Rigi—An environment for software reverse engineering, exploration, visualization, and redocumentation

Holger M. Kienle, Hausi A. Müller

University of Victoria, Canada

ARTICLE INFO

Article history:
Received 30 November 2008
Received in revised form 15 October 2009
Accepted 30 October 2009
Available online 10 November 2009

Keywords:
Reverse engineering
Program comprehension
Tool-building
Tool requirements

ABSTRACT

The Rigi environment is a mature research tool that provides functionality to reverse engineer software systems. With Rigi, large systems can be analyzed, interactively explored, summarized, and documented. This is supported with parsers to extract information from source code, an exchange format to store extracted information, analyses to transform and abstract information, a scripting language and library to automate the process, and a visualization engine to interactively explore and manipulate information in the form of typed, directed, hierarchical graphs. In this paper, we describe Rigi’s main components and functionalities, and assess its impact on reverse engineering research. Furthermore, we discuss Rigi’s architecture and design decisions that led to a decoupling of major functionalities and enable tool extensibility, interoperability and end-user programmability.

© 2009 Elsevier B.V. All rights reserved.

1. Introduction

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 wide spectrum of different tasks. Examples of such tasks are: architecture recovery, clustering, slicing, metrics gathering, and business rule extraction.

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 usually highly desirable to automate reverse engineering activities. The Rigi environment provides such tool support. The reverse engineering community has developed many prominent reverse engineering environments and tools including Bauhaus (Universities of Stuttgart and Bremen) [1,2], Ciao/CIA (AT&T Bell Labs) [3,4], Columbus (University of Szeged) [5], GUPRO (University of Koblenz) [6,7], Moose/CodeCrawler (Universities of Bern and Lugano) [8,9], PBS (University of Waterloo) [10,11], SHriMP (University of Victoria) [12,13], SolidFX (University of Groningen and SolidSource BV) [14], and SQuAVisiT (Technische Universiteit Eindhoven) [15,16]. Rigi was one of the first well-documented and widely used tools in this twenty year evolution of reverse engineering tools.

Rigi enables us to reverse engineer large software system based on the following key features. The subject system is described with the help of an exchange format that allows us to model information based on entity and binary relationship types. Typically, the information model is based on the higher-level programming language of the subject system and encodes static, higher-level information such as procedures, global variables, procedure calls, and global variable accesses. Rigi has a number of pre-defined models (e.g., for C and C++) and accompanying parsers that can extract information from the subject system and store this information in Rigi’s exchange format. Rigi has a visualizer that enables the reverse engineer...
to view the extracted information from the subject system as a directed graph where entities are represented as nodes and relationships as arcs between two nodes. This graph can be iteratively explored and manipulated by the reverse engineer (e.g., by zooming, layouting, filtering, and summarizing). The visualizer has a multi-window approach that allows the reverse engineer to create different views of the graph. These views are synchronized; for example, selecting a certain node in one view also selects the node in all the other views.

Rigi supports an approach to reverse engineering called structural redocumentation that is based on layered subsystem hierarchies. The reverse engineer identifies sets of lower-level entities that have commonalities based on clustering criteria and groups them into subsystems. This grouping is iteratively and recursively refined until the subject system is described at the desired levels of abstraction. The grouping of entities can be performed interactively by the reverse engineer in Rigi’s visualizer. Alternatively, groupings can be produced (semi-)automatically with the help of a scripting language (e.g., via coding of a clustering algorithm that groups entities based on their names, or based on coupling and cohesion metrics). Thus, clusterings (or other reverse engineering techniques) can be scripted and then reused in many related reverse engineering activities.

Structural redocumentation is supported by the visualizer with hierarchical graphs. Nodes can be grouped into so-called supernodes. A supernode is rendered like an atomic node, but can be expanded in a new graph window to reveal its subnodes. Furthermore, the supernode hierarchy can be visualized in a special tree view. Rigi’s approach to reverse engineering has been investigated in a number of (industrial) case studies. The case studies attest that Rigi is capable of supporting reverse engineers in program comprehension and architecture recovery based on structural redocumentation.

The rest of the paper is organized as follows. Section 2 describes Rigi’s functionality, covering the repository and data model, the fact extractors, and the graph-based, interactive editor. Section 3 provides a brief example of structural redocumentation of the Azureus BitTorrent client. Section 4 addresses Rigi’s availability and user support. Section 5 summarizes Rigi’s impact on reverse engineering research and tools in academia and industry. Section 6 distills tool-building experiences and lessons learned in the design and implementation of Rigi. Finally, Section 7 closes the paper with our conclusions.

2. The Rigi environment

The typical workflow for the reverse engineering of software systems can be characterized by three main activities: extract, analyze, and visualize [17]. A reverse engineering tool should provide support for all three activities. Furthermore, a repository is needed to store reverse engineering information.

Rigi’s conceptual architecture is depicted in Fig. 1 and reflects these requirements. The architecture exposes Rigi’s main functionalities: extraction of static facts from software systems (cf. Section 2.2), a repository to represent and store facts (cf. Section 2.1), and analyses and visualization of facts (cf. Section 2.3). We discuss each of the main entities in the following sections.

The architecture shows two extractors for C++ (Rigi VisualAge parser [18]) and web sites (Web2Rsf parser [19]) that exemplify that there can be an open-ended number of different extractors for various domains of structured text (e.g., programs in various programming languages, abstract syntax trees (ASTs), assembly code, cross reference lists, hierarchical documentation, or other structured information spaces). 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, the Rigi Standard Format (RSF). The fact extractors (implemented as scanners or parsers) write their output into an RSF file, which can then be read in by the graph editor, Rigiedit.
In contrast, Rigi's analyses and visualization are intertwined within Rigiedit. Analyses are written with the Tcl-based Rigi Command Language (RCL) and are stored in separate text files. The RCL library uses a Tcl-to-C binding to access and manipulate Rigi's internal, in-memory graph model, which is implemented in C/C++ to optimize performance. Analyses can be interactively loaded and invoked from Rigiedit (e.g., via selecting an entry from a pull-down menu). Once an analysis is invoked, it can update the graph model; for instance, a clustering analysis can create new supernodes 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.

While the extractors in Rigi operate batch-style (i.e., all source artifacts are processed in one run without user intervention), Rigiedit is an example of a human-involved tool because it performs automated analyses on request, whose results can then be interactively manipulated and refined by the user.

2.1. Repository and data model

The most central component of Rigi is the repository. It gets populated with facts extracted from the subject software system. Analyses read information from the repository and possibly augment it with further information.

Examples of concrete implementations of repositories range from textual or XML files to commercial databases. Rigi follows a lightweight approach by defining a file-based, textual exchange format called RSF. As a result, the fact base is contained in a single file, which can be imported an exported by Rigiedit.

RSF relies on sequences of tuples to encode graphs [21]. 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 TA [22], Bauhaus' Resource Graph [23], GUPRO's GraX [24], FAMIX's MSE, and the Graph Exchange Language (GXL) [25].

In order to model hierarchical graphs, nodes of the pre-defined type Collapse can be created, which can contain subnodes that are connected to the supernode via level arcs. For example, to group f1 and f2 into a function set supernode the following lines can be added to the above example:

```
type functionset Collapse
level functionset f1
level functionset f2
```

This technique can be applied recursively to create hierarchical graphs of any depth. In Rigiedit, a Collapse node can be double-clicked, which opens up a new graph window to reveal the node's immediate subnodes.

Besides the RSF introduced above (also called unstructured RSF), there is also a more storage efficient version of RSF (called structured RSF). Structured RSF introduces numerical IDs that can be used to refer to nodes and arcs. This binding is established with the following syntax: id!name. Structured RSF also expects a top-level Root node of type Collapse. To easily distinguish unstructured from structured RSF, the latter prefixes node and arc definitions with n! and a!, respectively.

The above example is encoded in structured RSF as follows:

```
n!type 1!Root Collapse
n!type 2!functionset Collapse
a!level 1 2
n!type 3!f1 Function
n!type 4!f2 Function
a!level 2 3
a!level 2 4
a!calls 3 4
```

Typically, fact extractors produce unstructured RSF because it is easier to generate and does not require to manage IDs. Rigiedit supports both structured and unstructured RSF. Thus, users can convert unstructured RSF to structured RSF by loading it first into Rigiedit and then saving the graph.

The facts that are stored in RSF adhere to a certain data model (or schema). An 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 Rigiedit. 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 Rigiedit) 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 (file Riginode)
Function

# arc types (file Rigiarcs)
calls Function Function
```

The Rigiedit file editor does not support type inheritance. As a consequence a node or arc represents exactly one type. To allow for needed flexibility, Rigi has a pre-defined node type called Unknown which can be used to model arcs that have multiple potential source or target node types. For example, a data model for Ada may introduce Function and Procedure node types. Since a calls arc can have source and target destinations of either types they have to be specified as Unknown. While this approach lacks expressiveness and reduces opportunities for conformance checks, it simplifies the specification and has worked well in practice.

Rigi defines (standard) data models for C, C++, and Cobol. These data models are capable of representing middle-level information [26]. A middle-level model focuses on the main program elements (functions, global variables, etc.) and their relationships (calls to functions, references to variables, etc.). Thus, on the one hand such a model omits low-level details of code as could be found in abstract syntax trees (ASTs), and on the other hand does not abstract information all the way up to generic architectural elements (i.e., components and connectors).

### 2.2. 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).

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 (vcppparse) that is built on top of IBM’s Visual Age compiler [18]. There are also fact extractors developed by other groups that support Rigi (e.g., Columbus’s C/C++ parser and SHriMP’s Java extractor).

Additional fact extractors have been developed as well for research projects and case studies, but those are not publicly available. For example, to analyze software documentation a scanner for BibTeX was developed [27], and to analyze the HTML of web sites a Java-based web crawler was written [19]. In contrast, an ad hoc approach was taken to extract facts from PL/AS code based on lexical pattern matching with Unix scripts [28]. These scripts extract higher-level structural information (e.g., call graphs) and represent them as C code, which can then be processed 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. This strategy is especially useful if the targeted language is not supported by Rigi’s extractors. For example, to analyze the Azureus BitTorrent client (written in Java) we used facts provided in FAMIX/MSE and wrote a Perl script to translate them into RSF [29].

### 2.3. Graph-based editor

The core of Rigi is a graph editor, Rigiedit, enhanced with additional functionality for reverse engineering tasks. Examples of reverse engineering functionality in the editor are the computation of cyclomatic complexity and the navigation to the underlying source code. Rigiedit’s user interface consists of a workbench and an open-ended number of graph windows that depict different views of the underlying graph data structure. **Fig. 2** shows a screenshot of Rigiedit with the workbench (top right), two graph windows (center and bottom left), and two dialogs (bottom right). While Rigiedit conveys most information graphically, there are also 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.

\[2\] 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.
The workbench consists of pull-down windows, shortcuts to frequently used menu operations, information about the domain, and access to RCL scripting. Rigi’s core functionality is similar to the functionality offered by generic graph editors. Graphs encoded with RSF and snapshots of graph windows can be saved and loaded (File menu). Selected nodes can be cut, copied and pasted as well as collapsed into a supernode (Edit menu). A selected supernode can be also un-collapsed with the Expand operation (Edit menu). Nodes can be selected by name or attribute values, or by using operations on already selected nodes such as outgoing/incoming nodes and forward/reverse tree (Select menu). The nodes in a graph window can be scaled (Scale menu), and can be laid out with different algorithms such as grid, tree, Spring, or Sugiyama (Layout menu).

In the graph windows, nodes and arcs can be interactively manipulated. Fig. 2 shows two graph windows: the main graph depicted in the center with 469 nodes and a subgraph with 41 nodes at the bottom left. In Rigi, nodes are drawn as squares, and arcs originate at the bottom of a square and end at the top of a square (i.e., there are no arrows to indicate direction). As can be expected, nodes can be selected by clicking on them and moved by dragging them. Right-clicking on a node or arc opens up a context menu that allows editing properties (i.e., type, name, attributes, and textual annotations).

Hierarchical graphs are a key concept in Rigi to support structural redocumentation. Typically, the reverse engineer starts out with a flat graph that contains only atomic nodes. For IBM’s SQL/DS system (which is over three million lines of PL/AS code) the flat graph that represents the middle-level model of the system contains 923 atomic nodes and 2395 arcs between atomic nodes. Grouping nodes into supernodes allows the reverse engineer to impose structure on a flat graph. The main graph depicted in Fig. 2 shows a hierarchical graph of SQL/DS that is the result of clustering atomic nodes into supernodes, which represent components. In this view only supernodes contain textual labels that show the components’ names so that they can be easily distinguished from atomic nodes. As a result of the clustering, this view has 469 nodes. This is a rudimentary example that illustrates how hierarchical graphs can reduce complexity. Section 3 explains with a more detailed example how hierarchical graphs enable reverse engineering with structural redocumentation.

Rigi supports hierarchical graphs in a number of ways. In a graph window, double-clicking on a node reveals its subnodes in a new graph window. The subgraph in Fig. 2 is the result of double-clicking on the C1-ARIXIFP node in the main graph. The Children operation in the Navigate menu is more powerful in that it allows us to show the subnodes of several selected nodes. Conversely, there is a Parents operation that shows the supernodes of selected nodes. The hierarchical structure starting from a certain node can be exposed in a dedicated tree view by first selecting the node and then choosing Overview from the Navigate menu. This tree view is composed by recursively following the 1 level arcs. For example, Fig. 3 shows the tree view of the SQL/DS system.

Each graph that is loaded into Rigi has to adhere to a certain domain model (cf. Section 2.1). Depending on the domain model, different node and arc types (along with different color coding) are available in Rigi. The workbench in Fig. 2 has a button labeled “Domain: plas” to show that the domain that models PL/AS code is active. Clicking on the Domain button allows the user to switch to a different domain. In the PL/AS domain, red nodes are variables (of type data), yellow nodes are modules (module), and purple nodes are data types (struct). There are also green nodes that depict components (subsystem) as explained above.

Rigi customizes its functionality to match the domain. For example, the dialogs for filtering reflect the domain’s node and arc types. Both filter dialogs for the PL/AS domain are shown in Fig. 2, bottom right. The main graph in Fig. 2 is the result of a filtering operation where all arcs between atomic nodes have been filtered out except for the yellow ones (i.e., arcs of
type call), which represent calls between modules or components (i.e., the view shows the system’s call graph, partially hoisted to the level of components).

Many reverse engineering tools now use graphs to convey information [30]. Examples of visualizers that have been developed by researchers for reverse engineering and program comprehension are ARMIN [31], CodeCrawler [9], GSEE [32], LSEdit [33], Sextant [34], SoftVision [35], VANISH [36], and work by Holten [37]. Besides these editors, there are also general-purpose graph layouters and editors. The EDGE graph editor is an early example [38]. AT&T’s Graphviz provides an interactive editor, dotty, which can be customized with a dedicated scripting language called lefty [39].

2.3.1. End-user programmability

The editor provides support for scripting with the Rigi Command Language (RCL). RCL is a scripting layer between the GUI and the underlying graph model implementation [40]. RCL is a collection of over 400 Tcl commands (that are indistinguishable from Tcl’s 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. All graph functionalities that can be invoked from the interactive GUI are also accessible via RCL commands. For example, in Fig. 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:

```tcl
# set arc filter for type "data" in window 3 (cf. main graph in Fig. 2)
rcl_filter_arctype_filtered data 3
# apply the arc filter for window 3
rcl_filter_apply 3 arc
```

Rigi has an RCL console as part of the workbench that allows the user to type in RCL commands, to select them from a command list, or to source 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, loading user preferences, or an automated demo). 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 [41] – 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 are free to ignore RCL and its features when using the editor interactively.

3. An example of structural redocumentation

In the previous section we have described Rigi’s functionality. We now turn to a brief example that illustrates how this functionality can be leveraged by the reverse engineer for structural redocumentation. Structural redocumentation is an iterative process that groups lower-level entities into higher-level ones until a representation is obtained that has a suitable level of abstraction. The actual representation and its abstraction level depends on the goals of the reverse engineering task.

Typically, lower-level representations contain entities that directly relate to the source code (e.g., functions and function calls) while higher-level entities represent functional abstractions, abstract data types, or components. The highest abstraction is typically a high-level architecture of the subject system. In the following example, we show how Rigiedit can be used to obtain a high-level architectural view of the Azureus BitTorrent client [29].

Azureus is written in Java and has more than 300,000 lines of code. The resulting RSF file that represents Azureus has 32,677 unique nodes and edges. Since we are interested in higher-level abstractions, the Java domain contains only nodes of type Class. Inheritance between classes is modeled with inheritsFrom arcs. Arcs that represent method calls (invokes) and field accesses (accesses) are hoisted up to the class level. To make sure that class nodes are unique, their names are fully qualified with their package names (e.g., java::lang::Object).
## Group all nodes whose name starts
## with 'prefix' into a new Collapse
## node with name 'label'

```tcl
proc cluster {prefix label} {
    rcl_select_none
    rcl_select_grep "\$prefix.*"
    set winnodes [rcl_select_get_list]
    set num [rcl_select_num_nodes]
    if { $num > 1 } {
        rcl_collapse Collapse
        "$label ($num)"
    }
}
```

## Create a high-level clustering
## for Azureus

```tcl
# Delete unwanted classes
cluster "java::lang" "dummy"
rcl_cut
...
# Group other packages
cluster "com::" "Azureus com Packages"
cluster "org::" "Azureus org Packages"
cluster "javax::crypto" "javax::crypto"
    ...
# Apply Sugiyama layout
sugiyama
```

### Fig. 4. RCL script to obtain an architectural view of Azureus.

### Fig. 5. Architectural view of Azureus that shows the dependencies to external classes.

When the Azureus RSF file is loaded into Rigi, all 4713 Class nodes are presented in a single view together with all relationships. The resulting view is not suitable for comprehending the system's architecture. The user of Rigiedit has to create new views of the system that reduce complexity via available mechanisms such as selecting, laying out, filtering, and grouping. For example, the user could select all classes that are part of a certain package and create a new view that contains only these nodes. Next, the user could use filtering to show only the inheritance relationships. Finally, the user could use laying out (e.g., forward tree) to improve the presentation of the inheritance trees. The perhaps most powerful mechanisms to reduce complexity is grouping of class nodes into supernodes that represent a certain concept, service or functionality.

Finding the right strategy to come up with meaningful views of a system can be seen as a creative act. For Azureus, one strategy that we used is hierarchical clustering based on the Java package structure. **Fig. 4** (left side) shows an RCL script that defines a cluster procedure that takes two parameters, prefix and label. The procedure can be used to group class nodes whose name starts with a common prefix into a new supernode. The name of the supernode is composed of the label and the number of the classes that it contains. Once the cluster procedure has been defined (e.g., by sourcing the file that contains the RCL code), the reverse engineer can invoke it from the Rigi workbench.

An example of a clustering based on the cluster procedure is shown in **Fig. 5**. Azureus has two top-level packages: com and org. A simple high-level clustering is created with a supernode for each top-level package (com(1064) and org(3127)). The remaining classes after this step represent external classes (contained in standard and third-party packages) that are called by Azureus. These classes are similarly clustered (or deleted if they are not relevant for comprehension). In this view, one can see that both com and org packages depend on each other. (This refutes the hypothesis that com packages constitute extra functionality that is layered on top of the org packages.) It is also apparent that while both com and org packages rely on networking functionality (java::net(16)), only org packages have UI functionality (java::awt(17) and java::swing(7)) and only com packages need cryptography support (java::crypto(23)). In practice, the reverse engineer interactively experiments with different clusterings until a meaningful result is obtained. Then, these steps can be encoded in RCL (cf. right side of **Fig. 4**).
Due to space constraints, we cannot show more examples of architectural views. Another example of an architectural view of Azureus is discussed elsewhere that groups packages according to three functionalities: user interface, plugins, and core infrastructure [29]. Also, Kazman and Carrière show with two case studies how stepwise refinement of views can be achieved [42]. They use Rigiedit to visualize the views. The initial views are “white noise” due to the large number of nodes and arcs. The final views after a number of iterations show high-level architectures with about a handful of components.

4. Availability and 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. Fig. 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 stated that the “best-documented software visualization tool we know is Rigi” [43]. 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 [21]. Rigiedit comes with three sample systems of increasing complexity that demonstrate how a software system can be reverse engineered with Rigi. The user runs these interactive demos directly within the editor.3 There is also a Wiki4 that allows users of Rigi to add information about Rigi and their experiences with it. The most useful resource currently available on the Wiki is the Frequently Asked Questions (FAQ).

5. Impact and relevance of Rigi

Rigi is a mature research tool that is in its current form now more than a decade old. Active development of Rigi stopped in 1999, but occasional maintenance and bug fixes were performed as well as porting to new Windows versions. The last official version of Rigi was released in 2003. In the following, we first track Rigi’s historic research contributions and then try to assess its relevance today. Rigi’s historic contributions are as follows:

software visualization with graphs: 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 [44]. An important feature of Rigi’s graph-based model is the introduction of hierarchical graphs for creating higher-level abstract representation of software systems [45]. 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 and implementation of software systems (i.e., forward engineering). For example, Software through Pictures used graphical editors to represent data-flow diagrams, entity–relationship models, and data structure definitions [46].

case studies and tool comparisons: A significant number of reverse engineering case studies have been conducted with Rigi. Some of these studies have been reported in the literature (e.g., [29,47–50,45]). Rigi has been used in industrial contexts as well (e.g., [28,51–54]). These case studies cover a broad range of systems and languages, including intermediate code and assembly languages, and attest to Rigi’s effectiveness to support reverse engineering tasks. Furthermore, the case studies were useful to identify improvements for Rigi in functionality, scalability, and user interaction [55–57]. Rigi has been the subject of at least four tool comparisons. Armstrong and Trudeau compared five tools (i.e., CIA, Dali, PBS, Rigi, and SNIFF+) with respect to their effectiveness in recovering the architecture of a system [58]. Bellay and Gall provide a detailed feature comparison of four tools (i.e., Imagix 4D, Refine/C, Rigi, and SNIFF+), applying the tools to an embedded system of 150K lines of code [59]. Gannod and Cheng define a tool classifying framework and apply it to compare seven commercial and nine research tools [60]. Interestingly, Rigi is the only research tool in the study that did offer an “open interface”, or API. Most recently, dos Santos Brito et al. have compared eight research tools (i.e., Scruple, Rigi, TkSee, SHriMP, DynaSee, GSEE, CodeCrawler, and Sextant) based on six functional and three non-functional requirements [61]. In this comparison, both Rigi and TkSee are meeting the most requirements (i.e., seven out of nine).

reverse engineering methodology: Rigi’s methodology of structural redodocumentation [45,28] has been successfully applied to and refined with a number of case studies (see above). 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 us to automate lower-level,
Rigi's RSF has been specifically designed to meet the requirements of a broad range of reverse engineering tasks that are otherwise tedious to perform and distracting for reverse engineers. Consequently, Rigi 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 [62].

exchange format: Rigi's RSF has been specifically designed to meet the requirements of a broad range of reverse engineering tools. In fact, it has been adopted or is supported by a number of other tools as well (e.g., [63–66, 31, 5, 67, 68, 1]). Several of these tools are mature products that are in use today and continue to support RSF. In fact, Gorton and Zhu state that “RSF has become the de facto standard for representing source information” [69].

RSF has also inspired other exchange formats in the reverse engineering domain such as Holt's TA [22] and GXL [25]. Another contribution of RSF is that it introduces a separation between the actual data and the data model, exploratory research and tool prototypes: The Rigi environment is also a tool platform that enables exploratory research in the reverse engineering domain. Especially, Rigi enables the rapid prototyping of novel analyses and visualizations with its domain-specific scripting language.

Several tools have been implemented on top of Rigi, most notably Dali [42, 70], Bauhaus [1, 71], Shimba [72, 73], and Nimeta [66, 74]. All of these tools provide major enhancements to Rigi's standard functionality. Dali has been partially developed within the Software Engineering Institute (SEI) and applied to industrial systems. The other three Rigi-based tools have been developed as part of dissertation works from Koschke [1], Systä [72] and Riva [66], contributing to reverse engineering research by realizing novel analyses and visualizations for program comprehension.

Rigi's versatility in customizing it to different reverse engineering domains is illustrated by an extension of Rigi for the understanding and maintenance of static Web sites [75, 19].

In summary, the major novel contributions that Rigi has made during its active development are (1) visualization of software structures with an interactive graph editor, (2) a reverse engineering methodology dubbed structural redocumentation, (3) encoding of reverse engineering data with a dedicated exchange format, (4) automating reverse engineering tasks using a scripting language and library, and (5) a tool extensibility mechanism based on scripting and levels of indirection that enables effective exploratory research via rapid prototyping of new functionality.

The fact that many current tools have incorporated these features attests that Rigi's contributions have had lasting impact on reverse engineering research and its community. As can be expected, current tools have improved upon these initial ideas. For example, visualization tools are now leveraging advanced graphics facilities such as OpenGL (e.g., [14, 37]), and offer dedicated out-of-the-box visualizations that are immediately useful starting points for program comprehension (e.g., [9, 76]). RSF represents only one point in the vast design space for exchange formats; consequently researchers have proposed other exchange formats that differ in the expressiveness of the schema or are XML-based. Similarly, tools have explored other approaches to tool extensibility (e.g., building on top of sophisticated IDEs such as Eclipse) and making use of scripting languages that are now more popular than Tcl (e.g., Perl).

Due to Rigi's age, it is less obvious how to assess the relevance that Rigi still has. For a current reverse engineering tool, its relevance is established by a unique research contribution such as a novel analysis or visualization technique. However, whether a novel tool's contributions have lasting impact on research can only be established in the long run. Perhaps paradoxically, if a tool has lasting impact then others will mimic its features thus making it no longer unique.

What can be tracked is Rigi's relevance in terms of its appearance in the literature compared to other tools. Table 1 shows a subset of popular reverse engineering tools and how often each tool's name has been mentioned in IEEE publications. We group the results into magazines and transactions on the one hand and conference proceedings on the other hand. To determine the scores, we did use the Advanced Search feature of the IEEE Computer Society site, searching for the tool's name (appearing as exact phrase) and the term “reverse engineering” (also exact phrase). As can be seen from Table 1, Rigi has been mentioned numerous times in both groups. Rigi's current relevance is supported by the fact that it is also mentioned in recent years with 16, 13 and 9 occurrences in the year 2008, 2007 and 2006, respectively.

---

5 When selecting the tools we were limited by considerations such as whether the tool's name was unambiguous in order to minimize false positives in the search results.

6 [http://www2.computer.org/portal/web/search/advanced/](http://www2.computer.org/portal/web/search/advanced/).
Of course, these numbers can only track whether Rigi has been mentioned per se, not whether it has contributed to the reported research. However, we are aware of two scientific contributions involving Rigi that have been reported in the recent past. Jin and Cordy have used Rigi in a study involving the OASIS methodology for tool integration [77], and Moise et al. have implemented an integration of Rigi with Microsoft Visual Studio .NET [78]. Furthermore, we have participated with Rigi in 2007 at the VISSOFT Tool Demo7 along with two other tools and won first prize [29]. In fact, a motivation for participating was the desire to find out if Rigi can produce competitive results. Based on the tool demo and our own experiences, we believe that regardless of Rigi’s shortcomings, its functionality and scalability can still compete with other research tools.

Another indication for a mature tool’s relevance is its adoption by tool users in academia and industry. We would argue that adoption is a key indicator of success because an adopted tool has indirectly proven its usefulness. Also, an adopted tool has somehow managed to circumvent the “many barriers to adoption (e.g., seemingly trivial usability issues can impede usage of a tool)” [79]. It is important to realize that the reasons that get a tool adopted in the end are difficult or impossible to determine—also because these reasons go beyond mere technical issues. For example, even though Rigi has some features that may impede adoptability (e.g., initial ramp-up costs because there are no pre-defined views), there are others that may foster it (e.g., detailed user manual and built-in demos). It seems however that Rigi’s usability and usefulness did tip the scale towards adoption for a significant number of users. We speculate further on the technical decisions that we believe had a positive influence on Rigi’s adoption in Section 6.

We also have anecdotal evidence about adoptions of Rigi. First, Rigi has been or is still being used for teaching reverse engineering topics at several universities including Universities of Alberta, Bologna, Botswana, Catholic University of Brasilia, Saskatchewan, Seattle, Southern California, Stuttgart, Tampere, Victoria, Waterloo, Zhejiang University of Technology, and Sao Paulo. Second, Rigi has been used in several industrial environments including IBM, SEI, Nokia, Klocwork Solutions, Boeing, CP Rail, and NASA.

On the one hand, the fact that Rigi has not been upgraded to the latest look-and-feel of Windows applications is certainly a deficiency. One the other hand, Rigi is one of a few major tools of the nineties that has withstood the test of time and is still downloadable and executable. This availability gives junior researchers a chance to test-drive an established research tool rather than just reading about it in papers, provides a base line against which emerging tools can be measured as well as an opportunity to repeat experiments reported in the literature, and allows senior researchers to go back to their roots.

6. Experiences and lessons learned

This section provides a discussion of Rigi’s approach to reverse engineering and program comprehension. We also distill experiences and lessons learned, analyze what made Rigi a highly successful tool, and document shortcomings and possible improvements. Many of our lessons may be obvious to researchers who are intimately familiar with the reverse engineering domain and who have significant expertise in building reverse engineering tools. However there is value in making these lessons more explicit so that they can serve as a baseline for discussions to advance the field. Also, we hope that these lessons are useful for researchers that are not intimately familiar with the reverse engineering domain.

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 constant jumping and iterating between the central activities of fact extraction, analysis, summarization and abstraction of facts, as well as visualization and exploration of facts. 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 on the spot. Furthermore, reverse engineering activities are quite diverse and depend on many factors. Consequently, reverse engineers have to adapt their tools to meet changing needs continuously. Thus, it is not sufficient for a reverse engineering tool to be general (i.e., it can be used without change in different situations), it also has to be flexible (i.e., it can be easily changed to be applicable in a new situation) [80]. Specifically, flexibility mandates tools that allow users to become productive very quickly, that can be easily customized and extended programmatically, and that are able to interoperate with other tools and fit into the existing working environments and processes of developers.

6.1. Graph visualization

In a sense, the graph editor is the heart of the Rigi system. In many projects, only the editor is used (i.e., ignoring the rest of the Rigi system), and only the editor’s interactive facilities are used (i.e., ignoring the myriad commands in the RCL library). 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 optimize performance for large 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 proven successful not only for performance, but also for maintenance and rapid prototyping. This strategy has also been adopted by other visualizers (e.g., SoftVision [35]). Nowadays, an implementation that is exclusively based on a scripting language, such as Perl, might be feasible, but may exacerbate the maintenance of the system.

Rigi has a fully-featured graph editor, but its graphical capabilities and the look-and-feel of the GUI have not evolved in recent years. As a result, many desirable advances in software visualization are now lacking. For example, for performance reasons, all nodes and arcs have the same shape, and arcs have the same line thickness. Thus, it is impossible to associate metrics with graphical entities in Rigi (e.g., size of nodes and arcs). To work around this limitation, a metrics extension in Rigi (e.g., size of nodes and arcs) to open a textual view to see them [81]. An approach similar to the elegant polymetric views would be definitely desirable [9]. Furthermore, Rigi does not support drag-and-drop operations [82]; 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 modern Windows applications. Lastly, to ease the creation of scripts it would be useful to have a better recording and play-back feature (e.g., analogous to Microsoft Office).

During the reverse engineering of the Azureus system [29] we noticed several deficiencies that make interactive exploration of software structures with Rigi unnecessarily cumbersome:

- 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 [83]) offer arcs with flexible line width.
- Rigi has no undo feature. As a result, one has to manage the creation of views carefully (e.g., by writing scripts or saving views) in order to avoid losing interesting layouts and having to re-create views unnecessarily.
- It is only possible to apply the Spring and Sugiyama layouts to all of the nodes in a view. It would be desirable to apply the layouts to selected nodes only. There are now other visualizers that have more flexibility in supporting mappings from node selection to visualization operations (e.g., SoftVision [35] and Sextant [34]).
- There is no easy way to add a node to a Collapse node. It can be accomplished with a sequence of expand, select, and collapse operations, but this is cumbersome and error-prone.

Another drawback of Rigi is that it provides little guidance for users on how to create useful, basic reverse engineering views of a new subject system effectively (i.e., without resorting to RCL scripts). This is especially a problem for inexperienced users and is detrimental to Rigi’s adoptability. Typically, fact extractors produce flat graphs. When such a graph is loaded into Rigi, the initial graph window shows all nodes and arcs—possibly thousands of them. Transforming this initial graph into meaningful views is a creative act that is not readily accomplished and requires some experience. However, experienced users can find many scripts in the RCL library to organize huge graphs effectively. Other (generic) reverse engineering tools suffer from the same shortcoming (e.g., SoftVision). A promising approach to tackle this problem is to offer standard views for certain domains (e.g., a class hierarchy view for the C++ domain, or a directory-based clustering for the C domain) that provide meaningful starting points for reverse engineering and program comprehension tasks. For example, CodeCrawler offers a number of standard views [9]. In Rigi, such standard views as well as project-specific views are encoded in scripts. Moreover, whenever the underlying subject system evolves (i.e., when the extracted facts change) and if the scripts are sufficiently flexible, then executing the relevant scripts will re-produce all the views.

Rigi’s user interface uses a multi-window approach when interacting with graphs. There are many operations on graphs that result in a new graph window. As a result, a typical Rigi session involves many windows thereby cluttering the display. While Rigi synchronizes the windows, users have difficulty keeping track of the different graph windows. Also, inexperienced users tend to create graph windows with views that are already available. On the other hand, the graph windows can leave bread-crumbs trails that show the user’s navigation history and enable them to resume exploration at a previous state. In contrast, many tools have a fixed number of views. The SHRiMP tools, for instance, has a single graph window. Both approaches have benefits and drawbacks; Schäfer et al. say that “the existence of multiple, nonintegrated views can cause disorientation in the case of Rigi, whereas the SHRiMP visualization can result in an information overload” [84]. Storey et al. have conducted a user experiment that compares Rigi’s multi-window with SHRiMP’s single-window approach [85]. While users favored SHRiMP for smaller graphs, they preferred Rigi for larger graphs. Thus, this experiment could not establish a clear-cut user preference for either approach. Similarly, designers of software development environments are also facing the difficult design decision of single vs. multiple windows. Generally, it seems that older environments such as HP’s SoftBench and Reiss’ Field [86] tend to favor a multi-window approach while more recent commercial IDEs favor a single-window approach. As a result, programmers may be more used to the latter style.

Since other software visualization tools share some of Rigi’s deficiencies, it is important to communicate such experiences to the research community in order to improve upon the state-of-the-art of software visualization tools.

6.2. Fact extraction

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) [87]. On the other hand, they are more flexible and lightweight than parsers [88].
Jackson and Rinard argue that while such lightweight approaches are unsound, they “may provide a useful starting point for reverse engineering and further investigation. Unsound analyses are therefore often quite useful for engineers who are faced with the task of understanding and maintaining legacy code” [89]. 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). In this 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.

Rigi’s C and Cobol extractors are parsers built with the help of a LALR(1) parser generator, Yacc. All parser-based approaches have in common that they are brittle in the sense that they easily break in the face of code anomalies such as syntax errors, dialects, and embedded languages [90]. This is especially a drawback for reverse engineering tools that want to cope with a broad variety of code. Producing a robust parser-based extractor is difficult for languages such as C and Cobol that exhibit many code anomalies, but less so for more “stable” languages such as Java. The former maintainer of Rigi’s C parser relates his experiences as follows [18]:

“The Rigi C parser consists of more than 4000 lines of Lex, Yacc, and C++ code. [Its] 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”.

The first author made similar experiences for the Bauhaus tool’s C parser that is based on a LR-attributed grammar.

In practice, code with anomalies frequently needs to be “tweaked” so that it can be processed by parsers. Often it is sufficient to supply pre-processor directives that suppress or change C constructs.8 While these tweaks may be small, they often require intimate knowledge of the extractor’s parsing algorithm and the subject system’s build process. As a result, we regularly receive email from users of Rigi that complain that they cannot get its C parser to run through.

Based on our experiences with both parsers and scanners, we advocate a lightweight approach to fact extraction that opportunistically collects facts about the legacy system. It has worked well for us, 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” [28]. Lightweight approaches seem to work best for extracting facts at the middle-level (e.g., call graphs)—this matches Rigi’s graph model which is also aimed at the middle-level (cf. Section 2.1). While lightweight extraction has many benefits, there is no direct support in Rigi to build such an extractor. Users are expected to produce an RSF file (and if needed to define a suitable domain specification) that can then be processed with Rigidit. Thus, users are free to employ an approach that best suits their needs. Besides standard Unix tools and scripting languages, user can leverage dedicated lightweight approaches (e.g., hierarchical lexical analysis, fuzzy parsing, and island grammars) [90,87].

6.3. Data representation and storage

RSF is a lightweight exchange format that is characterized by the following properties. It is a text-based format without nested 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 [91].

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 [92]. 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.9 For example, the following Unix pipes and filters command sequence will determine the arc types and the number of their occurrences in an RSF file:

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

The text-based approach also makes it easy to combine RSF with a version control system and to perform 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 2.1) are often not as expressive as one might expect. Favre did attempt to use RSF in an industrial context but found “its support to meta-models (called domains) too weak” for his purpose [94]. 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 qualification must be made 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 fully qualified name can easily be extracted with Unix tools. For object-oriented languages, the qualified names can become relatively long and somewhat difficult to read compared to

---

8 For example, the tweaks to parse the sources of the Mozilla web browser with cparser are described at http://www.rigi.cs.uvic.ca/brig/mozilla/scripts.html.

9 However, the CrocoPat tool developed by Beyer et al. offers a domain-specific language based on predicate calculus for query and manipulation of RSF relations [93].
procedural languages. Other formats avoid name qualification by supporting name scoping and/or assigning unique IDs to entities (e.g., GXL and FAMIX/MSE). There is a fine line in exchange formats between expressiveness on the one hand and simplicity on the other hand—and in some respects 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, RSF’s emphasis on readability combined with the need to qualify names means that it is storage inefficient. For example, for the Azureus case study [29] the system is represented with a middle-level model that consists of 32,677 nodes and edges, resulting in an RSF file of 11 MBytes.\(^{10}\) In practice, Rigi has no problem handling such file sizes because the information can be read in and processed on a line-by-line basis. In contrast, a fine-grained model that stores information that corresponds to an AST can be expected to require one or two orders of magnitude more storage space compared to a middle-level model.\(^{11}\) Furthermore, modeling of fine-grained information in RSF is awkward because it supports only limited typing and binary relations. Also, in RSF all information is kept in a single data file and there is no support for incremental processing or lazy resolution of references. As a result, these limitations make it impractical to use RSF for fine-grained information. In fact, we are not aware of a tool that uses RSF for this purpose.

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 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 (cf. Section 5).

6.4. 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” [55]. 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 2.3.1). 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 semi-automatically create a decomposition using Rigi, but only seconds to produce one automatically using a dedicated script. In any case, either method is faster and use the analyst’s time and effort more effectively than with a manual process of reading program listings” [28]. Such a programmatic approach also makes it possible to quickly re-produce steps in the workflow if the underlying subject system changes.

The scripting capabilities have allowed other researchers to quickly customize Rigi to implement their own (prototype) tools. Over the years many researchers have scripted little and more involved extensions as part of their research. In fact, we believe that end-user programmability using the RCL script library is one of the main reason for Rigi’s success and widespread adoption.

Perhaps the most extensive customization of Rigi was performed at the University of Stuttgart to build the first prototype of Bauhaus. The Bauhaus tool provides interactive (architectural) clusterings of software systems written in C [1]. 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” [1, p. 318].

Since then, the Bauhaus tool has been commercialized and Rigiedit has been replaced with a custom-coded visualizer [96].

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).\(^{12}\) For instance, the Shimba environment, which analyses 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) [73]. Depending on user actions the tools pass control between them. Dali also used scripting to enable interoperability of Rigi with other tools [42]. However, when different tools are involved significant effort is typically required to synchronize native models of the different tools.

---

\(^{10}\) Typically, a large part of the file size is caused by long names that are the result of name qualification. Because these names have many similar substrings, they can be compressed very effectively. The RSF file of the Azureus system is only 0.4 MBytes after gzip compression, and the structured RSF is 4.8 MBytes.

\(^{11}\) For example, the encoding of ASTs in the Refine tool is about 20–100 times larger than the size of the program text [95].

\(^{12}\) However, it is impossible to run Rigi as an engine without its GUI (i.e., so-called headless execution).
To summarize, scripting 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 strong typing and inferior maintainability.

6.5. Levels of indirection

The Rigi system provides a number of mechanisms that add flexibility and provide abstractions over certain domain concepts. We refer to such mechanisms as levels of indirection. Table 2 gives an overview of the major levels of indirection in the Rigi system and how they support tool interoperability and customizability as well as decouplings at the architectural level.

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 us to encode an open-ended number of concrete graphs.

The data model of the exchange format provides another level of indirection. It enables domain-retargetability [55] 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 SCED (cf. Section 6.4). Both tools have their own GUIs, but they send messages to each other to synchronize operations, thus 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 (Oleol/Tk) and barcharts (The BLT Toolkit) into Rigi [55, p. 61]. Since both tools are based on Tcl/Tk, they can 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 6.4) as well as personalization of Rigi’s GUI (cf. Section 2.3.1). 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. Conversely, the TK graphical library could be replaced with different GUI toolkits (such as the Gnome binding for GTK+ and Gnome) without having to change Rigi’s C/C++ code.

Each level of indirection enables certain degrees of freedom, but there are also constraints that limit them. These constraints are the results of design decisions – made consciously or incidentally – that affect properties for users (e.g., expressiveness, performance, usability, and learning curve) as well as developers (e.g., ease of implementation, maintenance, and extensibility). For example, RSF constrains how data is represented with its graph model, which is restricted to binary relationships. As a consequence, data with n-ary relations need be translated to binary ones. To represent hierarchical graphs, Rigiedit implements hard-coded semantics for interacting with Collapse nodes and level arcs. It would be interesting to provide a customizable specification of node and arc type semantics, behavior, and rendering. RCL enables us to customize the user interface, but this flexibility does not extend to the graph canvas. This means that, except for colors, the drawing of nodes and arcs (e.g., shape and line width) is fixed. As a consequence, Rigi is not well suited to experiment with visualization techniques (e.g., renderings that assign meaning to node shapes).

7. Conclusions

This paper has discussed the Rigi reverse engineering environment and its research contributions. Rigi allows the visualization of software in the form of graphs and supports a reverse engineering methodology called structural redocumentation.

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 us to define different data models, and the editor’s scripting layer provides end-user programmability. Rigi impacted research in reverse engineering and program comprehension significantly. 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.

Acknowledgements

We gratefully acknowledge the detailed and thoughtful comments of the anonymous reviewers as well as the editors of this special issue, which allowed us to improve the paper significantly.

This Rigi body of work grew out of collaborations with many colleagues in academia, industry and government. Rigi was a cornerstone of several NSERC Collaborative Research and Development (CRD) projects with IBM Toronto Center for Advanced Studies (CAS) in Toronto. We are deeply indebted to many people – friends, colleagues, and students – who, over the years, contributed significantly to the development of the Rigi environment and its affiliated tools and the dissemination of the Rigi research results. In particular, we would like to thank Kenny Wong, Peggy Storey, Scott Tilley, Johannes Martin, Brian Corrie, Jim McDaniel, Mike Whitney, Karl Klashinsky, Jochen Stier, Jim Uhl, Mehmet Orgun, Eva van Emden, Anke Weber, John Mylopoulos, Kosmas Kontogiannis, Jim Cordy, Ric Holt, Dennis Smith, Elliot Chikofsky, Anatol Kark, Jacob Slonim, Gabby Silbermann, Kelly Lyons, Marin Litou, Eric Buss, John Henshaw, Pat Finnigan, Stephen Perelgut, John Botsford, Arthur Ryman, Martin Stanley, Jenana Campara, Nicole Mansurov, Mike Godfrey, Rudolf Keller, Tim Lethbridge, Janice Singer, Andreas Winter, Rick Kazman, Rainer Koschke, Claudio Riva, Tarja Systä, Jens Weber-Jahnke, Arie van Deursen, Leon Moonen, Jean-Marie Favre, and Michele Lanza.

The work on this paper was funded in part by the National Sciences and Engineering Research Council (NSERC) of Canada (CRDPJ 320529-04 and CRDPJ 356154-07) as well as IBM Corporation and CA Inc. via the CSER Consortium.

Appendix. Supplementary data

Supplementary data associated with this article can be found, in the online version, at doi:10.1016/j.scico.2009.10.007.

References


[70] J.Wu, M.D.Storey, Amulti-perspectivesoftwarevisualizationenvironment,in:ConferenceoftheCentreforAdvancedStudiesonCollaborative
[73] J.Favre,Cacophonymetamodel-drivensoftwarearchitecturerestoration,in:11thIEEEWorkingConferenceonReverseEngineering,WCRE'04,
[75] B.Kullbach,A.Winter,Queryingasanenablingsubtechnologyinsoftwarereengineering,in:3rdIEEEEuropeanConferenceonSoftware
[79] M.Storey,Theories,methodsinprogramcomprehension:Past,presentandfuture,in:13thIEEEInternationalWorkshoponProgramComprehension,
[99] B.Kullbach,A.Winter,Queryingasanenablingsubtechnologyinsoftwarereengineering,in:3rdIEEEEuropeanConferenceonSoftware
[103] M.Storey,Theories,methodsinprogramcomprehension:Past,presentandfuture,in:13thIEEEInternationalWorkshoponProgramComprehension,