CCVisu: Automatic Visual Software Decomposition *

Dirk Beyer
School of Computing Science
Simon Fraser University
B.C., Canada

Download: `http://ccvisu.googlecode.com`
Project: `http://www.cs.sfu.ca/~dbeyer/CCVisu`

ABSTRACT

Understanding the structure of large existing (and evolving) software systems is a major challenge for software engineers. In reverse engineering, we aim to compute, for a given software system, a decomposition of the system into its subsystems. CCVISU is a lightweight tool that takes as input a software graph model and computes a visual representation of the system's structure, i.e., it structures the system into separated groups of artifacts that are strongly related, and places them in a 2- or 3-dimensional space. Besides the decomposition into subsystems, it reveals the relatedness between the subsystems via interpretable distances. The tool reads a software graph from a simple text file in RSF format, e.g., call, inheritance, containment, or co-change graphs. The resulting system structure is currently either directly presented on the screen, or written to an output file in SVG, VRML, or plain text format. The tool is designed as a reusable software component, easy to use, and easy to integrate into other tools; it is based on efficient algorithms and supports several formats for data interchange.

Categories and Subject Descriptors: D.2.7 [Distribution, Maintenance, and Enhancement]: Restructuring, reverse engineering, and reengineering; D.2.11 [Software Architectures]: Data abstraction; D.2.13 [Reusable Software]: Reusable libraries; D.2.2 [Design Tools and Techniques]: Modules and interfaces; E.1.c [Data Structures]: Graphs and networks; G.2.2 [Graph Theory]: Graph algorithms;

General Terms: Algorithms, Design, Documentation

Keywords: Reverse engineering, Software decomposition, Clustering, Software visualization, Software quality assurance, Force-directed graph layout

1. OVERVIEW

Many maintenance tasks (corrective, adaptive, perfective maintenance) require the software engineer to understand the structure of the system. A top-level structure is necessary to locate the module that needs to be changed. Once the module is located, a more fine-grained structure is necessary to identify the relations of that module to the rest of its subsystem. Ideally, the structure used for maintenance purposes is such that everything that the programmer has to know, in order to perform the change request, can be located inside one subsystem or component. In a graph model for the software system, where nodes represent software artifacts and edges represent dependencies between the artifacts, the subsystems that we are looking for correspond to subgraphs that are strongly intra-connected (high cohesion) and loosely inter-connected (low coupling). CCVISU is a tool for such a decomposition of large software systems. From the graph model of the software system, it computes a layout that has the following property: artifacts that are connected by many dependencies are placed at close positions, and unrelated artifacts have distant positions. In other words, cohesive parts of the graph are grouped together and separated from other cohesive parts. This property of the layout holds on the top level (grouping of and relations between subsystems) as well as on the lower levels (relatedness of artifacts within the groups). In contrast to other techniques to visualize graphs (e.g., multidimensional scaling) CCVISU's layout algorithm is not fed with a similarity or distance matrix as input, but *discovers* the information of relatedness from the graph and presents these facts as distances in the computed layout. It not only visualizes information that we already know, but also *reveals* new facts about the system. A comprehensive discussion of related work is given in a technical report [2].

Visual Clustering. The tool implementation is based on force-directed graph layout, which consists of two components: an energy model that maps a layout to an energy for evaluation —the smaller the number, the better the layout—, and an algorithm that computes a layout with minimal energy. A *layout* of a graph G is a function p that maps each node of the graph to a position in the 2- or 3-dimensional real space. An *energy model* is an evaluation function U that assigns to each layout p a real number. The layout p is the *best layout* for G if $U(p)$ is the global minimum of function U. The energy model encodes the layout goal, i.e., the user's choices of what is considered as good layout. For visual clustering, this means to produce layouts that provide separation of cohesive subgraphs and interpretable distances. CCVISU uses the clustering energy model that was introduced and first used in our initial study on co-change clustering [1]. An *energy minimizer* is an optimization algorithm that searches for a good approximation of the best layout. The energy minimizer starts with an initial layout, where the positions of the nodes are randomly assigned. Then, in every iteration, the algorithm tries to improve the layout according to the energy model (by using the first derivation of the energy function to compute a direction and a distance for the new placement of each node). This framework allows for a flexible tool implementation: the same efficient algorithm for the minimization of the energy can be used for different energy models.

Contribution of CCVisu. We provide software engineers and software-engineering tool designers with a lightweight component for the task of computing clustering layouts for software graphs. Some characteristics of CCVISU are: (1) The input graph is given

*This research was supported in part by the NSERC grant RGPIN 341819-07.

ICSE'08, May 10–18, 2008, Leipzig, Germany.
ACM 978-1-60558-079-1/08/05.

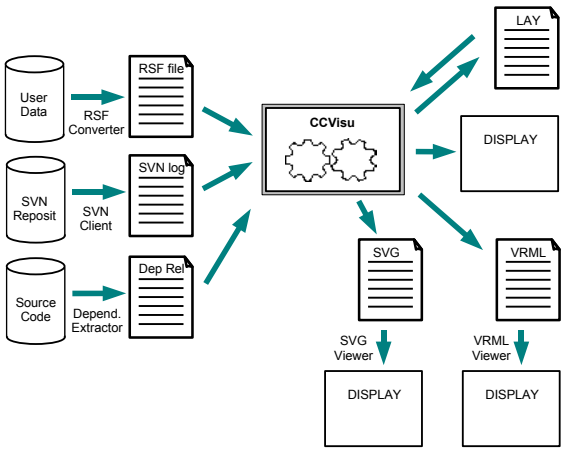

Figure 1: CCVisu's input/output flow

in the text-based format RSF (Rigi Standard Format), which simplifies data exchange between software tools for different purposes in the reengineering process. (2) The tool is lightweight and implemented in Java. It has a clean and slim interface to facilitate integration as reusable component or as external tool into other reengineering toolkits. (3) The tool is applicable to all kinds of software graphs, e.g., call graphs, containment graphs, inheritance graphs, co-change graphs. (4) The tool is easy to use, and easy to extend because it is designed as an open framework. (5) The layout algorithm is highly efficient, based on the well-known *Barnes-Hut* algorithm. (6) Besides the energy model for visual clustering, the tool provides several other energy models for different layout goals, and CCVISU lets the user optionally choose other energy models via command-line options. Thus, the component can be used for general force-directed graph layout.

Availability. CCVISU is the first freely available open-source tool for force-directed graph layout which implements several energy models for different layout goals, in particular for visual clustering. It is released under the GNU LGPL, and we hope that other researchers and engineers can ease their work by using CCVISU, and that developers find it easy to integrate the component into their applications. Example layouts (together with their input graphs) are available on the project web page.

Input and Output Formats. Figure 1 shows the more general usage of the tool. The input is a textual representation of a graph in Rigi Standard Format (RSF), a standard format for relations in the reengineering community. The input graph needs to be undirected, irreflexive, and connected. As example of an application-specific input reader, the tool can read a CVS or SVN log file to transform it into a co-change graph. To display a previously computed layout, the input can also be a text file containing a layout (LAY).

The layout of the artifacts can be produced in four forms. First, the text file (LAY) can later be read by CCVISU or other tools, such that the tool can be embedded in different environments over a simple text-based interface. Second, the VRML format allows the use of an external VRML viewer (or a web browser with VRML plug-in) to view the layout (2D as well as 3D). Third, the SVG format allows the use of external SVG viewers (or a web browser with SVG plug-in) to view the layout. SVG viewing is more efficient than VRML viewing and can be used to display much larger layouts (thousands of nodes), but does not support 3D effects. Fourth, the layout can be directly displayed on the screen. This form is the preferred output method for huge graphs, when VRML and even SVG viewers are not able to display the layout.

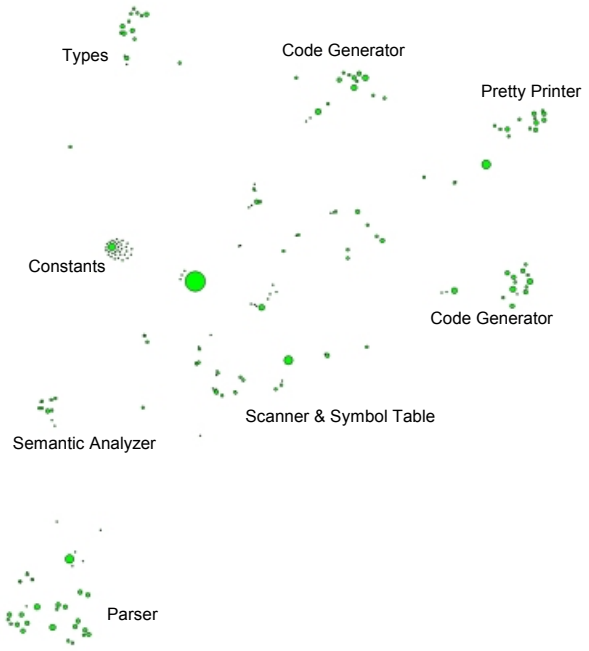

Figure 2: Types and functions of a compiler system

2. APPLICATION EXAMPLES

Call Graphs. If the input of CCVISU is a call graph of a software system, the generated layout places the software artifacts (methods, classes, packages) close together if they are coupled by many calls, and separate, if they rarely interact. Cohesive subsystems appear as groups and are separated from other cohesive subsystems.

Usage Graphs. Figure 2 shows the resulting layout of applying CCVISU to a usage graph for a compiler system. The nodes of the input graph (extracted by Bauhaus) are types and functions, and the edges represent the usage dependency. The software graph consists of 205 nodes and 767 edges. The visual decomposition reveals the different components of the compiler: scanner, parser, types, constants, semantic analyzer, pretty printer, and code generator.

Co-Change Graphs. Changes of software systems are less expensive and less error-prone if they affect only one subsystem. Thus, groups of files that frequently change together are subsystem candidates [1]. A co-change graph, which represents common changes in the history of the system, can be easily extracted from the system's version repository.

Non-Software Graphs. CCVISU was applied by several people to graphs that are different from software graphs, because the clustering principles behind CCVISU work for many dense graphs, for example, social networks, co-author relationship graphs, web-based document-access graphs, document-relationship graphs extracted from Google Desktop Search.

Acknowledgements. We thank Damien Zufferey, who helped implementing several feature extensions of the tool as undergraduate research assistant, and Andreas Noack, for the valuable discussions during our work on co-change visualization [1].

3. REFERENCES

[1] D. Beyer and A. Noack. Clustering software artifacts based on frequent common changes. In *Proc. IWPC*, pages 259–268. IEEE, 2005.

[2] D. Beyer and A. Noack. Mining co-change clusters from version repositories. Technical Report IC/2005/003, EPFL Lausanne, 2005.