Visualization for Software Risk Assessments

Jordi Vidal Rodríguez
jordi@software-improvers.com

Tobias Kuipers
tobias.kuipers@software-improvers.com

Software Improvement Group
www.software-improvers.com

1. Introduction

The Software Improvement Group performs so-called Software Risk Assessments (SRAs) [10]. An SRA is performed to identify the risks inherent in a software system. The types of risks that are identified during an SRA can be varied, depending on the system and the requirements of the customer.

Risks can be identified with respect to maintainability, performance, operational costs, and so on. The systems the SRAs are performed onvary widely in size, technology and complexity. They can be web applications consisting of 20 forms, or multimillion lines-of-code Cobol legacy systems.

An SRA consists roughly of two parts: A first part where the stakeholders in the system are interviewed and documentation about the system is analysed. In the second part of the assessment the source code of the system is analysed using various tools that we have developed at the Software Improvement Group.

The Software Analysis Toolkit (SAT) that has been developed at the Software Improvement Group routinely calculates a number of metrics for the system under assessment. Furthermore it calculates a graph structure that contains all the dependencies in the system. The various dependencies in the system are typed and can be data dependencies from a specific module to a specific view on a database, which in turn is dependent on a specific table in a database, which in turn triggers a stored procedure in the database (and so on). Effectively using the information in this graph depends largely on the ability to interactively view (parts of) the graph, and relating those parts to specific locations in the source code, and to metrics about those parts of the source code.

An example graph of all dependencies in a 150,000 line web application is given in figure 1.

1.1. Scenarios

Visualization needs during an assessment are twofold: first of all, visualization should facilitate the understanding of the system. This initially requires visualizing the overall structure of the system: control dependencies between modules, and data dependencies between modules and databases. Afterwards, when a general understanding of the system is reached, the visualization should provide detailed views to validate ideas that have occurred about the functioning of the system.

Interactively manipulating the view on the software system is key in both phases. As an example, consider the following scenarios.

Scenario 1

A system consisting of 3,000,000 lines of Cobol is analysed using the SAT. This results in a graph containing all the dependencies. (More about the data representation that results from the SAT analysis in section 3). The first visualization shows that there are 20 database tables in the system, and that only about 10 modules (out of 1200) access these tables. Closer inspection of the 10 modules shows that these are so-called utility modules, and that all modules that call these utilities can be considered to perform database access. The view should than be adjusted to remove the database utilities, and replace them with direct edges from the modules that called the utilities to the database tables.

Since these systems typically use different technologies to access persistent data, the view needs to be adjusted to accommodate for that fact: for gaining a general understanding of the system database access through DB2 should be visualized in the same way as, say, access through IMS. However, when looking at detailed IMS usage DB2 should obviously be removed from the view.

Scenario 2

As an example of a more detailed view consider the system displayed in figure 1. This is a web application built using Microsoft Active Server Pages. After a first inspection, and interviews with the developers of the system the consultant performing the assessment comes up with the following

hypothesis: All asp files in the system include a standard "library" file, and a single file that contains the non-dynamic portion of the page. In order to validate this hypothesis she looks at the graph in figure 1. Obviously there is nothing to see there, since there are far too many edges and nodes.

The graph needs to be interactively filtered to first show only the asp files, and their includes. If there are too many to immediately see whether the above hypothesis holds, than a threshold needs to be set to show only asp files with less than two includes: if they exist then the hypothesis does not hold.

1.2. Graph Visualization Requirements

The assessments described in the scenarios above could be performed using an interactive graph browser that supports a number of operations. The operations we currently are looking for in a graph visualization tool are: abstractions (leading to nested graphs), searching, filtering, undo facility, and automatic layout. For instance, the set of operations should enable us to replace a node by direct edges between all its sibling nodes. These operations will be applied on large graphs (up to 100,000 nodes). Finally, an scripting facility to enable automatization and an annotation facility to store comments on findings are seen as crucial for efficiently carrying out SRAs.

1.3. Position

Our position is that in spite of good tools solving partly SRA's requirements, there is still missing a general tool to handle all SRA aspects smoothly, embracing from data model, visualization to reporting.

In the next section, we give an overview of the various software visualization tools that we are currently using, or have tried in the past.

In section 3 we describe the data model that we use for our assessments, and how it relates (both in theory and in practice) to our visualization tooling. Section 4 discusses the various challenges we see when using existing visualization tooling.

We end the paper asking ourselves whether we should start to produce our own visualization software...

2. Related Tools

The most common techniques for software system comprehension are graph visualization and a combination of browsing/navigation/query. Specialized instances of these techniques are scattered among tools. The most relevant and inspiring tools for our activities are described next.

Rigi [7] uses a nested graph model. However the graph's levels are displayed using multiple windows. It has basic graph operations for name pattern search, selection and abstraction. Visualization and interaction can often be cumbersome.

SHriMP [9] is a nested graph navigation tool, lacking any graph manipulation feature. It complements Rigi. It maintains context and focus via a modified fish-eye algorithm.

Dalí [6] is a workbench that supports extraction and fusion of architectural views. It highlights the need to fuse views from different source extractions, leaving the data gathering to other tools for the purporse. It is an open approach to integrate tools and uses a common data repository.

Portable Bookshelf [5] is aimed at re-engineering and migration, mainly as a navigation tool by means of directed graphs. Software landscapes visualize the main part of the system and keep context with neighbouring subsystems.

Code Crawler [3] is a tool that combines object oriented software metrics into some predefined such graph models like tree, matrix correlations and histograms. The nodes (or entities) can distinguish up to 3 data dimensions, visualized as x, y size and colour.

CIAO [2] is a flexible navigator that can visualize graph models and query the system at source code level. It can be used in any project by specifying a new data model.

SPOOL [8] is a tool set to bridge to other comprehension tools. It allows browsing of high level constructs, query the design and structural searching, but lacks abstraction operations. Other helper tools are used for source code analysis. It aims to integrate several tools to allow flexibility in creating user-defined views of any system.

As the above descriptions reveal, these tools do not satisfy all requirements as stated in section 1.2. Consequently we looked at graph layout libraries such as Dot, aiSee, GVF, JViews and Tom Sawyer, which offer advanced features. Dot is not an interactive tool, although there are some libraries containing dot that alleviate this problem. aiSee was found to offer good layout algorithms but with an unfriendly user interface. The last two are powerful graph libraries allowing nested, multiple layouts.

3. Model Requirements

The tools above demonstrate the usefulness of the various features they were developed for, but fail to satisfy the complete list of features we need to perform Software Risk Assessments.

During assessments we need to contrast different views at different abstraction level of the system, subsystem or a slice of source code. Due to the variety of systems that can be analysed, no specialized tool fully suits the purposes. Instead an open, extensible tool should be created to cope with the business demands.

In general data is gathered during the assessment process into a generic data model (in similar fashion as FAMIX [4]). From it we generate three generic types of views: Directed graphs, charts and source code. The visualization tool should be able to display these three views, their relations, and navigate effortlessly between them.

The model we generate views from are described below.

3.1. Data Model

The are two main aspects the data model must meet. Fisrt, it have to allow navigation from one view to another, this is, regardless of the inspection starting point we can navigate to any related view forth and back even at different abstraction level.

Current tools attempt to provide such functionality (i.e. SHriMP, SPOOL) but limited to a pair of views with their own data model. On the other side, (commercial) source navigators (such as Eclipse [1]) provide excellent features but miss high level views.

Second, it holds data from three subareas: the artifacts' relations, visualized as a directed graph; the software metrics, visualized in charts; and the source code, visualized as enhanced text.

The model must be able to support any programming language, as in the case of large legacy systems. Thus object–oriented, procedural, functional and scripting languages must all be supported transparently. Only the artifact gathering tool is tailored to the target language.

Next, we describe what features both the data model and each view should provide support.

3.2. Graphs

Most of our interest is on exploring large and highly connected hierarchical typed directed graphs and derive some knowledge. We have seen there is no single tool that supports all required features for effective exploration: nested graphs (SHriMP), abstraction operations (Rigi), incremental layout, context keeping (SHriMP, PBS) and annotations. Instead each tool supports one or few of these.

3.3. Source Code

Source code inspection is a common practise in software assessment. Either beginning from source code or from a model, we are interested in obtaining alternative views of the same set of artifacts.

When inspecting a piece of source code, the corresponding subgraph and a set of related metrics should be displayed. Similarly, when pointing to either a graph or a chart entity, the related piece of source code would show marked up.

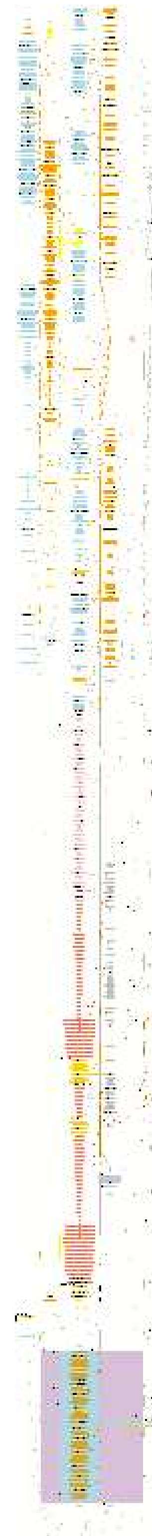

Figure 1. A graph containing all the dependencies of a system under assessment

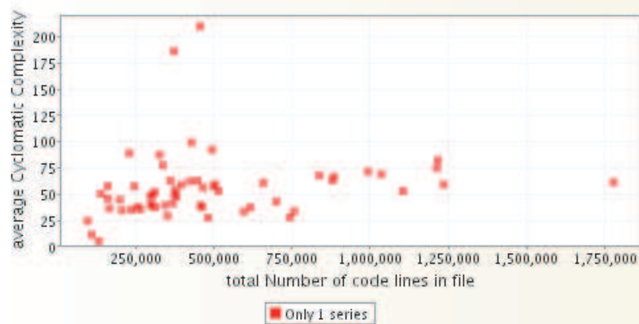

Figure 2. Complexity of systems versus their size

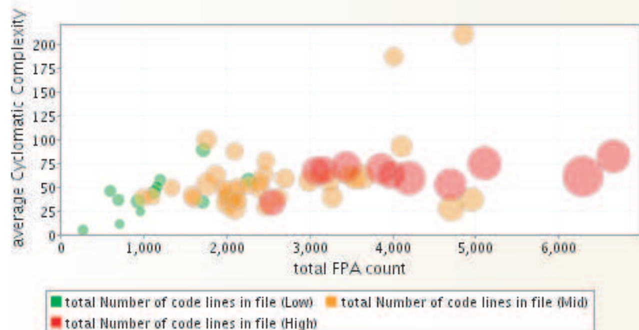

Figure 3. Complexity of systems versus function points and lines of code

3.4. Metrics

Software metrics gathering, at different granularity levels, is routine in our assessments. Currently we visualize them as independent charts (i.e. bar charts, pie charts, bubble charts).

Metrics are valuable to quickly point out measurable features such as complexity and size among other source code attributes.

Linking the metrics with the other views also aids in inspecting. For instance, selecting a complex system's program and then browsing its source code. We also realize it is helpful to merge metric information into the graph model as in [3].

4. Visualization Challenges

To carry out SRA we actually use practises from reverse engineering, software metrics measurements and automatic generation of documentation. Currently we are using a set of independent tools for the purpose. These tools are integrated by our Software Analysis Toolkit. The visualization part currently consists of graphs visualized using dot, charts displayed using JFreeChart, and marked up source code using a purpose built system. We browse the source code and make annotations manually.

We have conducted experiments with some tools mentioned in related work section. Other tools were dismissed after checking the list of features or after seeing the demos. The tools we have experimented with did not serve our goals. Most tools perform a single task well, but not others. Furthermore, we had problems integrating these tools within the Software Analysis Toolkit, and the usability for some of these tools is so terrible that we wonder whether they are used at all.

Our wished features for graph visualization tools are as follows:

4.1. Focus and Context

Usually we focus on a small part of the large system. When zooming in, the involved nodes should be placed close to each other to fit one screen while maintaining its context (i.e. its immediate neighbours). The context may be essential to identify possible erroneous relations. Solutions like nested graphs, multiple views, fish-eye views and showing neighbours seem feasible.

4.2. Annotations

The assessment process produces large amounts of results referencing both the detailed and the coarse level. Crucial is the reproducibility of the assessments, for updated versions of the system, and retrieval of old assessment results for comparison. This opens the way to trend analysis.

Even rudimentarily supported, by saving views and other data files, an integrated annotation tool would increase productivity. The Film strip feature in SHriMP, or the Saving View in Rigi are both promising methods. Structured storage would help in the report writing phase.

4.3. Layout

The graph layout reveals important relations that can be spotted by a quick visual inspection. However, obtaining a good layout is not trivial, not to mention that for large graph no layout has given satisfactory results.

Therefore, using the focus+context to reduce the graph size to display, layout algorithms can be used again. Nevertheless, it is also of importance to keep the mental map as the exploration proceeds. For instance, smooth transistions and minimal alterations to the graph structure should be enough. Animation cues are not discarded at all.

A not less important aspect is labeling. Labels should be readable at any zoom factor. Although if the node is too small, there is no need to show its label. SHriMP [9] approach seems the most advanced approach so far.

4.4. Graph Operations

During the understanding process we manipulate the graph by, for instance, grouping (abstraction) common artifacts into subnodes (hence nested graphs). Other operations are: navigation, search, filtering and selection. We are currently not convinced that this list is exhaustive, but more experiments are needed.

Navigation should allow to track the visited elements. Different approaches could be: a) list visted nodes in a separate view; b) move visited nodes close each other; c) not alter the layout. Options a), b) maintain context, while c) is adequate when only the final target is necessary.

The search space can be textual or structural. Locating certain names of artifacts is textual. Locating chains of node types and edge types is structural. Questions like "Is database X accessed directly or indirectly by any program in Y?" should be answered by a structural search. This search involves textual and structural search combined.

Abstraction as supported by Rigi merged with the SHriMP capabilities of nested graphs would be a nice start.

We have pointed out the importance of keeping focus and context while carrying operations that affect the graph structure. When having to understand small portions of large graphs, these operation features help raise model comprehension by reducing confusion by sudden changes of the layout.

5. Conclusions

A number of tools exist that partly solve the complex reverse engineering task of Software Risk Assessment.

Projects like SPOOL try to go further, providing environments were multiple tools collaborate to tackle as much of the reverse engineering tasks as possible in an elegant, clear way.

A standarized, integrated environment would allow users to access simultaneously a broad set of tools with the advantage that they could tightly collaborate each other, resulting in a productivity boost. The diverse techniques could be tried under a single environment allowing to more efficiently compare, investigate, create new research techniques benefiting from the already developed helper tools. For instance, to try a new visualization technique, only the view has to be programmed.

We have been developing a generic data repository to hold all data we derive from large software systems to perform assessments. We currently can visualize aspects of this data in various ways.

Our search for an interactive graph viewer that suits our needs has so far been interesting, but has not led to the tool we want. We have listed our wishes regarding such a tool.

Nevertheless, we are currently contemplating building the tool ourselves. We would like to invite the software visualization community to challenge our ideas, to tell us such a tool already exists, to tell us everything we know is wrong, and finally, to collaborate with us to build a system that would perfectly suit our needs.

References

[1] *The Eclipse project*. http://www.eclipse.org.

[2] Y.-F. R. Chen, G. S. Fowler, E. Koutsofi os, and R. S. Wallach. Ciao: A graphical navigator for software and document repositories. In *Proc. Int. Conf. Software Maintenance, ICSM*, pages 66–75. IEEE Computer Society, 1995.

[3] S. Demeyer, S. Ducasse, and M. Lanza. A Hybrid Reverse Engineering Approach Combining Metrics and Program Visualization. In *Proceedings of the Working Conference on Reverse Engineering*, pages 175 – 186, 1999.

[4] S. Demeyer, S. Ducasse, and S. Tichelaar. Why unifi ed is not universal. UML shortcomings for coping with round-trip engineering. In *Proceedings UML'99 (The Second International Conference on The Unified Modeling Language)*, 1999.

[5] P. Finnigan, R. Holt, I. Kalas, S. Kerr, K. Kontogiannis, H. Mueller, J. Mylopoulos, S. Perelgut, M. Stanley, , and K. Wong. The Software Bookshelf. In *IBM Systems Journal*, volume 36, pages 564–593, 1997.

[6] R. Kazman and S. J. Carriere. View Extraction and View Fusion in Architectural Understanding. In *Proceedings of the 5th International Conference on Software Reuse*, Victoria, B.C., 1998.

[7] H. Müller. *Rigi*. http://rigi.cs.uvic.ca.

[8] S. Robitaille, R. Schauer, and R. K. Keller. Bridging Program Comprehension Tools by Design Navigation. In *Proceedings of the International Conference on Software Maintenance*, pages 22–31, October 2000.

[9] M.-A. Storey. *SHriMP*. http://shrimp.cs.uvic.ca.

[10] A. van Deursen and T. Kuipers. Source-Based Software Risk Assessment. In *Proceedings of the International Conference on Software Maintenance*, 2003.