YARN: Animating Software Evolution

Abram Hindle, Jack ZhenMing Jiang, Walid Koleilat, Michael W. Godfrey, Richard C. Holt
University of Waterloo
{ahindle,zmjiang,wkoleila,migod,holt}@cs.uwaterloo.ca

Abstract

A problem that faces the study of software evolution is how to explore the aggregated and cumulative effects of finely grained changes that occur within a software system over time. In this paper we describe an approach to modelling, extracting, and animating the architectural evolution of a software system. We have built a prototype tool called YARN *(Yet Another Reverse-engineering Narrative) that implements our approach;* YARN *mines the source code changes of the target system, and then generates* YARN *"balls" (animations) that a viewer can unravel (watch). The animation employs a static layout of the modules connected by animated edges that model the changing dependencies. Furthermore, the edges can be weighted by the number of dependencies or the importance of the chance. We demonstrate our approach using the open source database system* PostgreSQL *as the target system.*

1. Introduction

Successful software systems evolve in many ways and for many reasons: bugs are found and fixed, new features and deployment environments are requested by users, and systems are refactored by developers to improve the internal design. However, the visualization and comprehension of change over time is a problem that still faces the study of software evolution.

For a given system, a developer may have access to static "snapshots" of its software architecture and the internal dependencies. These snapshots may be hand drawn by an expert or even automatically generated from the source code. However, in practice these approaches are not well suited to the task of comprehending how the system has evolved: hand drawn snapshots are usually not maintained as the system ages, and automated architecture visualization tools tend to emphasize static views of the current system version.

Emphasizing the changes to a system's architecture requires refocusing the supporting tools toward calculating architectural deltas and then representing them effectively to the user. At the extractor level, this might include performing fact extraction incrementally; in our case, we establish a baseline architecture for the system, and the examine the CVS commits that contain the code for the changes. After analyzing the results and reconciling the changes against the baseline, the resulting architectural model of the system and its changes can be presented to the user; we have chosen to use an animated visualization to do this.

Animation is an intuitive and natural way to show change over time visually. We can start by showing the state of the architectural dependencies within the system at a chosen baseline version, and then allow the user to view the resulting changes progressively as animations to the architectural visual model. Given an interval of time we can take advantage of cumulative views and show the differences in the context of the whole system rather than just in the context of the instance. This is often a problem with existing extractors because they may provide only only a snapshot of a change and are not necessarily easy to reconcile to the state of the system.

Tools to aid in studying the evolution of software systems typically deal with very large data sets; often the data sets are so large that it is impossible or impractical to view all of the data at once. Animation enables us to traverse this rich and large information space and interpret the data in a visual way rather than a textual or statistical way.

Some tools that animate graphs do so by moving the nodes around; instead, we avoid moving nodes around and focus more on the dependencies between the nodes. We feel that by keeping the nodes in place, we allow for easier comparison and allow one to perceive change easier, although it does restrict how we can animate information. Animation exploits the temporality of the data in the repository and better illustrates the dynamic behaviour of the evolving dependencies between modules. We employ two approaches to representing the dependencies: the cumulative addition of dependency and the difference of edge weights between changes.

We have included a demonstration of *YARN* and its generated *YARN* Balls (animation) in a flipbook-like form (figures 1, 2, 4, 5, 6, 7); thus, the reader can manually animate the printed Yarn Ball like a flipbook.

1.1 Problem

Our goal is to animate architectural aspects of finely grained change in a useful and informative way. To do so, several questions must first be considered: How do we show the architectural evolution of a software system? What can we hope to gain from such a visualization? Given the large amount of data available what is a compact way to explore and exploit it? How can we show the cumulative effects of change? Can we show change effectviely without getting “lost”? Can we provide useful parameters to let the user fine tune the results? For example, we considered:

Color: In section 3.3 we discuss the various ways we can use edge color to indicate and highlight various flavours of change.

Edge Width: Edge width can be used to indicate the size of changes; unfortunately, it is hard to represent negative values with edge width. Edge widths can also be used to indicate age of changes; for example, edges can shrink as they age.

Edge Existence: Edges can be shown cumulatively or not. That is, we can show a single change itself or we can show the cumulative effect of the change. Non-cumulative edges are problematic for negative weights, however.

These parameters can also represent more complex combinations of attributes, rather than just a single attribute. We can encode a particular emphasis such as highlighting large

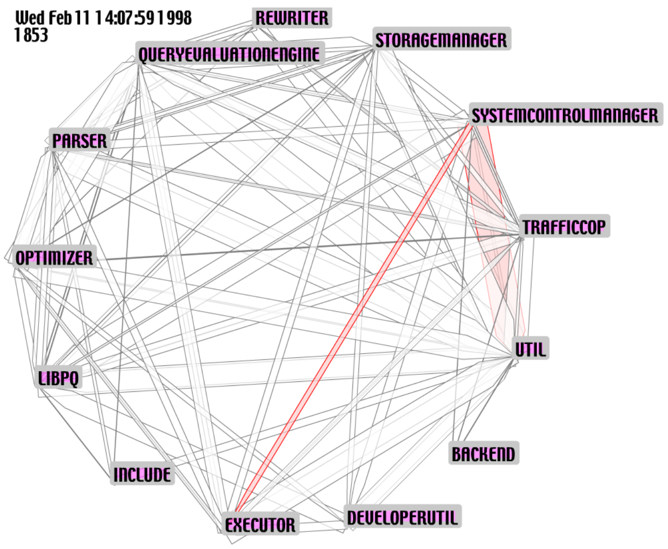

Figure 1. *PostgreSQL YARN* **Flipbook shot 1/6**

architectural changes, or a focus on a certain kind of behaviour.

2. Related Research

There is a considerable amount of related work to this problem; much of the research comes from the work done in the Mining Software Repositories [4], Software Evolution and Software Maintenance [9] research communities.

One of the earliest uses of program visualization and animation is the well known film “Sorting Out Sorting” by Baecker et al. [3], which animated how values can be sorted by various algorithms. More recently, Gall et al. used 3D graphics to compare releases of a project side by side [7]. Marcus et al. have also used 3D visualization and animation of source code [11]; their use of animation was for interactivity rather to represent time.

Our architectural views are similar to those of Rigi [12] and Shrimp [13]. In particular, we note that Shrimp makes use of animation in its visualization, although it is used to support iterative navigation rather than for representing change over time.

Finally, we note that our architectural model of *PostgreSQL* was adapted from that of Dong et al. [5].

3. Tools

We first use *CVSup* to obtain a local copy of the *PostgreSQL* CVS repository. Then *C-REX* is used to extract the architectural information from the repository. Once extraction is done, we run *HistODiff*, which makes use of *C-REX*’s output to compute the number of dependencies between subsystems, output the dependency graph, and highlight architecturally important change. Finally, all the dependency graphs are read in by *YARN* which produces the *YARN* Balls, or animations that can be played back by the user. Figure 9 illustrates the extraction flow that we have used in our analysis.

3.1 C-Rex

We use *C-REX* [10] as our fact extractor, as it has been designed specifically for conducting historical architectural analysis. It has several advantages over most architectural fact extractors. First, traditional snapshot fact extractors, such as *LDX* [14], *RIGI* [12], and *CPPX* [1], are designed to retrieve architectural information from only one version of a system. *C-REX* is an evolutionary source code extractor; it extracts information from version control systems and recovers architectural information over a period of time. Second, source code might not compile properly due to the use of programming language dialects, syntax errors, etc. In this situation, most parser-based extractors will fail, but *C-REX* avoids fully parsing the source code by making use of the *ctags* source code tokenizing tool [2]. This makes *C-REX* more robust than most extractors. Finally, most of extractors operate on the preprocessed code or the object code. Because of compilation flags, a parser-based extraction results may contain information specific to only a particular configuration; *C-REX* operates on the original source code, therefore it can extract more information relevant to software evolution than parser-based extractors.

C-REX analyzes the main branch of a system's source code repository. *C-REX* extracts all the changes from each revision and groups revisions into transactions. It outputs two types of information: a *Global Symbol Table* and a set of *Transaction Changes*.

The **Global Symbol Table** maps all of the programming language entities ever defined during the history of software development to the file locations where these entities are defined. An entity can be of any C language types, such as a macro, variable, function, struct, enum, etc.

Transaction Changes list the entity changes committed by the same author, at approximately the same commit time, with the same log message. It contains the author's name, a unique hash value to identify this transaction, the commit time, and the log message as well as detailed entity changes. An entity change can be one of three types depending on the scope of the entity: *modified* if the entity exists in both the previous and current system revision, *added* if it exists only in the current revision, or *removed* if it exists only in the previous revision.

Within each changed entity, *C-REX* keeps tracks of changes in entity types, dependencies, and comment changes. If the entity is a function, *C-REX* also tracks changes in parameters and return types.

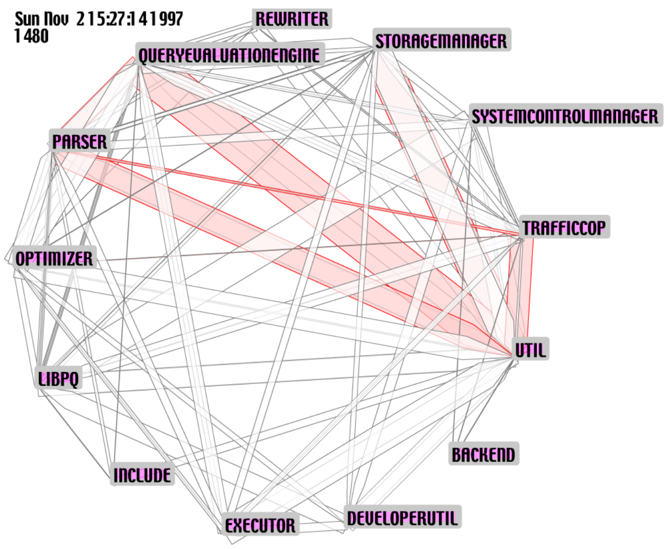

Figure 2. *PostgreSQL YARN* **Flipbook shot 2/6**

3.2 HistODiff

HistODiff performs many tasks: it associates symbols to files; it resolves references between changing architectures; it performs "lifting" of architectural information; it filters the observed changes using heuristics to identify key changes; it produces graphs for viewing; and it creates reports of changes that are deemed interesting. It makes the assumption that each file is associated with a single module within the module hierarchy.

Symbol Mapping: *HistODiff* resolves the context of multiple symbols to functions and macros. The symbols are supplied by *C-REX*'s output. The symbol mappings are important because the changes provided indicate the symbols the changes depend upon.

Architectural Mapping: *C-REX* produces a list of transactions where entities such as functions, macros, and variables are added, deleted, and modified. This list of transactions is used to update the architectural dependency graph with the change in dependencies.

Lifting: The change information is "lifted" to the architectural level, where the top-level subsystems and their dependencies are modelled. Two kinds of graphs can be produced: a dependency graph between subsystems over time, and a difference graph that shows the dependency changes betweeen subsystems before and after a given transaction. The graphs have directed weighted edges that indicate the number of calls between modules.

Filtering: In large projects such as *PostgreSQL* there will be tens of thousands of transactions, but not all are architecturally significant. Large change transactions are often noticeable because they affect many files or have a large line count, but small changes of one or two lines can also add drastically alter the architecture and change the dependencies between modules. These are important changes to make note of, because they could indicate some kind of architectural violation or important feature addition.

Our filtering heuristics highlight transactions that affect the number of dependencies between the top level subsystems and that satisfy one of the following criteria:

- The transaction adds a dependency between two subsystems that were previously unconnected.
- The transaction removes a dependency between two subsystems that causes them to be unconnected.
- The transaction doubles the number of dependencies between two subsystems.
- The transaction reduces the number of dependencies betweeen two subsystems by half.

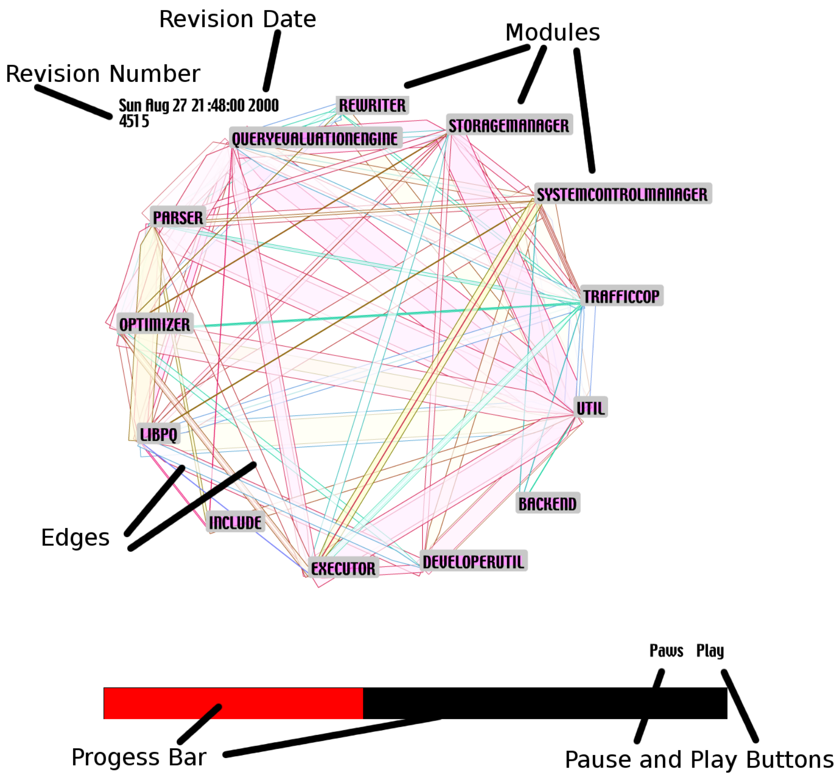

Figure 3. Screen-shot of *YARN* with *PostgreSQL*

3.3 YARN

The goal of *YARN* (Yet Another Reverse-engineering Narrative) is to provide a narrative animation; that is, the story of the evolution of a software project over time. *YARN* uses the animation parameters and *HistODiff* output to generate *YARN* Balls (animations) can be unravelled (watched) by the user to learn about the history of the system's architecturally significant changes.

YARN uses *HistODiff*'s graph output to create a graphical animation of the architectural changes of a system. The thickness of the edges suggests how many dependencies exist between two modules, we use the function $log^2(weight(u,v))$ to determine the edge's thickness based on its weight. The nodes are statically laid out so they don't change position over time. This allows for some sense of coherency between changes.

Edges are directed; when displayed, the edge of lesser weight is shown inside of the edge going in the reverse direction. Edges are also rendered transparently, thus intersections of edges are both visible and visually resolvable.

Edges can be animated into two different ways:

Cumulative view: Edges exist over the entire time that there is a dependency between two modules. This view emphasizes the current state of the system and what edges are have been changed.

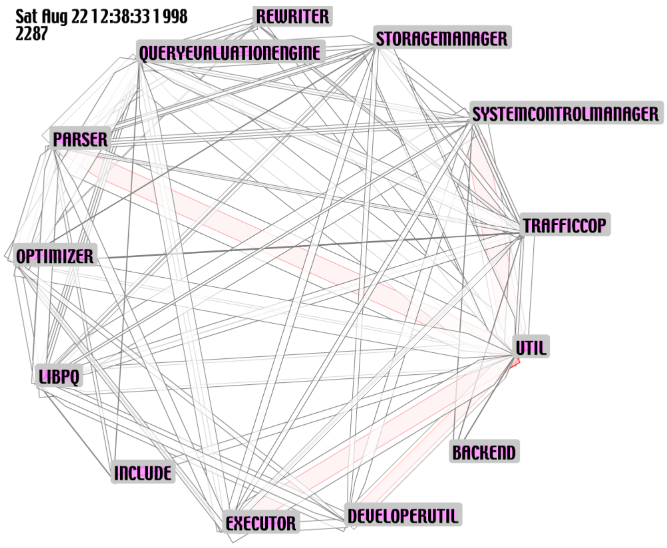

Figure 4. ***PostgreSQL YARN*** **Flipbook shot 3/6**

Non-cumulative view: Edges only exist per each change. This view emphasizes what the actual changes are by removing the extra information.

We have used several coloring schemes. Each uses color in a different way in the animation to emphasize certain aspect of the changes over time. Three such schemes are:

Color Changes on Modification: This coloring algorithm changes the color of modified edges; this (obviously) serves to highlight edges that change. When dependencies between two modules change, the edge is highlighted with the current color. The current color is a function of time, thus the colors of newly modified edges are different colors than previous edges. This allows older unchanged edges to maintain an out-dated color while edges that are changing display more current colors. Edges which change frequently are noticable due to their constantly changing color. Although this scheme illustrates that change is occurring, this coloring scheme can lead to some ambiguity: edges that get brighter often look like they are decaying rather than changing.

Highlight and Decay: Each edge that is modified is highlighted when a change occurs. It is highlighted by changing the color to a bright bold color; then over the period of a few changes the color of that edge decays back down to a neutral color. This color function emphasizes recent changes. Possible disadvantages include the decaying colors could look like new changes, also selecting an appropriate decay time could be difficult depending on how busy the graph is.

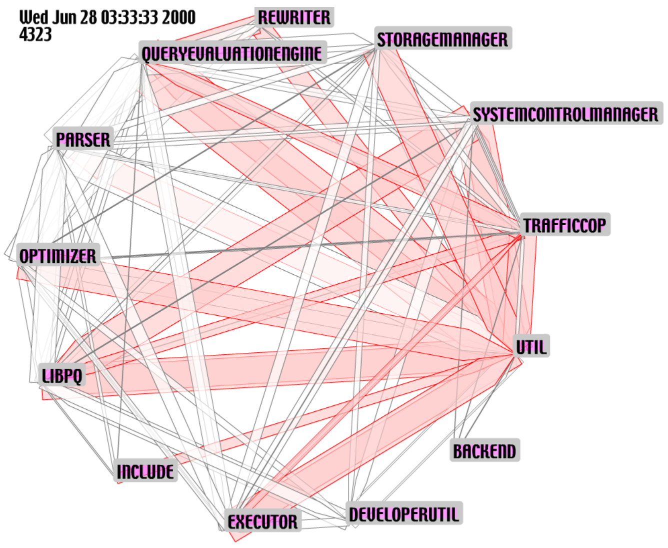

Figure 5. *PostgreSQL YARN* **Flipbook shot 4/6**

Highlight the Important changes: This coloring algorithm is much like highlight and decay algorithm except only the important changes are highlighted instead of just new changes. This view emphasizes changes that have been tagged as important.

All of the these algorithms assume that the edges are growing and shrinking in width and that only the edges are being animated.

Edge widths can be diplayed in several different ways:

Cumulative Width: This edge width function is a scaling function such as $log^2(weight(u, v))$ (where u and v are modules and $weight(u, v)$ is the number of dependencies from u to v at the current step). Cumulative Width shows how many dependencies currently exist between the two different modules.

Decaying Edge Width: The older an edge gets the more it decays and the more it shrinks in width. Over time an edge decays (shrinks) until it reaches a minimum width. When a change occurs it modifies the width of the edge back to its width according to the scaling function.

Edge Width as Age: Instead of the number of dependencies, edge width alone indicates when the last change to the dependencies between two modules.

The changes animated are transactions, and one frame represents one transaction. For instance, *PostgreSQL* had over 10,800 frames of animation.

In the top corner, the current date of the transaction is shown. Underneath is the order of the revision. At the bottom a time-line shows relationally where the transaction is with respect to the other revisions. The time-line is click-able to allow jumping throughout the evolution of the project.

The animations are created are vector graphics animations in SWF (Macromedia Flash) format, they can be embedded into webpages and viewed by most modern browsers. This could be used in the such hypermedia software evolution systems like the Software Bookshelf [6] or SoftChange [8].

Modules can be laid out manually or automatically. Automatic layout algorithms currently include radial or matrix layout. The radial layout is useful for systems like *PostgreSQL* where there are many dependencies.

Figures 1, 2, 4, 5, 6, and 7 depicts 6 frames from *YARN* (cropped) over time. Figure 3 depicts a screen-shot of *YARN* in action.

4. Case Study of *PostgreSQL*

4.1 Architecture

PostgreSQL is a well known open source DBMS that is in wide use. *PostgreSQL* has a well defined layered architecture (Fig. 10). The use of layering provides many advantages including the separation of concerns and abstraction.

The three layers are:

Client Interface Layer: This layer accepts inputs from the users through a variety of user interfaces. It submits the queries to the Backend layer below and returns the answers.

Backend: This layer parses the user's query and presents it to the optimizer which attempts to produce the most efficient execution plan for the evaluation. In order to execute the plan tree, this layer uses the I/O functions in the Data Store Layer.

Data Store Layer: This layer deals with managing space on disk, where the data is stored. Upper layers require this layer to write or read pages.

Important sub modules of the layers include:

LibPQ: enables the client or user to ask queries of the RDBMS.

System Control Manager: handles authentication and starts and stops `postgres` processes.

Traffic Cop: delegates query jobs to the various query sub modules.

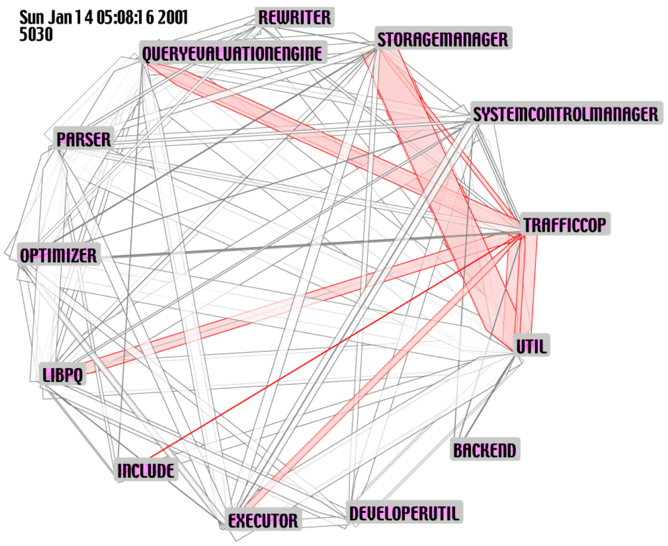

Figure 6. *PostgreSQL YARN* **Flipbook shot 5/6**

Parser: This modules tokenizes and parses SQL queries.

Rewriter: is the primary module for query rewriting queries which are recursive or can be optimized.

Optimizer: attempts to choose an optimal query plan structure for a given query.

Executor: executes the query plan (execution tree).

Command: is a query which doesn't need the Optimizer.

Catalog: is where the RDBMS stores the meta data, e.g information about tables, columns, values etc.

Nodes: are responsible for storing the queries in a specified common data structure called Nodes Structure.

Util: consists of utility procedures and routines that different modules use to do their jobs, especially Backend.

Storage Manager: is found inside of the data layer managing files and pages of the database.

4.2 A Visual Walk Through of PostgreSQL

We evaluate the history of architectural changes of *PostgreSQL* by manually inspecting a few "architecturally significant" transactions. *HistODiff* flagged out about 100 transactions that it considered to be architecturally significant; that is, they have either added or removed dependencies which makes two subsystems connected or disconnected, or have significantly increased or decreased the degree of coupling between subsystems. We divided the architecturally significant transactions into 3 intervals of equal number of transactions:

- 1996 to 1998: *PostgreSQL* is released as open source software. Portability and reimplementation of features such as ODBC are included (see figures 1 and 2).
- 1998 to 2000: *PostgreSQL* is still in flux, ODBC updates occurred and *PostgreSQL* was extended by the PL/pgSQL language (see figures 4 and 5).
- 2000 to 2005: *PostgreSQL* was being maintained, less features are added, more auditing and bug fixing occurs (see figures 6 and 7).

There were many significant changes from 1996 to 2005, and we have highlighted a few and have provided screenshots of the revisions. These screenshots help highlight the cumulative progression of architectural dependency over time as well they highlight the architecturally significant changes that occurred over time.

The first notable change is from figure 1 which shows the result of adding PL/TCL. PL/TCL is a extension to *PostgreSQL* which allows the TCL programming language to be used in stored procedures. The 2 dependencies that are were changed are highlighted, these were between the System Controller and Util (where PL/TCL was stored).

4.3 *PostgreSQL*: 1996 to 1998

The theme of *PostgreSQL* from 1996 to 1998 was the transition from a closed project to an open source project used by many users on many different platforms ranging from UNIX to Win32. It covered releases 1.06 to 6.3.

Security: Many changes were related to authentication. These included host and plain-text password based authentication, and a check if *PostgreSQL* was running as a root user.

Portability: There were two kinds of portability issues dealt with: wrapping systems calls and distributing the output of tools such as bison and flex. System call wrapping included wrapping signal with portable signal call code, as well as providing portable file open and close calls (for SunOS). Flex and Bison output was added for systems which lacked compatible versions of the utilities (FreeBSD and AT&T UNIX).

Extensibility: The server programming interface was added, it allows *PostgreSQL* to be used from languages such as C and TCL. As well PL/TCL was added which allows stored procedures to use TCL. Figure 1 depicts this change.

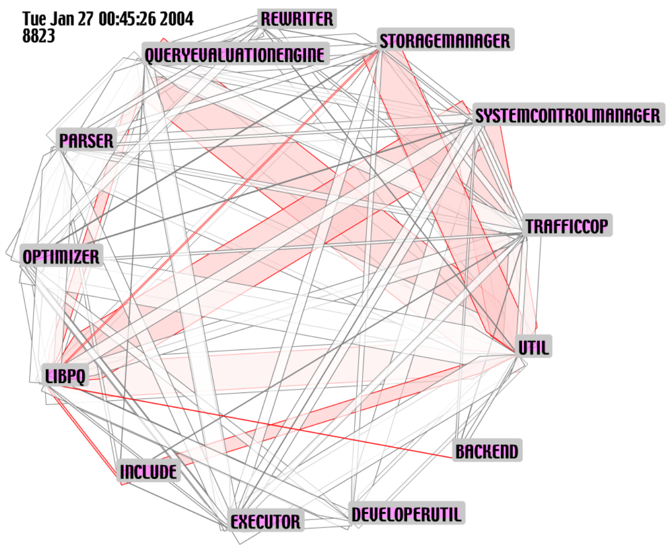

Figure 7. *PostgreSQL YARN* **Flipbook shot 6/6**

Updated Code: Old transactions code *Time Travel* was removed. The old ODBC driver was updated with a new ODBC driver. Figure 2 depicts this change. There were also issues regarding the precedence of attributes from certain tables not being handled appropriately. The heap tuples code was optimized by inlining the code via a macro.

4.4 *PostgreSQL*: 1998 to 2000

The next period is from 1998 to 2000, which approximately corresponds to 11 releases from Releases 6.4 to Release 7.0.3.

ODBC: they have updated ODBC driver to version 0.0244.

Extension: The developers added a second procedural language PL/pgSQL and then later moved from the contributions directory into the official distribution. This change is depicted in figure 4

Enhancements: Developers improved the client/server asynchronous communication; added support for outer joins, and read/write locks; rewrote the *Memory Management Process(mmgr)* utilities (this is depicted in figure 5) for better handling of out of memory error; changed the header comments of functions.

4.5 *PostgreSQL*: 2000 to 2005

The period 2000 to 2005 covered *PostgreSQL* releases 7.0.3 to 7.4.8. This period contains one third of the revisions that were flagged important in the repository. Large changes to the source included new implementations of SQL statements, improving triggers, JOINS and dropped columns. Most of the changes were maintenance and improvement of properties such as robustness, security and performance. Security improvements were spurred by problems found by “white hat hackers”, who found flaws related to interrupt handling and critical sections. The changes that occurred due to these flaws are depicted in figure 6. Better support for Win32 signal handlers was also added (see figure 7). Figure 8 depicts the changes made to the system starting from 2001 to 2005, also the larger changes are highlighted.

5. Future Work

The main extension to *YARN* will be to its *YARN* Ball user interface. More layout algorithms will be added to YARN. We plan to add more interactivity to the animation. This would include dragging and dropping modules as well as expanding sub modules. This would allow for hierarchical navigation of sub modules. Different views of the

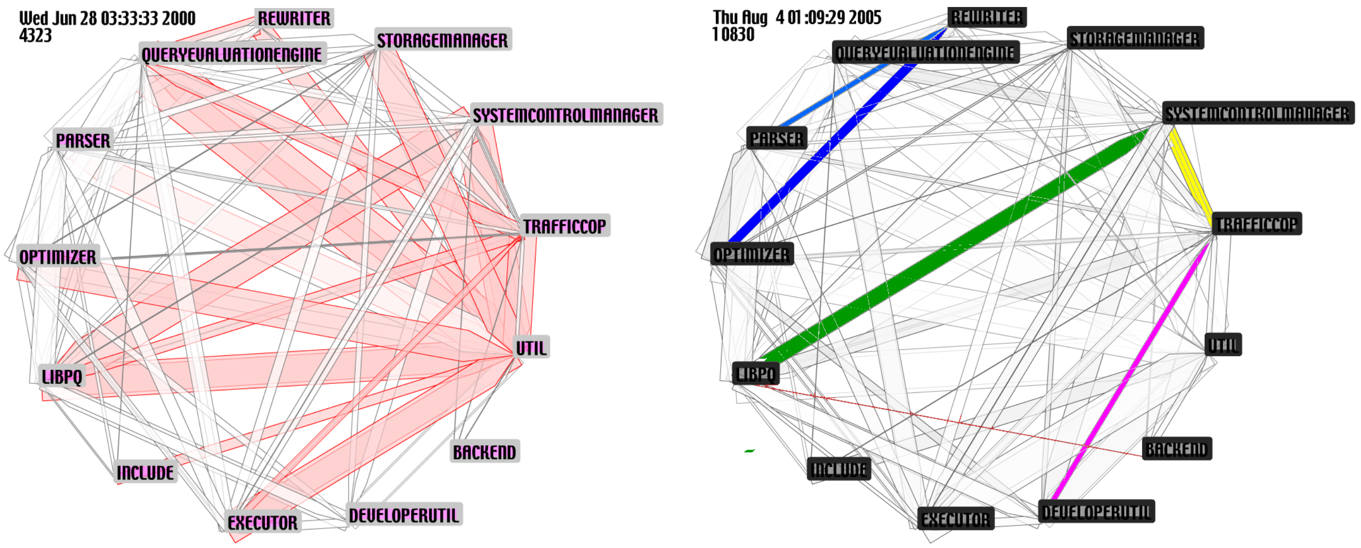

Figure 8. Important architectural changes done during the last 5 years

architecture are planned such as source control view which shows which modules are coupled together per commit.

Other work includes evaluating the use of animation for maintenance work. We have yet to answer the question if this animation is useful to developers or just researchers.

6. Conclusions

In conclusion we proposed and implemented a system which can extract the dependency information from a repository and aggregate it into an animation. These animations, or *YARN* Balls, then can be embedded into webpages or shared in order to communicate change based dependency information about software projects. Many different kinds of animations can be produced.

The contributions of this paper thus included: an approach for animated the evolution of a project's dependencies in a coherent static manner; a system to view changes and the cumulative effects of the changes; provide a finely grained view of the evolution of the project.

References

[1] CPPX: Open Source C++ Fact Extractor. Technical report, University of Waterloo. http://swag.uwaterloo.ca/ cppx.

[2] Exuberant Ctags. Technical report. http://ctags.sourceforge.net.

[3] R. Baekcer and D. Sherman. Sorting out sorting. 30 minute colour sound film, 1981.

[4] D. Church. A survey of techniques for the recovery and observation of software evolution. Unpublished, 2004.

[5] X. Dong, L. Zou, and Y. Lin. Conceptual/concerte architecture of postgresq. Technical report, University of Waterloo, 2004.

[6] P. Finnigan, R. Holt, I. Kalas, S. Kerr, K. Kontogiannis, H. Muller, J. Mylopoulos, S. Perelgut, M. Stanley, and K. Wong. The software bookshelf. pages 564–593, November 1997.

[7] H. Gall, M. Jazayeri, and C. Riva. Visualizing software release histories: The use of color and third dimension. In *ICSM*, pages 99–108, 1999.

[8] D. M. German, A. Hindle, and N. Jordan. Visualizing the evolution of software using softchange. In *Proceedings SEKE 2004 The 16th Internation Conference on Software Engineering and Knowledge Engineering*, pages 336–341, 3420 Main St. Skokie IL 60076, USA, June 2004. Knowledge Systems Institute.

[9] A. Hassan. Mining software repositories to guide software development. http://plg.uwaterloo.ca/ aeehassa/home/pubs.html, 2005. Accessed July.

[10] A. E. Hassan and R. C. Holt. C-REX: An Evolutionary Code Extractor for C - (PDF). Technical report, University of Waterloo. http://plg.uwaterloo.ca/ aeehassa/home/pubs/crex.pdf.

[11] A. Marcus, L. Feng, and J. I. Maletic. 3d representations for software visualization. In *SoftVis '03: Proceedings of the 2003 ACM symposium on Software visualization*, pages 27–ff, New York, NY, USA, 2003. ACM Press.

[12] H. A. Muller and K. Klashinsky. Rigi - A System for Programming-in-the-large. Technical report, In IEEE 10th International Conference on Software Engineering(ICSE-1998), April 1998.

[13] M.-A. D. Storey, C. Best, and J. Michaud. Shrimp views: An interactive environment for exploring java programs. In *9th International Workshop on Program Comprehension (IWPC 2001), 12-13 May 2001, Toronto, Canada*, pages 111–112. IEEE Computer Society, 2001.

[14] J. Wu and R. C. Holt. Linker-Based Program Extraction and Its Uses in Studying Software Evolution. Technical report, In Proceedings of the International Workshop on Unanticipated Software Evolution (FUSE 2004), March 2004.

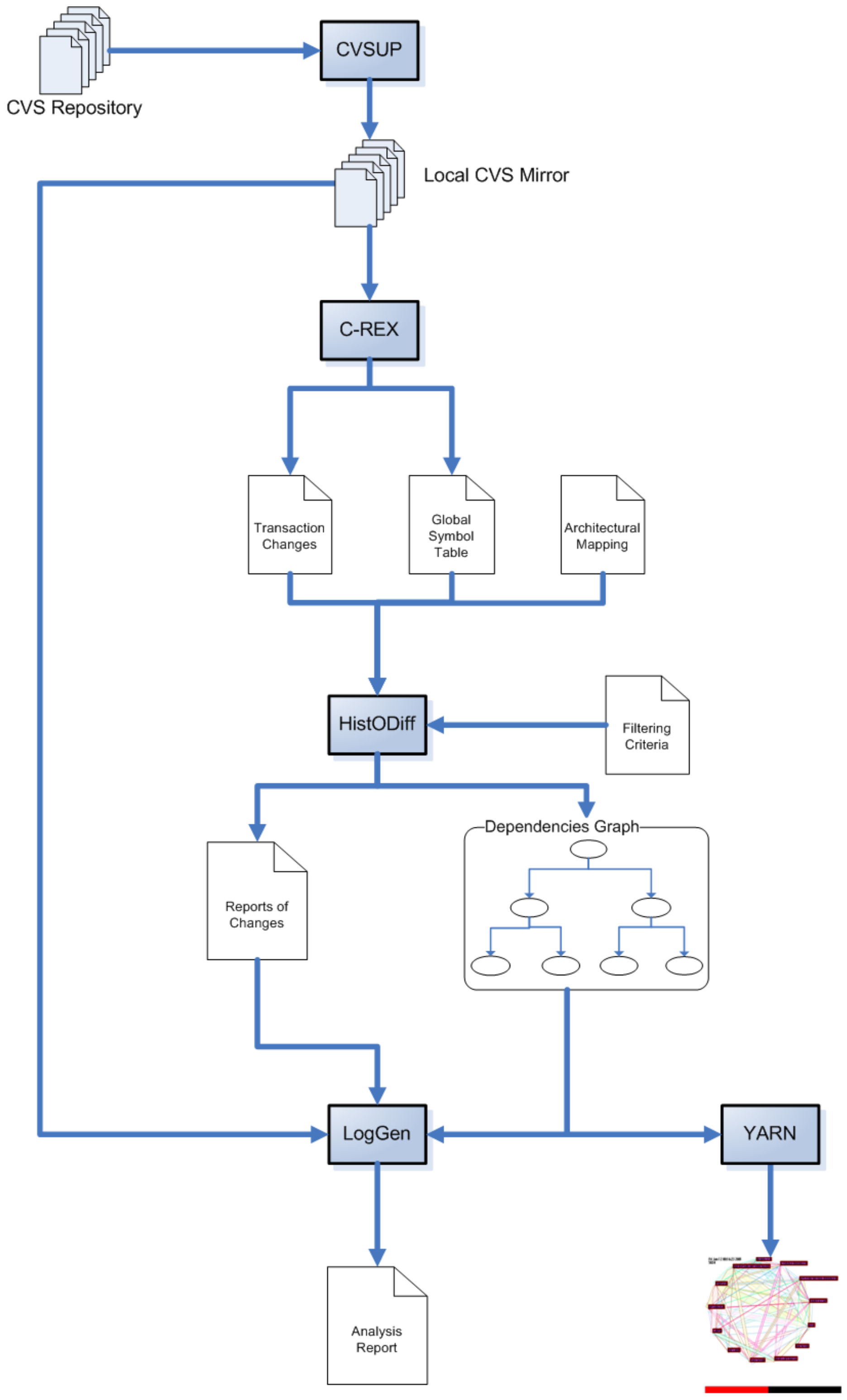

Figure 9. Flow Control of Extraction with our tools

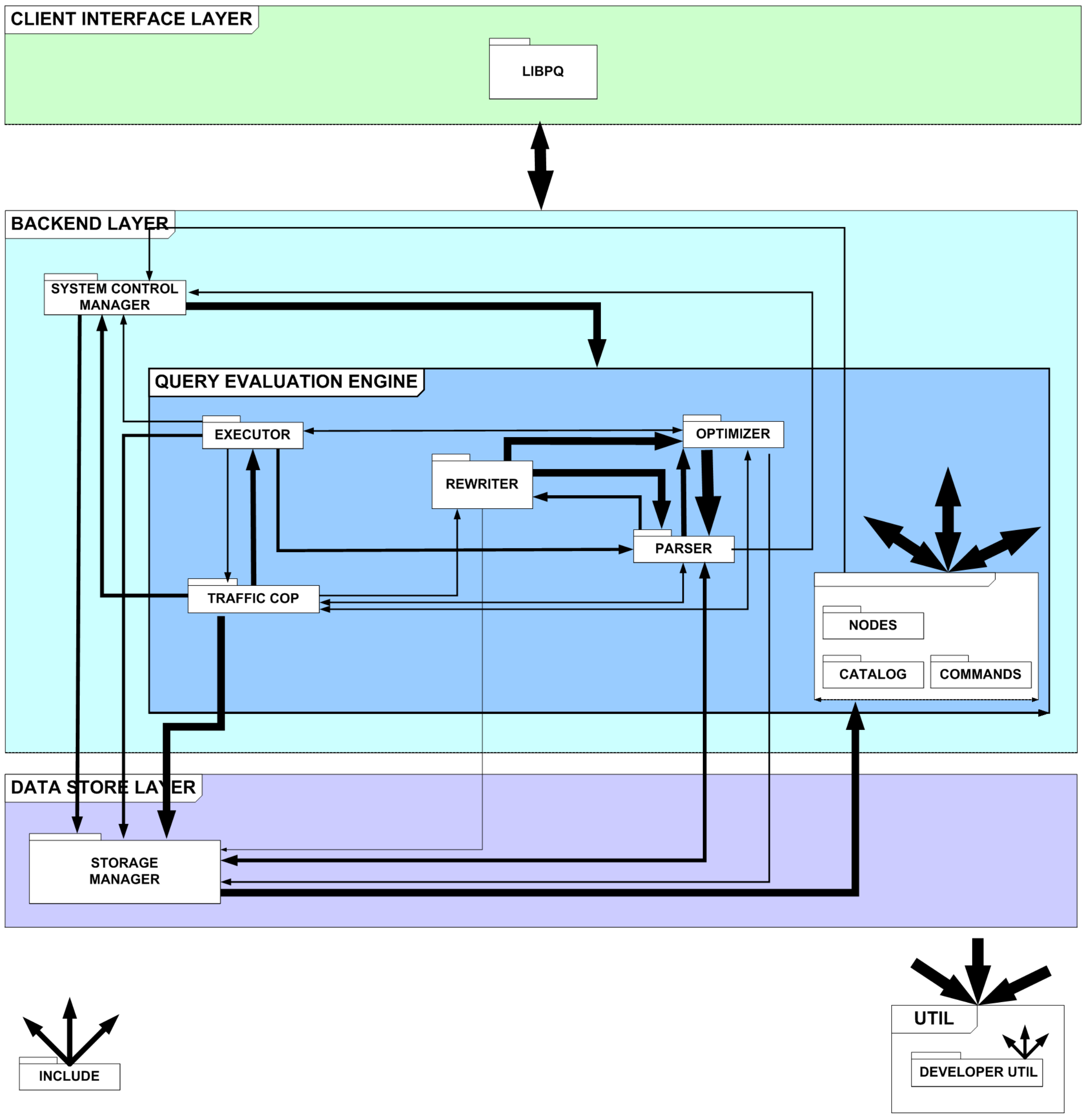

Figure 10. Top Level Architecture of *PostgreSQL*