship models, and data structure definitions [65].

case studies: A significant number of reverse engineering case studies have been conducted with Rigi. These case studies cover a broad range of systems and languages and testify of Rigi's effectiveness to support reverse engineering tasks. Furthermore, the case studies were useful to identify improvements for Rigi in functionality and user interaction design.

In the following we briefly describe some of the documented case studies. For the `xfig` drawing tool (which is written in C) an architecture analysis was conducted with Rigi for a working session at WCRE 2000 [34]. In a similar study, a ray tracer written in C (12 KLOC) was reverse engineered with Rigi [41] [38]. Rigi has also participated in the Sortie structured tool demonstration along with five other tools [24]. The demonstration's purpose was the reverse engineering of the Sortie legacy system, which consists of 30 KLOC of Borland C++ code [54]. For the VISSOFT 2007 Tool Demo Challenge, Rigi was used to recover the architecture of the Azureus BitTorrent client (Java, 300 KLOC) [23]. Rigi was also used to visualize link dependencies of two web sites (550 and 3500 pages) [33].

[1] Even though we use the terms environment and tool interchangeably for Rigi, we view an environment as more powerful than a tool. Environments provide many tools with diverse functionalities under one roof, whereas a single tool is designed for a particular (major) function.

Rigi has been used in industrial contexts as well. For example, it has been used on an industrial health care application of 57,000 lines of Cobol code [40], and on IBM's SQL/DS database system [1] consisting of two million lines of PL/AS [2] code [67]. At Nokia, Rigi has been used to recover the architecture of a mobile phone application written in C [51]. In another industrial project, Rigi has been used to analyze and visualize the structural dependencies of 700,000 lines of C/C++ code [35].

software redocumentation methodology: The reverse engineering of (industrial) systems with Rigi has generated valuable experiences. These experiences have also shaped Rigi's methodology of how to reverse engineer—or *redocument*—(legacy) systems.

Rigi supports an approach to reverse engineering that recognizes that this activity has inherently creative elements that cannot be fully automated by a tool. However, a tool should allow to automate lower-level, repetitive tasks that are otherwise tedious to perform and distracting for reverse engineers. Consequently, Rigi is a *human-involved tool* [16] that offers both interactive and automated functionality. Furthermore, there is no fixed segregation of interactive vs. automated functionality in Rigi because task-specific interactions can be automated by reverse engineers on demand using Rigi's scripting functionality. This has been coined *programmable reverse engineering* [63].

exchange format: Rigi defines the Rigi Standard Format (RSF) [66], which has been adopted by a number of other tools as well. RSF has inspired other tuple-based formats such as Holt's Tuple-Attribute language (TA) [13]. Rigi's graph model was also inspirational for the Graph Exchange Language (GXL) [14].

exploratory research and tool prototypes: The Rigi environment can be seen as a tool platform that enables exploratory research in the reverse engineering domain. Especially, Rigi enables the rapid prototyping of novel analyses and visualizations. The following tools are based on Rigi, but provide major enhancements to Rigi's functionality:

- Bauhaus [25]
- Nimeta [52, sec. 8.5] [53]
- Dali [19] [46]

The Bauhaus and Nimeta tools have been developed as dissertation works and realize novel analyses and visualizations for program comprehension. Dali has been partially developed within the Software Engineering Institute and applied to industrial systems.

As can be seen, Rigi had an important impact on research tools and, to a lesser degree, also on industry.

3. The Rigi Environment

Rigi is composed into three main entities that provide functionalities for (1) information representation, storage, and exchange (repository), (2) fact extraction, (3) and interactive graphical manipulation (editor). We discuss each of the main entities in the following sections.

The typical workflow for the reverse engineering of software systems can be characterized by three main activities: extract, analyze, and visualize [49]. A good reverse engineering tool should provide support for all three activities. In Rigi, fact extraction

[2] PL/AS is a PL/1 dialect used within IBM.

is supported with a number of parsers and scanners (who produce output for the repository); analyses (both interactive and automatic) as well as visualizations are supported with the graph editor.

3.1. *Conceptual Architecture*

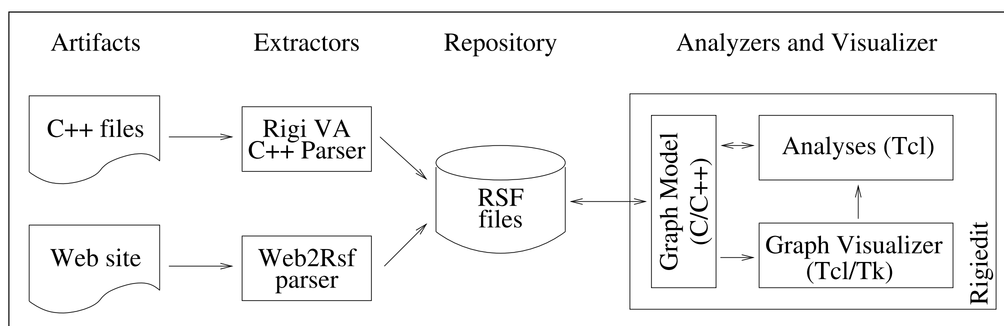

Fig. 1. Rigi's conceptual architecture

Rigi's conceptual architecture is depicted in Figure 1. The architecture exposes Rigi's main functionalities: extraction of facts from software systems (cf. Section 3.3), a repository to represent and store facts (cf. Section 3.2), and analyses and visualization of facts (cf. Section 3.4). The architecture shows two extractors for C++ (Rigi VisualAge parser) and web sites (Web2Rsf parser) that exemplify that there can be an open-ended number of different extractors for various domains.

As can be seen, Rigi is an example of a tool that decouples its extractors from the rest of the system via its exchange format, RSF. However, Rigi's analyses and visualization are intertwined. Analyses (written in Tcl) are stored in separate text files, and use an API to access and manipulate Rigi's internal, in-memory graph model (implemented in C/C++). Analyses are invoked interactively from Rigi's visualizer (e.g., via selecting an entry from a pull-down menu). Once an analyses is invoked, it can update the graph model; for instance, a clustering analysis can create new high-level nodes and group existing nodes into hierarchical structures. A changed state of the graph model is then immediately reflected by the graph visualizer. Consequently, analyses cannot be executed without running the visualizer itself.

3.2. *Repository and Data Model*

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.

Rigi follows a lightweight approach by storing extracted facts in a single text file using a dedicated exchange format.[3] This approach is taken by many reverse engineering tools [21] [17]. Rigi defines the Rigi Standard Format (RSF) [66], which uses sequences of tuples to encode graphs. A tuple either represents a node and its type, an arc between two nodes, or binds a value to an attribute. The following example shows the RSF to represent two C functions (`f1` and `f2`) and a function call to `f2` in the body of `f1`:

```
type f1 Function
type f2 Function
Calls f1 f2
```

Even though RSF has been created to represent facts for software reverse engineering, it can encode any kind of information represented as typed, directed, attributed graphs. Examples of other exchange formats in the reverse engineering domain are Holt's Tuple-Attribute language (TA) [13], Bauhaus' Resource Graph (RG) [5], GUPRO's GraX [9], and the Graph Exchange Language (GXL) [14].

The facts that are stored in RSF adhere to a certain data model (or schema).[4] A RSF data model is explicitly defined with a simple specification language that is also tuple-based. There are three separate files that encode the nodes (`Riginode`), arcs (`Rigiarc`), and attributes (`Rigiattr`) of a data model. There is an optional `Rigicolor` file to specify the colors of nodes and arcs for the Rigi editor. Rigi's approach is to keep the data and the data model separate and there is no explicit association between them. In practice, the user of the editor first specifies a data model (also called domains in the Rigi editor) and then loads an RSF file. When the RSF is loaded it is validated against the chosen domain.

The node specification contains a list of node types (one per line); the arc specification contains a triple for the arc type and the source and destination node types (also one per line). For example, a valid data model for the above example would be

```
# node types
Function
# arc types
Calls Function Function
```

Rigi defines (standard) data models for C, C++, and Cobol. These data models are capable of representing *middle-level* information [30]. A middle-level model focuses on the main program elements and their relationships, omitting complete syntax of code while not abstracting information all the way up to generic architectural elements (i.e., components and connectors).

3.3. *Fact Extractors*

A reverse engineering activity starts with extracting facts from a software system's sources. Sources can be intrinsic artifacts that are necessary to compile and build the system (such as source code, build scripts, and configuration files), or auxiliary artifacts (such as logs from configuration management systems and test scripts).

[3] The RevEngE project augmented Rigi with an object-oriented database system (ObjectStore) as its persistent storage manager [43]. However, the results were not incorporated into the Rigi distribution.
[4] An analogous example is data encode in XML that adheres to a certain XML Schema or Document Type Definition (DTD).

Rigi's fact extractors target source code exclusively. Both the C (`cparse`) and Cobol (`cobparse`) parsers are based on Lex and Yacc. There is also a C++ parser (`vacppparse`) that is built on top of IBM's Visual Age compiler [32]. [5] There are also fact extractors developed by other groups that support Rigi (e.g., Columbus/CAN's C/C++ parser and SHriMP's Java extractor).

For research projects additional fact extractors have been developed as well, but those are not publicly available. For example, to analyze software documentation a scanner for LaTeX was developed [62]. Similarly, to analyze the HTML of web sites, a Java-based web crawler, called Web2Rsf, was developed [33]. In contrast, an ad-hoc approach was taken to extract facts from PL/AS code based on lexical pattern-matching with Unix scripts [67]. These scripts extract higher-level structural information (e.g., call graphs) and represent them as C code, which can then be process with the C parser.

Another fruitful approach to obtain facts is to leverage a "foreign" extractor and to write a converter that transforms the information that the extractor produces into RSF. For example, to analyze Azureus we used facts provided in FAMIX/MSE and wrote a Perl script to translate them into RSF [23].

3.4. *Graph-Based Editor*

The core of Rigi is a graph editor/drawing tool, `rigiedit`, enhanced with additional functionality for reverse engineering tasks. Rigi's core functionality is similar to the functionality offered by generic graph editors. Graphs can be loaded, saved, and laid out; the windows rendering a graph can be scrolled and zoomed; the graph's nodes and arcs can be selected, cut, copied, and pasted; and so on. The types of nodes and arcs along with color information are described in Rigi's domain model (cf. Section 3.2). Examples of reverse engineering functionality in the editor are the computation of cyclomatic complexity and the navigation to the underlying source code that is represented by a node. Rigi combines graphical visualization with textual reports (e.g., a list of a node's incoming and outgoing arcs) to provide information about the graphs at different levels of detail.

Figure 2 shows a screenshot of the editor with a graph and two dialogs to filter the graph data by node and arc types. The visualized graph shows the structure of IBM's SQL/DS system, which is over one million lines of PL/AS code [67]. Red nodes are variables, yellow nodes are modules, and purple nodes are types. All arcs have been filtered out except for the yellow ones, which represent calls between modules (i.e., the graph shows the system's call graph). The full graph contains 923 nodes and 2395 arcs, but in this filtered view only 189 yellow arcs are actually visible.

Many reverse engineering tools now use graphs to convey information [26]. Examples of visualizers similar to Rigi that have been developed by researchers for reverse engineering and program comprehension are CodeCrawler [28], G^{SEE} [10], LSEdit [55], VANISH [18], Hy+ [4], G+ [3], ARMIN [47] [20], and SoftVision [59] [58]. Besides these editors, there are also general-purpose graph layouters and editors. The EDGE graph editor is an early example [44] [50]. AT&T's Graphviz provides an interactive editor, `dotty`, which can be customized with a dedicated scripting language, `lefty` [12] [45].

[5] Unfortunately, IBM has dropped support for VisualAge C++ in many of the previously supported platforms. As a result, the parser is currently only operational on AIX.

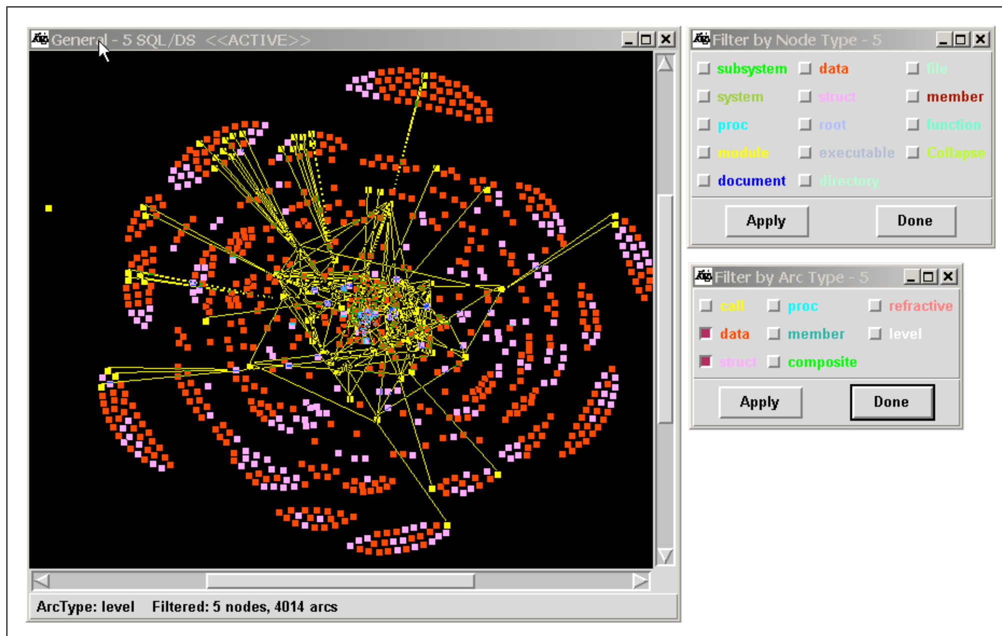

Fig. 2. Software structure graph in Rigi

3.4.1. *End-User Programmability*

The editor provides support for scripting with the Rigi Command Language (RCL). RCL is a transparent scripting layer between the GUI and the underlying graph model implementation [61] [60]. RCL is a collection of Tcl commands (that are indistinguishable from built-in commands). RCL provides low-level commands to manipulate graphs (e.g., creation of nodes and arcs as well as loading, saving, filtering, selecting, moving, and layouting operations) and higher-level functionality to analyze graph properties. Generally, all graph functionalities that can be invoked from the GUI are also accessible via RCL commands. For example, in Figure 2 the user can un-filter the red data arcs by first ticking the appropriate box in the “Filter by Arc Type” window and then clicking the “Apply” or “Done” button. The same effect can be obtained with the following RCL script:

```
# set arc filter for window 5 (cf. Figure 2)
rcl_filter_arctype_filtered data 5
# apply the filter
rcl_filter_apply 5 arc
```

Rigi has an RCL console that allows the user to type in RCL commands, to select them from a command list, or to execute a script file.

The Tk-based GUI is also implemented with Tcl and RCL. This approach makes it possible to personalize Rigi’s user interface (e.g., adding a new menu item). Personalization can be done at start-up or during execution of the editor. For example, Rigi provides a hook to run a Tcl script whenever a certain domain is selected by the user. This way, domain-specific analyses and GUI behavior can be easily realized.

Since RCL inherits the features of Tcl—prototyping based on scripting [48]—it allows the user to rapidly develop new functionality and to customize the GUI. In other words,

Rigi offers *end-user programmability*. Importantly, RCL introduces flexibility without increasing complexity because users can ignore RCL and its features when using the editor interactively.

4. Availability and User Support

Rigi has a dedicated web site that is available at www.rigi.cs.uvic.ca. The site offers pre-compiled downloads of Rigi for various operating systems, including Windows and Linux. The source code is available for download as well. The pre-compiled downloads contain an executable (`rigiedit`) that can be directly invoked to bring up the GUI-based graph editor (cf. Figure 2). There is also support to process C (`cparse`) and Cobol (`cobparse`) source code as well as utilities to convert and manipulate RSF files (`sortrsf`, `htmlrsf`, and `rsf2gxl`).

In order to flatten Rigi's learning curve and to increase adoptability there are extensive resources and documentation. In fact, Lanza has stated that the "best-documented software visualization tool we know is Rigi" [28]. Rigi comes with a user's manual of 168 pages that explains Rigi's purpose, provides a quick tour of Rigi, and then covers all of Rigi's functionality in detail [66]. The Rigi editor comes with three sample systems of increasing complexity that demonstrate how a software systems can be reverse engineered with Rigi. The user runs these interactive demos directly within the editor.[6] The Rigi site also provides a description of a subset of RCL and an example of reverse engineering Mozilla. There is also a Wiki[7] that allows users of Rigi to add information about Rigi and their experiences with it. The most useful resource currently available on the Wiki are the Frequently Asked Questions (FAQ).

Active development of Rigi stopped in 1999, but occasional maintenance and bug fixed were performed as well as porting to new Windows versions. The last official version of Rigi was released in 2003. To this date, Rigi is still used in academia and industry for teaching and research.

5. Experiences

This section provides a discussion of Rigi's approach to reverse engineering and program comprehension. We also distill experiences and lessons learned that analyze what made Rigi a successful tool and what could be improved.

Perhaps the most important lesson that we learned is that approaches that support reverse engineering have to be lightweight and flexible. The reverse engineering process is characterized by trial-and-error. As a result, there is a constant jumping between the central activities of fact extraction, analysis, and visualization. There is also a rapid generation of hypotheses by the reverse engineer in the form of questions about the system under analysis that need to be answered "instantaneously" in the best case. Furthermore, reverse engineering activities are quite diverse and depend on many factors. Consequently, reverse engineers have to continuously adapt their tools to meet changing needs. Thus,

[6] A walk-through of one of these demos is also available on the Rigi web site at www.rigi.cs.uvic.ca/downloads/demos/list-d/rigi.listdemo.html

[7] www.program-transformation.org/Transform/RigiSystem

it is not sufficient 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). Specifically, flexibility mandates tools that allow users to become productive very quickly, that can be easily customized and extended, and that are able to interoperate with other tools.

5.1. *Graph Visualization*

In a sense, the graph editor is the heart of the Rigi system. In many projects, only the editor is used (ignoring the rest of the system), and only the editor's interactive facilities are used (ignoring RCL). Since the graph editor has been developed from the start to render and manipulate large graphs, it scales up to graphs with thousands of nodes. To obtain the needed performance for larger graphs, the graph model and the core algorithms to manipulate it are implemented in C/C++, whereas higher-level algorithms are scripted in Tcl. This split has been proven successful and was applied by other visualizers as well (e.g., SoftVision [58]). Nowadays, an implementation that is exclusively based on a scripting language such as Perl would be probably feasible.

Rigi has a fully-featured graph editor, but its graphical capabilities have not been evolved in recent years. As a result, many desirable advances in software visualization are now lacking. The most important deficiencies are that all nodes and arcs have the same shape, and arcs have the same line thickness. Because of this, it is not possible to associate metrics with graphical entities in Rigi. To work around this limitation, a metrics extension in Rigi provides the values as node annotations (but the user has to open a textual view to see them) [57]. An approach similar to polymetric views would be very desirable [29]. Furthermore, Rigi does not support drag-and-drop [6]; nodes (and associated arcs) have to be copied via the clipboard. This style of interaction is less intuitive for users that expect behavior of typical Windows applications. Lastly, to simplify creation of scripts it would be useful to have recording and play-back feature (analogous to Microsoft Office).

During the reverse engineering of the Azureus system [23] we noticed several deficiencies that make interactive exploration unnecessarily difficult with Rigi:

- An arc between two nodes does not indicate how many dependencies it represents. To find out this information it is necessary to open up a textual view with an arc list. As a result, it is tedious to separate major from minor (or "spurious") dependencies. Other tools (e.g., Softwarenaut [31]) offer arcs with flexible line width.
- We sometimes destroyed interesting layouts of views because Rigi has no undo.
- There is also no easy way to move a node into a `Collapse` node. This can be only accomplished with a sequence of expand, select, and collapse operations.
- When opening up a view it is not possible to right away apply an automatic layout if the viewed graph is disconnected (precludes Spring) or cyclic (precludes Sugiyama).
- It is not possible to apply the Spring or Sugiyama layouts on a subset of selected nodes. Other visualizer are more flexible in supporting mappings from node selection to visualization operations (e.g., [59]).

There are many approaches to construct a fact extractor for a particular target language. For instance, an extractor can be implemented as a compiler front-end. This implies a parser that produces a parse tree without ambiguities. Such a parser could be implemented from scratch or with a parser generator like Yacc.

In contrast to parsing, there are lightweight approaches such as lexical extractors (or "scanners"), 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) [42]. On the other hand, they are more flexible and lightweight than parsers [37]. For preprocessed languages such as C there is another important difference between parser-based and lexical approaches. Parser-based extractors see the source code after it has been preprocessed, whereas lexical extractors typically operate on the un-preprocessed source.

Rigi's C and Cobol extractors are parsers built with the help of a LR(1) parser generator, Yacc. As a result, the extractor is *brittle* [22] (i.e., it easily breaks if it encounters anomalies such as syntax errors, dialects, and embedded languages [36]). This is a typical problem when reverse engineering a software system with an external tool that comes with its own parser [42]. In fact, we often receive email from users of Rigi that complain that they cannot get its C parser to run through. Typically, the target code needs to be "tweaked" with pre-processor directives that suppress or change C constructs. [8] The former maintainer of the C parser relates his experiences as follows [32]:

> "The Rigi C parser consists of more than 4000 lines of Lex, Yacc, and C++ code. [It's] code is much more complex and very hard to comprehend, in particular the grammar written in Yacc. Although several researchers spent significant amounts of time on the C parser over many years, it still has problems with some input."

Based on our experiences with both parsers and scanners, we advocate a lightweight approach to fact extractor construction that opportunistically collects facts about the legacy system. Typically, lexical extractors are language neutral and reverse engineers write ad-hoc patterns to extract information required for a particular task. This approach is more flexible, results-driven and accommodates the iterative nature of the reverse engineering process nicely. This approach has served us well, for example, in the reverse engineering of the SQL/DS system, where facts were extracted from the system "with a collection of Unix's csh, awk, and sed scripts" [67].

Our experiences are supported by Jackson and Rinard, that say that while such lightweight approaches are *unsound*, they "may provide a useful starting point for further investigation. Unsound analyses are therefore often quite useful for engineers who are faced with the task of understanding and maintaining legacy code" [15]. For example, when migrating a software system to a different platform, it may be in a state such that it cannot be compiled (e.g., because of missing or mismatched header files) [42]. In such a case a parser-based approach will fail. In contrast, a lightweight approach is still able to provide useful—even though incomplete—information such as a partial call graph.

[8] For example, the tweaks to parse the sources of Mozilla are described at `www.rigi.cs.uvic.ca/rigi/mozilla/scripts.html`.

RSF is a lightweight exchange format that is characterized by the following properties. It is a text-based format without nesting structures that is easy to read, manipulate, and repair by humans with any text editor. An interesting feature is its composability: two RSF files can be simply appended to form a new, syntactically valid one. Generally, flat formats such as RSF are easier to compose than nested ones such as XML [7].

To extract information from a repository, there has to be a query mechanism. Querying is an important enabling technology for reverse engineering because queried information facilitates interrogation, browsing, and measurement [27]. Since RSF contains one tuple per line and tuple values are separated by white spaces, standard Unix tools can be used to process it. This is especially important since Rigi does not offer a dedicated query language for RSF. For example, to find out about arc types (and the number of their occurrences) the following commands can be piped:

```
cat file.rsf | cut -d " " -f1 | sort | uniq -c
```

The text-based approach also makes it easy to combing RSF with a version control system and to do file differencing.

RSF is domain-neutral with respect to the stored information. However, the information has to be structured according to its graph model. The constraints that can be placed on the graph with the data model (cf. Section 3.2) are often not as expressive as one might wish for. Furthermore, RSF is a flat format in the sense that there is no scoping of entities. Since nodes are denoted by their name, some form of name mangling needs to be done to avoid name clashes. Rigi's C and C++ parsers concatenate names of namespaces, classes, functions, and variables to produce a node name that is unique for the entire software system. For example, a parameter `argc` of function `main` in file `myfile.c` would be encoded as `argc^main^myfile.c`. Note that the constituents of the mangled name can be easily extracted with Unix tools. For object-oriented languages, the mangled names can become relatively long and somewhat difficult to read compared to procedural languages. Other formats avoid name mangling by supporting name scoping and/or assigning unique IDs to entities (e.g., GXL and FAMIX/MSE). There is a fine line for exchange formats between expressiveness on the one hand and simplicity on the other—and RSF may err on the side of simplicity.

In principle, RSF can represent any kind of information, ranging from fine-grained to coarse-grained facts. However, since all information is kept in a single file and needs to be processed batch-style, it is not practical to store find-grained information of a system (i.e., the full syntax tree). RSF's emphasis on readability combined with the need to mangle names means that it is not storage efficient. For example, for the Azureus case study [23] the system was represented with 32,677 nodes and edges, resulting in an RSF file of 11 MByte.[9] In practice, Rigi has no problem to handle such file sizes because the information can be read in and processed on a line-by-line basis.

An exchange format has to accommodate requirements of both tool users and tool builders, which may be contradictory. 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

[9] Typically, a large part of the file size is caused by long names that are the result of the name mangling. Because these names have many similar substrings, they can be compressed very effectively. The RSF file of the Azureus system is only 0.4 MByte after `gzip` compression.

be parsed easily and efficiently. We believe that RSF is close to a design “sweet-spot” in balancing diverse requirements of different stakeholders. This is supported by the many tools that have chosen to use RSF.

5.4. *Prototyping of New Functionality*

A reverse engineering tool should accommodate the rapid development of new functionality. This is not only of importance for researchers that want to develop proof-of-concept prototypes to demonstrate the feasibility and effectiveness of a novel approach. It is also important for reverse engineers that need a particular analysis or visualization for a specific program comprehension task. In this context, Tilley states that “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” [64]. He concludes that “a successful reverse engineering environment should provide a mechanism through which users can extend the system’s functionality.”

In Rigi, prototyping of new functionality is accomplished with RCL (cf. Section 3.4.1). RCL enables, for example, to express graph transformations programmatically. Typically, such transformation are first manually performed in an exploratory and interactive manner by a reverse engineer and then coded in RCL. For the reverse engineering of the SQL/DS system, “it took two days to semiautomatically create a decomposition using Rigi, but only minutes to automatically produce one via a prepared script. Either method would be much faster and use the analyst’s time and effort more effectively than would a manual process of reading program listings” [67]. Such a programmatic approach also makes it possible to quickly reproduce steps in the workflow if the system changes.

The scripting capabilities have allowed other researchers to quickly customize Rigi to implement their own (prototype) tools. The perhaps most extensive customization is Bauhaus from the University of Stuttgart. The Bauhaus tool provides interactive (architectural) clusterings of software systems written in C [25]. Bauhaus uses Rigi to realize the tool’s user interface, and to provide graph-based visualizations of the clustering results. Importantly, the user can interactively and intuitively select and combine analyses and invoke them on a subset of the visualized graph. The main implementor of Bauhaus relates his experiences with Rigi as follows:

> “Rigi was extended in many directions to adapt it to our needs. The adaptations were opportunistic; not everything what might have been useful could be worked into Rigi, e.g., an undo mechanism would have been helpful. But all of our major requirements were more or less easy to fulfill with Rigi” [25, p. 318].

Scripting also enables interoperability with other tools in terms of data and control integration. Rigi can call out to other tools, and other tools can use Rigi as a service provider (e.g., to perform computations for them). [10] For instance, the Shimba environment for analyzing systems based on both static an dynamic information was realized by combining Rigi (for static graph-based information) and SCED (for sequence and statechart diagrams) [56]. Depending on user actions both tools pass control between them. Dali also uses scripting to enable interoperability of Rigi with other tools [19].

[10] However, it is not possible to run Rigi without its GUI (so-called headless execution).

To summarize, scripting of (GUI) customizations with Tcl and RCL eases experimentation and allows rapid prototyping of functionality. In a research environment, these benefits outweigh potential drawbacks of scripting such as lack of a strong type system and inferior maintainability.

5.5. *Levels of Indirection*

The Rigi system provides a number of mechanism that add flexibility and provide abstractions over certain domain concepts. We refer to such mechanisms as *levels of indirection*. Figure 3 gives an overview of the major levels of indirections in the Rigi systems and how they support tool interoperability and customizability as well as decouplings at the architectural level.

	Interoperability	Customizability	Architecture
RSF exchange format	data integration	graph instances	decoupling of extractors
RSF data model	data integration	graph models	decoupling of extractors
RCL scripting	control integration	graph-based analyses	decoupling of graph model impl.
	presentation integration	GUI	decoupling of widget sets

Fig. 3. Levels of indirections in Rigi

As discussed before, the exchange format provides a decoupling of the graph editor from the extractors. This approach operates at the architectural level and is well known from compiler construction, where intermediate representations are used to decouple the front-end from the middle-end and/or the middle-end from the back-end. The exchange format also provides data integration with other tools. Lastly, the exchange format supports customization of the data that Rigi is dealing with because it allows to encode an open ended number of concrete graphs. This is perhaps an obvious feature, but because of its obviousness this level of indirection is easily missed.

The data model of the exchange format provides another level of indirection. It enables domain-retargetability by customizing the node and arc types (and their attributes). Interoperability is enhanced because data is now typed and entities have semantics. For example, this approach makes it possible to have several fact extractors for the same programming language—but with different trade-offs such as speed or brittleness—that all adhere to the same data model and hence can be transparently exchanged.

RCL scripting enables control and presentation integration with other tools. For example, Shimba leverages this level of indirection to integrate Rigi with another tool, SCED (cf. Section 5.4). Both tools have their own GUIs, but they send messages to each other to synchronize operations, realizing control integration. In contrast to control integration, presentation integration is only practical if the integrated tools use the same toolkit (or a bridge is used that translates from one toolkit to the other). For example, to provide additional analyses and visualizations, Tilley has integrated a spreadsheet (Oleo/tk) and barcharts (The BLT Toolkit) into Rigi [64, p. 61]. Since both tools are based on Tcl/Tk, they could be integrated seamlessly at the GUI-level. RCL also enables customizability of Rigi's functionality in terms of domain-specific analyses that programmatically manipulate the graph model (cf. Section 5.4) as well as personalization to Rigi's GUI (cf.

Section 3.4.1). At the architectural level, RCL also decouples the graph editor from the underlying implementation of the graph model. In principle the current C++ implementation could be replaced with one that uses different algorithms and data structures, or uses a different implementation language.

6. Conclusions

This paper has discussed the Rigi reverse engineering environment and its research contributions. Rigi has the following key features. It offers an exchange format with a graph-based data model; fact extractors for C++, C and Cobol; and an interactive graph editor. Rigi's architecture decouples the fact extractors from the graph editor (via the exchange format). The exchange format allows to define different data models, and the editor's scripting layer provides end-user programmability. Rigi had a significant impact on research in reverse engineering and program comprehension. It has been used to analyze many (industrial) systems, and has enabled the prototyping of novel reverse engineering tools. Rigi's exchange format is supported by many tools and has inspired other exchange formats.

We have also discussed our tool-building experiences with Rigi, identifying benefits and drawbacks of design decisions that we made for fact extractors, exchange formats, and prototyping of new functionality for tool interoperability and customizability. Generally, we advocate lightweight techniques in the construction of reverse engineering tools because this approach reflects the underlying characteristics of the reverse engineering process (which is highly iterative and based on trial-and-error).

References

[1] Erich Buss and John Henshaw. A software reverse engineering experience. *Conference of the Centre for Advanced Studies on Collaborative Research (CASCON'91)*, pages 55–73, October 1991.

[2] Yih-Farn Chen, Michael Y. Nishimoto, and C. V. Ramamoorthy. The C information abstraction system. *IEEE Transactions on Software Engineering*, 16(3):325–334, March 1990.

[3] Mariano Consens, Alberto Mendelzon, and Arthur Ryman. Visualizing and querying software structures. *14th ACM/IEEE International Conference on Software Engineering (ICSE'92)*, pages 138–156, May 1992.

[4] Mariano P. Consens and Alberto O. Mendelzon. Hy+: A hygraph-based query visualization system. *ACM SIGMOD International Conference on Management of Data*, pages 511–516, May 1993.

[5] Jörg Czeranski, Thomas Eisenbarth, Holger M. Kienle, Rainer Koschke, Erhard Plödereder, Daniel Simon, Yan Zhang, Jean-François Girard, and Martin Würthner. Data exchange in Bauhaus. *7th IEEE Working Conference on Reverse Engineering (WCRE'00)*, pages 293–295, November 2000.

[6] Jörg Czeranski, Thomas Eisenbarth, Holger M. Kienle, Rainer Koschke, and Daniel Simon. Analyzing `xfig` using the Bauhaus tool. *7th IEEE Working Conference on Reverse Engineering (WCRE'00)*, pages 197–199, November 2000.

[7] S. Ducasse and S. Tichelaar. Dimensions of reengineering environment infrastructures. *Journal of Software Maintenance and Evolution: Research and Practice*, 15(5):345–373, September/October 2003.

[8] Stephane Ducasse, Michele Lanza, and Sander Tichelaar. Moose: an extensible language-independent environment for reengineering object-oriented systems. *International Symposium on Constructing Software Engineering Tools (COSET'00)*, June 2000.

[9] Jürgen Ebert, Bernt Kullbach, and Andreas Winter. GraX—an interchange format for reengineering tools. *6th IEEE Working Conference on Reverse Engineering (WCRE'99)*, pages 89–98, 1999.

[10] Jean-Marie Favre. G^{SEE}: a generic software exploration environment. *9th IEEE International Workshop on Program Comprehension (IWPC'01)*, pages 233–244, May 2001.

[11] Rudolf Ferenc, Arpad Beszedes, Mikko Tarkiainen, and Tibor Gyimothy. Columbus—reverse engineering tool an schema for C++. *18th IEEE International Conference on Software Maintenance (ICSM'02)*, pages 172–181, October 2002.

[12] Emden R. Gansner and Stephen C. North. An open graph visualization system and its applications to software engineering. *Software—Practice and Experience*, 30(11):1203–1233, September 2000.

[13] Ric Holt. TA: The tuple-attribute language. http://plg2.math.uwaterloo.ca/~holt/papers/ta-intro.html, 1997.

[14] Richard C. Holt, Andreas Winter, and Andy Schürr. GXL: Towards a standard exchange format. *7th IEEE Working Conference on Reverse Engineering (WCRE'00)*, pages 162–171, November 2000.

[15] Daniel Jackson and Martin Rinard. Software analysis: A roadmap. *Conference on The Future of Software Engineering*, pages 135–145, June 2000.

[16] Jens Jahnke. *Managing Uncertainty and Inconsistency in Database Reengineering Processes*. PhD thesis, University of Paderborn, Germany, August 1999.

[17] Dean Jin. Exchange of software representations among reverse engineering tools. External Technical Report ISSN-0836-0227-2001-454, Department of Computing and Information Science, Queen's University, December 2001.

[18] Rick Kazman and Jeromy Carriere. Rapid prototyping of information visualizations using VANISH. *InfoVis'96*, pages 21–28, October 1996.

[19] Rick Kazman and S. Jeromy Carriere. Playing detective: Reconstructing software architecture from available evidence. *Journal of Automated Software Engineering*, 6(2):107–138, April 1999.

[20] Rick Kazman, Liam O'Brien, and Chris Verhoef. Architecture reconstruction guidelines, third edition. Technical Report CMU/SEI-2002-TR-034, Software Engineering Institute, Carnegie Mellon University, November 2003. http://www.sei.cmu.edu/pub/documents/03.reports/03tr034.pdf.

[21] Holger M. Kienle. Exchange format bibliography. *ACM SIGSOFT Software Engineering Notes*, 26(1):56–60, January 2001.

[22] Holger M. Kienle and Hausi A. Müller. Leveraging program analysis for web site reverse engineering. *3rd IEEE International Workshop on Web Site Evolution (WSE'01)*, pages 117–125, November 2001.

[23] Holger M. Kienle, Hausi A. Müller, and Johannes Martin. Dependencies analysis of Azureus with Rigi: Tool demo challenge. *4th IEEE International Workshop on Visualizing Software for Understanding and Analysis (VISSOFT 2007)*, pages 159–160, June 2007.

[24] Holger M. Kienle and Xiaomin Wu. Report for the Sortie structured tool demonstration. Technical report, University of Victoria, 2001. http://holgerkienle.wikispaces.com/SortieToolDemo.

[25] Rainer Koschke. *Atomic Architectural Component Recovery for Program Understanding and Evolution*. PhD thesis, University of Stuttgart, Germany, 2000.

[26] Rainer Koschke. Software visualization in software maintenance. *Journal of Software Maintenance and Evolution: Research and Practice*, 15(2):87–109, March 2003.

[27] Bernt Kullbach and Andreas Winter. Querying as an enabling technology in software reengineering. *3rd IEEE European Conference on Software Maintenance and Reengineering (CSMR'99)*, pages 42–50, March 1999.

[28] Michele Lanza. Codecrawler—lessons learned in building a software visualization tool. *7th IEEE European Conference on Software Maintenance and Reengineering (CSMR'03)*, pages 1–10, March 2003.

[29] Michele Lanza and Stephane Ducasse. Polymetric views—a lightweight visual approach to reverse engineering. *IEEE Transactions on Software Engineering*, 29(9):782–795, September 2003.

[30] Timothy C. Lethbridge, Sander Tichelaar, and Erhard Ploedereder. The dagstuhl middle metamodel: A schema for reverse engineering. In J.-M. Favre, M. Godfrey, and A. Winter, editors, *International Workshop on Meta-Models and Schemas for Reverse Engineering (ateM'03)*, volume 94, pages 7–18. Elsevier, May 2004.

[31] Mircea Lungu, Michele Lanza, and Tudor Girba. Package patterns for visual architecture recovery. *10th IEEE European Conference on Software Maintenance and Reengineering (CSMR'06)*, pages 183–192, March 2006.

[32] Johannes Martin. Leveraging IBM VisualAge for C++ for reverse engineering tasks. *Conference of the Centre for Advanced Studies on Collaborative Research (CASCON'99)*, pages 83–95, November 1999.

[33] Johannes Martin and Ludger Martin. Web site maintenance with software-engineering tools. *3rd IEEE International Workshop on Web Site Evolution (WSE'01)*, pages 126–131, November 2001.

[34] Johannes Martin, Bruce Winter, Hausi A. Müller, and Kenny Wong. Analyzing xfig using the Rigi tool suite. *7th IEEE Working Conference on Reverse Engineering (WCRE'00)*, pages 207–209, November 2000.

[35] Daniel L. Moise and Kenny Wong. An industrial experience in reverse engineering. *10th IEEE Working Conference on Reverse Engineering (WCRE'03)*, pages 275–284, November 2003.

[36] Leon Moonen. Generating robust parsers using island grammars. *8th IEEE Working Conference on Reverse Engineering (WCRE'01)*, pages 13–22, October 2001.

[37] Leon Moonen. Lightweight impact analysis using island grammars. *10th IEEE International Workshop on Program Comprehension (IWPC'02)*, pages 219–228, June 2002.

[38] H. A. Müller, S. R. Tilley, M. A. Orgun, B. D. Corrie, and N. H. Madhavji. A reverse engineering environment based on spatial and visual software interconnection models. *5th ACM SIGSOFT Symposium on Software Development Environments (SDE 5)*, pages 88–98, December 1992.

[39] Hausi A. Müller. *Rigi - A Model for Software System Construction, Integration, and Evolution based on Module Interfaces Specifications*. PhD thesis, Rice University, Houston, Texas, August 1986.

[40] Hausi A. Müller, J. R. Möhr, and Jim G. McDaniel. Applying software re-engineering techniques to health information systems. In T. Timmers and B. Blums, editors, *Software Engineering in Medical Informatics: Proceedings of the IMIA Working Conference on Software Engineering in Medical Informatics (SEMI)*, pages 91–110. North-Holland Publishing, 1992.

[41] Hausi A. Müller, Mehmet A. Orgun, Scott R. Tilley, and James S. Uhl. A reverse-engineering approach to subsystem structure identification. *Journal of Software Maintenance: Research and Practice*, 5(4):181–204, July/August 1993.

[42] Gail C. Murphy and David Notkin. Lightweight lexical source model extraction. *ACM Transactions on Software Engineering and Methodology*, 5(3):262–292, July 1996.

[43] John Mylopoulos, Martin Stanley, Kenny Wong, Morris Bernstein, Renato De Mori, Graham Ewart, Kostas Kontogiannis, Ettore Merlo, Hausu Muller, Scott Tilley, and Marijana Tomic. Towards an integrated toolset for program understanding. *Conference of the Centre for Advanced Studies on Collaborative Research (CASCON'94)*, pages 19–31, October 1994.

[44] Frances J. Newbery. An interface description language for graph editors. *IEEE Symposium on Visual Languages (VL'88)*, pages 144–149, October 1988.

[45] Stephen C. North and Elefherios Koutsofios. Applications of graph visualization. *Graphics Interface'94*, pages 235–245, 1994.

[46] Liam O'Brien. Architecture reconstruction to support a product line effort: Case study. Technical Note CMU/SEI-2001-TN-015, Software Engineering Institute, Carnegie Mellon University, July 2001. `http://www.sei.cmu.edu/pub/documents/01.reports/pdf/01tn015.pdf`.

[47] Liam O'Brien and Vorachat Tamarree. Architecture reconstruction of J2EE applications: Generating views from the module viewtype. Technical Note CMU/SEI-2003-TN-028, Software Engineering Institute, Carnegie Mellon University, November 2003. `http://www.sei.cmu.edu/pub/documents/03.reports/pdf/03tn028.pdf`.

[48] John K. Ousterhout. Scripting: Higher-level programming for the 21st century. *IEEE Computer*, 31(3):23–70, March 1998.

[49] Thomas Panas, Jonas Lundberg, and Welf Löwe. Reuse in reverse engineering. *12th IEEE International Workshop on Program Comprehension (IWPC'04)*, pages 52–61, June 2004.

[50] Frances Newbery Paulish and Walter F. Tichy. Edge: An extensible graph editor. *Software—Practice and Experience*, 20(S1):S1/63–S1/88, June 1990.

[51] Claudio Riva. Reverse architecting: an industrial experience report. *7th IEEE Working Conference on Reverse Engineering (WCRE'00)*, pages 42–50, November 2000.

[52] Claudio Riva. *View-based Software Architecture Reconstruction*. PhD thesis, Vienna University of Technology, October 2004.

[53] Claudio Riva and Jordi Vidal Rodriguez. Combining static and dynamic views for architecture reconstruction. *6th IEEE European Conference on Software Maintenance and Reengineering (CSMR'02)*, pages 47–55, March 2002.

[54] Margaret-Anne D. Storey, Susan Elliott Sim, and Kenny Wong. A collaborative demonstration of reverse engineering tools. *ACM SIGAPP Applied Computing Review*, 10(1):18–25, Spring 2002.

[55] Nikita Synytskyy, Richard C. Holt, and Ian Davis. Browsing software architectures with LSEdit. *13th IEEE International Workshop on Program Comprehension (IWPC'05)*, pages 176–178, May 2005.

[56] Tarja Systä, Kai Koskimies, and Hausi Müller. Shimba—an environment for reverse engineering Java software systems. *Software—Practice and Experience*, 31(4):371–394, 2001.

[57] Tarja Systä, Ping Yu, and Hausi Müller. Analyzing Java software by combining metrics and program visualization. *4th IEEE European Conference on Software Maintenance and Reengineering (CSMR'00)*, pages 199–208, February 2000.

[58] Alexandru Telea, Alessandro Maccari, and Claudio Riva. An open toolkit for prototyping reverse engineering visualizations. *Symposium on Data Visualization 2002 (VISSYM'02)*, pages 241–249, May 2002.

[59] Alexandru Telea, Alessandro Maccari, and Claudio Riva. An open visualization toolkit for reverse architecting. *10th IEEE International Workshop on Program Comprehension (IWPC'02)*, pages 3–10, June 2002.

[60] Scott R. Tilley. Domain-retargetable reverse engineering II: Personalized user interfaces. *10th IEEE International Conference on Software Maintenance (ICSM'94)*, pages 336–342, September 1994.

[61] Scott R. Tilley, Hausi A. Müller, Micheael J. Whitney, and Kenny Wong. Domain-retargetable reverse engineering. *9th IEEE Conference on Software Maintenance (CSM'93)*, pages 142–151, September 1993.

[62] Scott R. Tilley, Michael J. Whitney, Hausi A. Müller, and Margaret-Anne D. Storey. Personalized information structures. *11th ACM International Conference on Systems Documentation (SIGDOC'93)*, pages 325–337, October 1993.

[63] Scott R. Tilley, Kenny Wong, Margaret-Anne D. Storey, and Hausi A. Müller. Programmable reverse engineering. *International Journal of Software Engineering and Knowledge Engineering*, 4(4):501–520, December 1994.

[64] Scott Robert Tilley. *Domain-Retargetable Reverse Engineering*. PhD thesis, Department of Computer Science, University of Victoria, 1995.

[65] Anthony I. Wasserman and Peter A. Pircher. A graphical, extensible integrated environment for software development. *ACM SIGPLAN Notices*, 22(1):131–142, January 1987.

[66] Kenny Wong. *Rigi User's Manual: Version 5.4.4*. Department of Computer Science, University of Victoria, June 1998.

[67] Kenny Wong, Scott R. Tilley, Hausi A. Müller, and Margaret-Anne D. Storey. Structural redocumentation: A case study. *IEEE Software*, 12(1):46–54, January 1995.

[68] Jingwei Wu and Margaret-Anne D. Storey. A multi-perspective software visualization environment. *Conference of the Centre for Advanced Studies on Collaborative Research (CASCON'00)*, pages 15–24, October 2000.