Co-Change Visualization Applied to PostgreSQL and ArgoUML*

(MSR Challenge Report)

Dirk Beyer
EPFL, Switzerland

ABSTRACT

Co-change visualization is a method to recover the subsystem structure of a software system from the version history, based on common changes and visual clustering. This paper presents the results of applying the tool CCVisu, which implements co-change visualization, to the two open-source software systems PostgreSQL and ArgoUML. The input of the method is the co-change graph, which can be easily extracted by CCVisu from a Cvs version repository. The output is a graph layout that places software artifacts that were often commonly changed at close positions, and artifacts that were rarely co-changed at distant positions. This property of the layout is due to the clustering property of the underlying energy model, which evaluates the quality of a produced layout. The layout can be displayed on the screen, or saved to a file in SVG or VRML format.

Categories and Subject Descriptors: D.2.7 [Software Engineering]: Distribution, Maintenance, and Enhancement — *Restructuring, reverse engineering, and reengineering*

General Terms: Design

Keywords: Software visualization, software clustering, software structure analysis, force-directed graph layout

1. METHOD

In reverse engineering and reengineering, we often want to extract a description of the system structure from available resources. Even if a structure description (often 'as-designed') is available, it can be useful to complement it by an extracted description of the 'as-build' structure. *Co-change visualization* is a method that extracts such a description, and aims to help in reverse engineering and re-engineering activities like understanding the structure of the system, change impact and change propagation analysis, coupling analysis, architecture and design quality analysis.

Approach and tool used. The approach of co-change visualization is introduced in [2], and implemented in the tool CCVisu [1]. It requires as input the version history, which is almost always available, and automatically produces a visualization that groups together components that were often changed together, and separates independent components.

Input data. For the two example systems, we take as input the version log information, as (in case of Cvs) obtained by applying the command `cvs log -Nb` (only default branch, ignore tags). We consider the whole development period from project start to Feb. 8, 2006 (extraction date). From this input, we extract the *co-change graph on file level*. The nodes in the (bipartite, undirected) co-change graph are files and commits. An edge between a file node f and a commit node c exists if file f was changed by commit c. The table below presents the characteristics of the graphs. For the details of the method and **related work** we refer to [2, 1].

System	Files	Commits	Changes	Log file
PostgreSQL	4,125	20,500	88,468	17 MB
ArgoUML	10,142	10,137	57,091	16 MB

2. RESULTS AND EVALUATION

The commit nodes and the edges are omitted in the visualizations for readability. The file nodes were drawn in different colors, to compare the grouping suggested by the layout with an authoritative decomposition, according to documentation. The area of a circle is proportional to the number of changes of the file. Each layout was computed within 5 min on a 1.7 GHz Pentium M laptop, using only 200 iterations of the minimizer. The layouts in SVG or VRML format provide (interactively) the file names as annotation and basic zoom features. The figures in this paper are annotated with the names of the subsystems (gray boxes). The layouts in SVG and VRML format, the Cvs log files, and the co-change graphs in RSF, are available on the supplementary web page at http://mtc.epfl.ch/~beyer/ccvisu_msr.

PostgreSQL. In the authoritative decomposition we considered 12 subsystems of PostgreSQL, and assigned a color to each subsystem: executor (red), optimizer (blue), parser (cyan), storage (magenta), catalog/commands/nodes (yellow), access (dark cyan), port (olive-green), regression test (brown), interfaces (light blue), include (light green), utilities (light gray), and documentation (green).

We can use the colors to evaluate if CCVisu has positioned the 4,125 files in groups in agreement with the authoritative decomposition. Figure 1 clearly separates the main clusters of the documentation (top right, largest circle at bottom is TODO file), the interfaces for libpg (center right) and ecpg (bottom right), and the regression test files (top left) from the backend files (center left), and from the include and utilities (center). To get more insights into the backend files on the left (here not separated), we need to 'zoom' into this part by restricting the co-change graph to the backend, and computing a new layout for this subgraph.

Figure 2 visualizes the backend only. The subsystems that formed the large group on the left in Fig. 1 are now better

*This research was supported in part by the MICS NCCR of the SNSF.

MSR'06, May 22–23, 2006, Shanghai, China.
ACM 1-59593-085-X/06/0005.

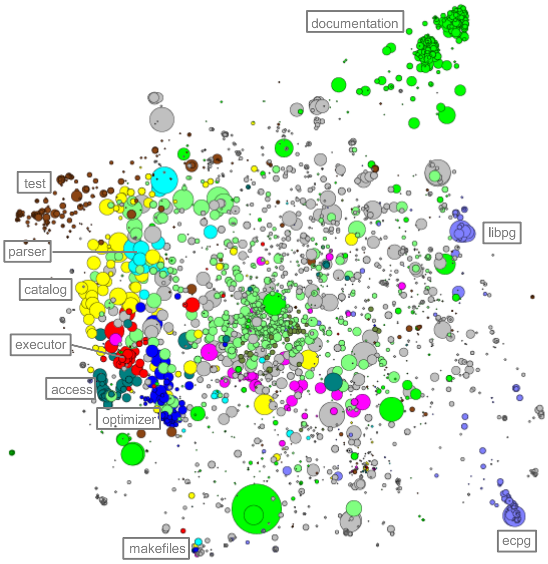

Figure 1: PostgreSQL

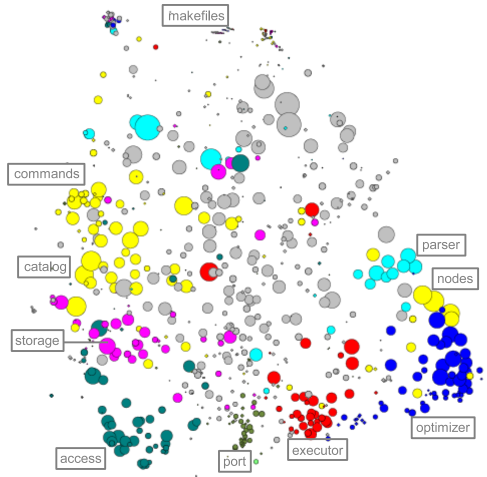

Figure 2: PostgreSQL — backend only

separated. The layout separates from the rest the executor, port, and access subsystems (bottom). It puts the optimizer, nodes, and parser into one group (right), but does not merge the three groups, which makes sense according to the authoritative decomposition. The commands, catalog, and storage files (left) are not separated but also not merged. The gray nodes blur the otherwise clear picture: they represent the utility files, which are used by all subsystems, and therefore they are placed correctly by the method. The three groups containing files in every color at the top are three collections of makefiles — since they necessarily change together, they are placed together although they are assigned to different subsystems in the authoritative decomposition.

ArgoUML. The authoritative decomposition divides the files into 9 parts: old development files (uci in green, uci-gef in brown), documentation (yellow), test files (blue), cognitive (cyan), diagrams (magenta), UI (dark cyan), UML-UI (red), and model (light blue). The files that could not be assigned to any subsystem are drawn in light gray (consisting of utilities, makefiles, configuration files, etc.).

The placement of the 10,142 files of ArgoUML is shown in Fig. 3. The old development branch is clearly separated (top right). Also, the www documentation files are shown

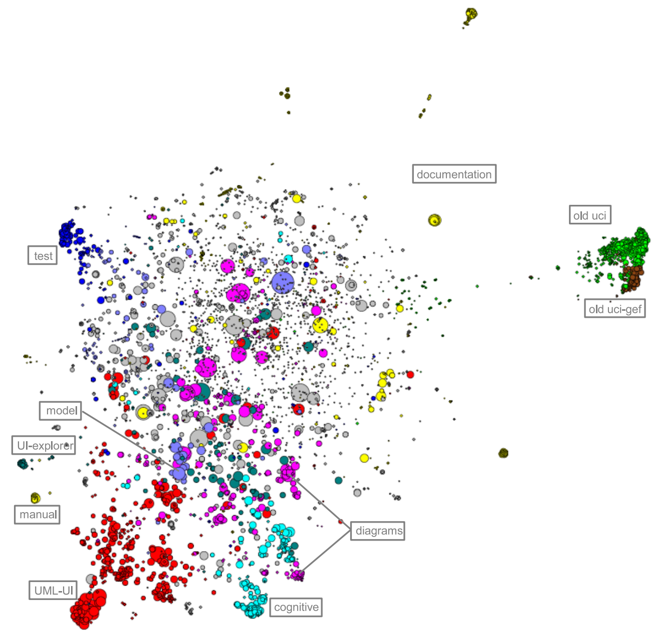

Figure 3: ArgoUML

as several clusters (right side), and some implementation-dependent parts of the documentation (e.g., cookbook, config) are placed close to the corresponding source files. Furthermore, the test files (top left) are nicely separated from the rest. The files for the UML-UI (almost completely) form the red clusters at the bottom, and also the files of the 'cognitive' subsystem are separated. The cluster with the most files of the UI subsystem is the explorer, which is separated from the rest on the very left (close to the manual cluster).

The diagrams and model files are spread over the picture. A restriction of the visualization to the source files, as done for PostgreSQL, leads to a picture (not shown here) where the diagrams subsystem is separated, but the model subsystem does still not form a cluster because this subsystem is responsible for interfacing and exchanging data. For example, the largest circle (light blue) is the class ModelFacade. The visualization allows the following interpretation: the subsystems for UML-UI, cognitive, diagrams, and the explorer component of the UI subsystem are reasonably loosely coupled, and the rest is dependent on many other components (expected to be necessary for UI, models, parts of diagrams). We omit a more detailed discussion for space.

Conclusion. The resulting visualization provides the software engineer with valuable information, e.g., for reverse engineering it reveals the subsystem structure, for program understanding it illustrates which artifacts depend on each other, and for quality assessment it can be used to highlight unstable parts of the system. That the co-change graph is indeed a good prediction model can be shown by comparing two layouts that result from splitting the co-change data into a (virtual) past and future. The method relates not only source code files, but also, e.g., SQL query files, test files, and documentation files, to program source code files.

3. REFERENCES

[1] D. Beyer. Co-change visualization. In *Proc. ICSM'05, Industrial and Tool volume*, pages 89–92, 2005.

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