YARN: Animating Software Evolution

Abram Hindle, 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 modeling, 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 change. 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 re-factored 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 their 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 often not maintained as the system ages, and automated architecture visualization tools tend to emphasize static views of the current version of the system.

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 using an animated visualization.

Animation is an intuitive and natural way to show change over time visually. We 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 of the changing 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.

The set of data we analyze is often quite large and is difficult to represent all at once in a way that is both meaningful and useful. Animation enables us to traverse this rich and large information space and interpret the data visually rather than in a textual or statistical way.

Keeping the positions of the nodes fixed encourages the user to create a stable mental model of the system, allowing him/her to concentrate on the changing dependencies over time and observe the system's architectural properties as they evolve. Animation exploits the temporality of the data in the repository and helps to illustrate the dynamic behavior of the evolving dependencies between modules. We employ two approaches to representing the dependencies: the cumulative addition of dependencies and the difference of edge weights between changes.

Our motivation was that we would like to see what the architecture looked like at each revision. We had some idea of what the changes would look like, but in a textual form the differences were not obvious. There was so much data it was hard to interpret without some abstraction or visualization. As well we found that it was difficult to communicate what we saw to others. We wanted to be able to share our results with others to show them what we saw without hav-

ing to do the same extraction. We hope this animation will better illustrate the change in coupling over time between modules.

In this paper, we describe our approach to extracting, modeling, and animating architectural evolution, as implemented in our tool *YARN* . We have included an example use of *YARN* and its generated *YARN* Ball animations in a flip-book-like form (figures 1, 2, 4, 5, 6, 8); thus, the reader can manually animate the printed YARN Ball like a flip-book.

2. Related Research

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 of source code [10]. Telea et al. [14] used animation for interactivity rather than to represent time. Mesnage et al. [11] created web embeddable presentations of software evolution matrices using VRML.

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. [4].

3. Tools

C-REX is used to extract the architectural information from the CVS repository of *PostgreSQL*. 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.

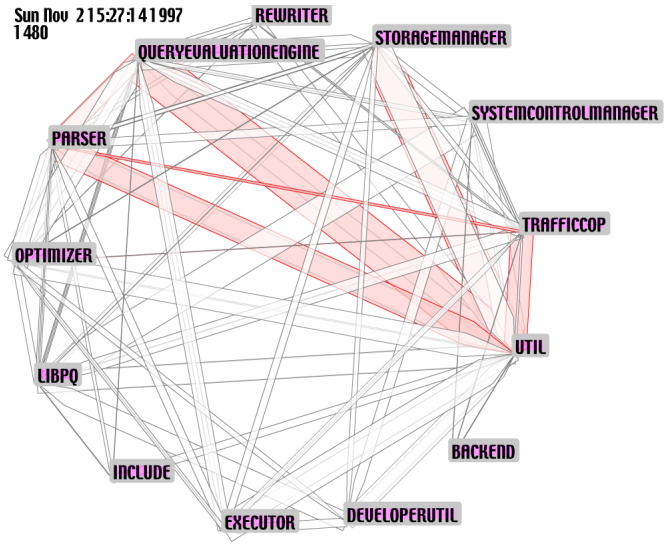

Figure 1. *PostgreSQL YARN* **Flip-book shot 1/6**

3.1 C-Rex

We use *C-REX* [9] 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* [15], *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 type, 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.

Unfortunately *C-REX* is unaware of the actual types within a system and is thus not aware of virtual dispatch that is prevalent in many languages like C++ or Java.

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 modeled. Two kinds of graphs can be produced: a dependency graph between subsystems over time, and a difference graph that shows the dependency changes between 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 is a huge number of changes, but not all are architecturally significant. Large change transactions are 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.

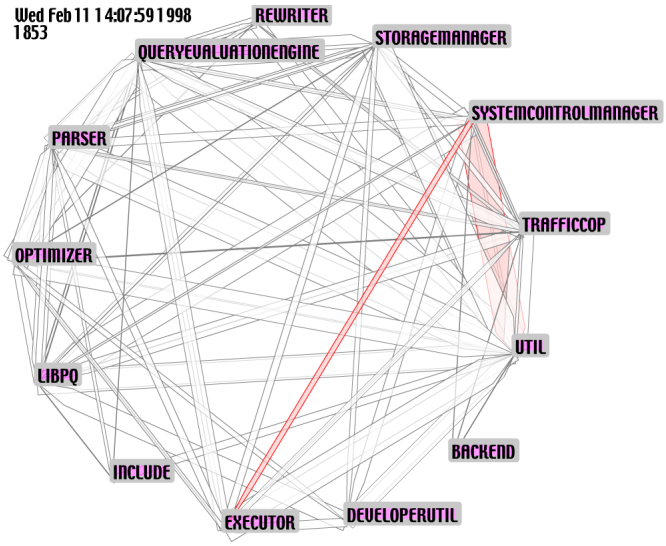

Figure 2. *PostgreSQL YARN* **Flip-book shot 2/6**

Our filtering heuristics highlight transactions that affect the number of dependencies between the top level subsystems and that satisfy one or more 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 between two subsystems by half.

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 unraveled (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 are shown 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.

Non-cumulative view: Edges are shown only when they change. This view emphasizes what the actual changes are by removing the extra information.

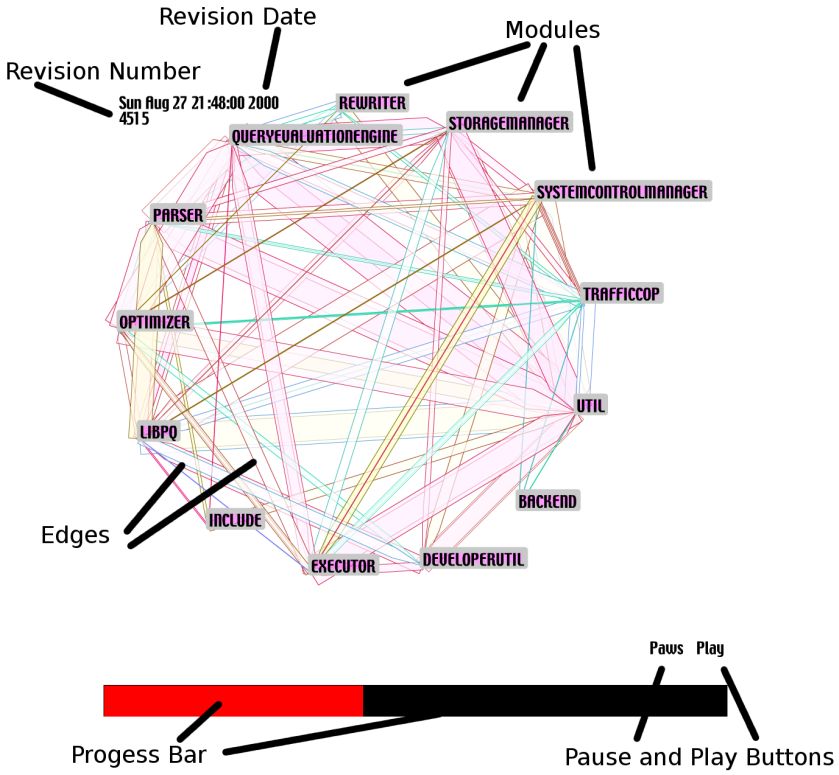

Figure 3. Screen-shot of *YARN* with *PostgreSQL*

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:

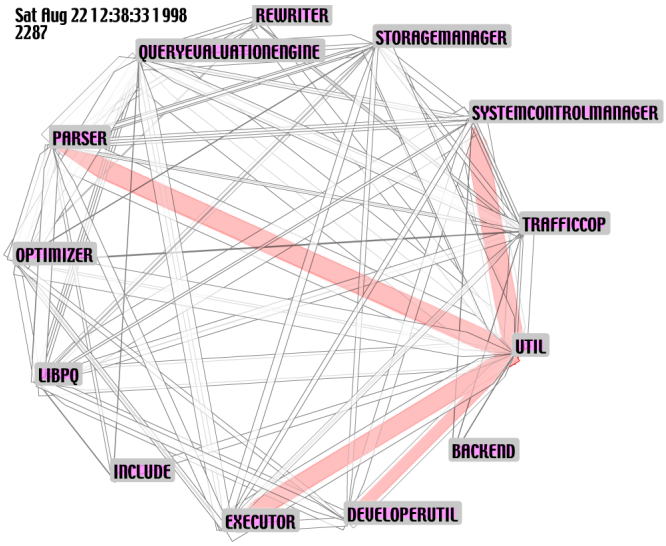

Figure 4. *PostgreSQL YARN* **Flip-book shot 3/6**

Color Changes on Modification: This coloring algorithm changes the color edges each time the edge is modified; 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 noticeable 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. Suggested uses for this scheme is to highlight the rate of changes and show all the changes that are occurring.

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. Highlight and decay emphasizes new changes, old changes disappear rapidly, it makes the current change more obvious by making the past fade away.

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. Our algorithms designated certain changes to be important based on change metrics. This view emphasizes changes that have been tagged as important. This scheme is useful to highlight changes that flagged as important instead of each and every change that comes in. This scheme demonstrates how frequent or infrequent "important" changes are.

All of the these algorithms assume that the edges are growing and shrinking in width and that the nodes remain stationery while the edges are being animated. Edge widths can be displayed in several different ways:

Cumulative Width: This edge width function is a scaling function such as $log^2(weight(t, u, v))$ (where u and v are modules and $weight(u, v)$ is the number of dependencies from u to v at the current step t). Cumulative Width shows how many dependencies currently exist between the two different modules. This scheme is useful as it gives a context and show the size of the system. It shows slowly bloating dependencies of the system or the drastic reductions.

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. This scheme emphasizes the current change over the past changes, it doesn't allow for much historical comparison but serves to highlight the content of the revision.

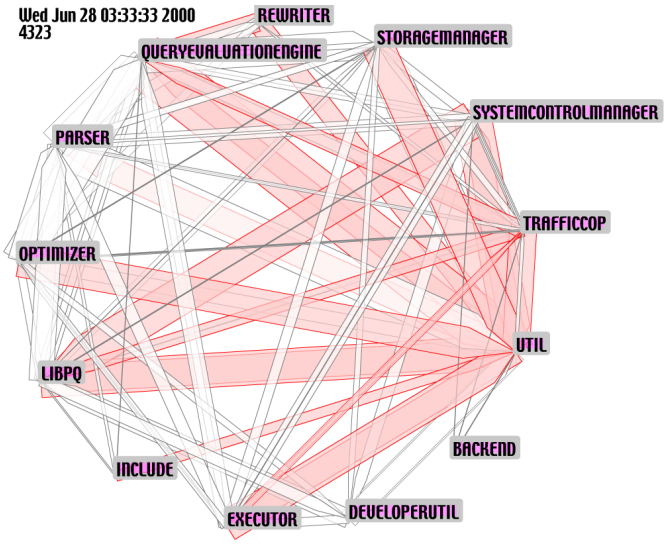

Figure 5. *PostgreSQL YARN* **Flip-book shot 4/6**

Edge Width as Age: Instead of the number of dependencies, edge width alone indicates when the last change to the dependencies between two modules. This scheme reflects the frequency of change in dependencies between to modules by the edge width itself. This is useful to determine which dependencies are frequently added.

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; see Figure 3. 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 to any part of the evolution of the project.

The animations are created in SWF (Macromedia Flash) format using vector graphics, and can be embedded into web-pages and viewed by most modern browsers. This could be used in hypermedia software evolution systems such as 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 8 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 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, expands it, and presents it to the optimizer which uses information 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: handles the flow control of queries.

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 Timeline 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 history, consisting of a sequence of transactions, into 3 parts with equal numbers of flagged changes.

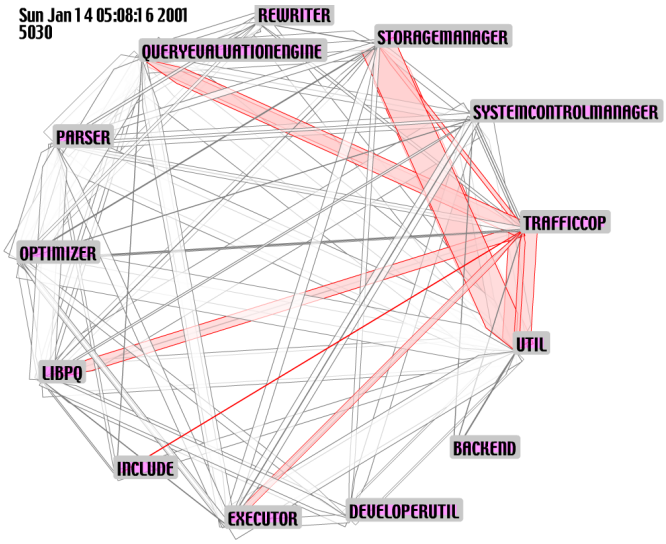

Figure 6. *PostgreSQL YARN* **Flip-book shot 5/6**

The length of time taken by these parts varied considerably because the rate of occurrence of changes varied.

- 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* is maintained, as a reasonably mature system. Less features are added, more auditing and bug fixing occurs (see figures 6 and 8).

There were many significant changes from 1996 to 2005, and we have highlighted a few and have provided screen-shots of the revisions. These screen-shots help highlight the cumulative progression of architectural dependency over time as well they highlight the architecturally significant changes that occurred over time. We will choose 6 notable changes, and discuss them.

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. Figure 7 depicts the changes made to the system starting from 2001 to 2005, also the larger changes are highlighted.

5. Visual Story of PostgreSQL

The first notable change (figure 2) depicts the reimplementation of the ODBC driver by Insight. This change was highlighted because of the new dependencies between LibPQ and Parser and Utils where were more than doubled.

The second notable change is from figure 1, the change is highlighted using the Decay and important changes only feature. We can see that two corresponding dependencies that changed during this time are highlighted, namely those between the System Controller and Util. Although YARN does not tell the full story, it gives us a feeling for the story. A brief overview of changes. Further digging into the revisions around this time reveals that the embedded language PL/TCL was added. PL/TCL is a extension to *PostgreSQL* which allows the TCL programming language to be used in stored procedures.

The added dependencies between Util and System Control Manager were the most notable in figure 4 (which was color-enhanced to make the change noticeable for print). By investigating the revisions we found this was the addition of the PL/PgSQL embedded language. A language for stored

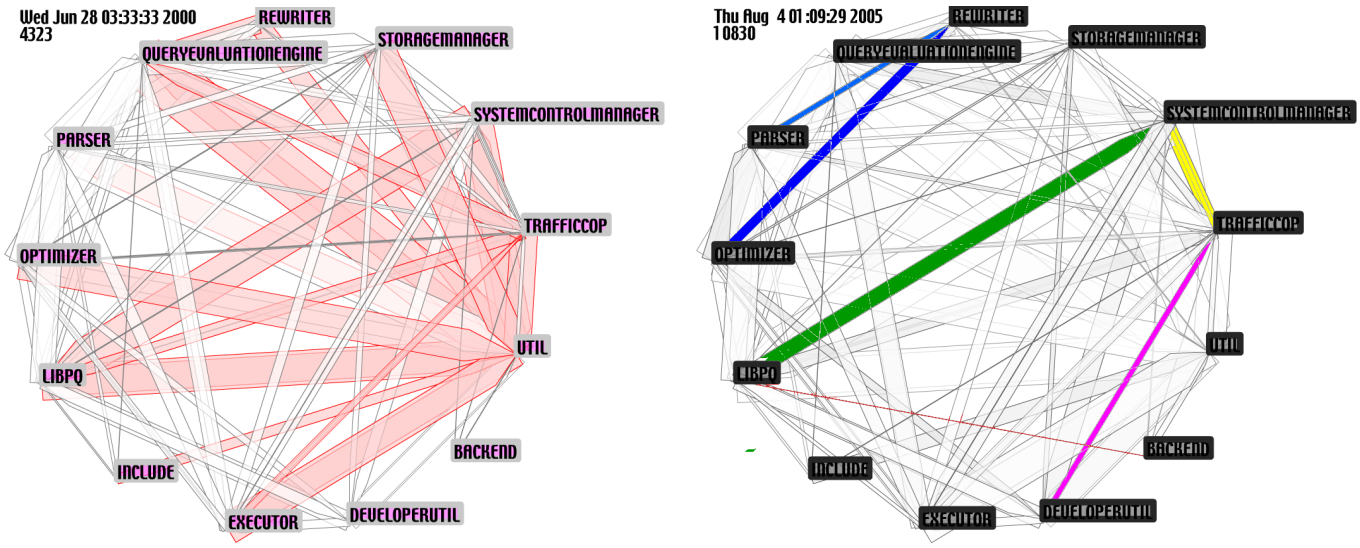

Figure 7. Important architectural changes done during the last 5 years

procedures in *PostgreSQL* which wasn't TCL. It is important and notable because the inclusion of language crosscut multiple modules and introduced more coupling.

The fourth change (figure 5) we show is actually a increase and a reduction of coupling. This change was highlighted by our importance metric due to a reduction of dependencies. It is hard to see since we haven't shown the immediate preceding changes but the coupling of LibPQ to the System controller has been reduced. Most of the other changes seem to show an increase in coupling. Investigation of the related revisions reveals that the *Memory Management Process(mmgr)* submodule was rewritten. This change also included several bug fixes and removal of dead code.

The fifth change (figure 6) we see new coupling from Parser and Optimizer to the Rewriter, Include to TrafficCop, Include to LibPQ and LibPQ to Executor. These changes were made to *PostgreSQL* due to the advice given by "white hackers" regarding security holes in the interrupt and issues with some of the critical sections. Instead of allowing the interrupt to happen anywhere, they will be handled at well defined spots.

The sixth change (figure 8) was important due to the cross cutting changes that occurred and the new dependencies between various modules and LibPQ and Include. Further digging reveals that there was was a change to the Win32 signals code. This change made signal handling more portable via abstractions for the `kill` and `sigsetmask`.

6. Survey

We created an informal user survey that could be answered over email or live over chat. 10 people took the survey, their backgrounds ranged from high school Source Forge contributors, CS undergraduate students, PhD students to full time developers. Most respondents answered all the questions.

All users could use the functionality of a YARN Ball without much difficulty. The navigation controls and the play and pause buttons intuitively worked. Most users expected the pause animation on jump feature.

Users did not know initially what the animation was about. They had to be told it was about revisions to the source code and coupling. Some users assumed this was the run time coupling until they were informed otherwise.

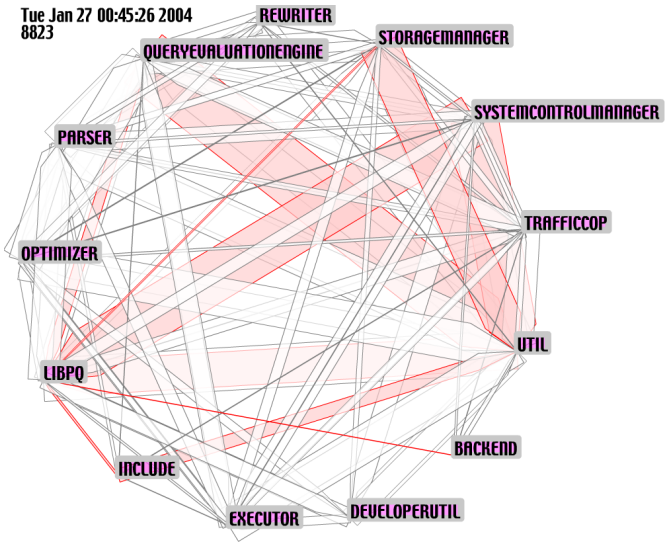

Figure 8. *PostgreSQL YARN* **Flip-book shot 6/6**

Color-wise, most users like the highlight and decay based color schemes where changes are highlighted in red and then slowly decay back to a neutral color.

Users could infer coupling from the animation for changes with large and small coupling. Generally users understood what the edge weights meant. Users did not seem to gain a better understanding of software architecture. Due to a general lack of experience with PostgreSQL many didn't have any expectations with regards to the PostgreSQL architecture, those who did have expectations said they were met, and PostgreSQL was not unique from other software like Linux.

Some users said they would use the YARN Balls if they were available but some where not about to go out of their way to produce them.

Survey respondents suggested improvements such as making the visualization 3D or project it onto a sphere, or use other metrics. Some said more data should be displayed like the change log and that a legend was needed. Some suggested tweaking parameters like color decay speed etc. or color to indicate the starting or ending module.

7. Future Work

One 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 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.

8. Conclusions

In this paper, we have described an approach to extracting, modeling, and animating architectural evolution of software systems, as implemented in our prototype tool YARN. The YARN tool aggregates this data into animations, or YARN Balls, which can be explored by the user to better understand the architectural evolution of the system under study. These YARN Balls can be embedded into web pages or shared in order to communicate change based dependency information about software projects. Many different kinds of animations can be produced.

The main contributions of this work includes: 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; animations which provide a finely grained view of the evolution of the project; and an informal user survey evaluating the perception and usefulness of the visualization.

Acknowledgments

Part of this research was funded by an NSERC PGS Scholarship.

References

[1] CPPX: Open Source C++ Fact Extractor. Technical report, University of Waterloo, 2001. http://swag.uwaterloo.ca/ cppx.

[2] Exuberant Ctags. Technical report, May 2006. http://ctags.sourceforge.net.

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

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

[5] S. Ducasse, M. Lanza, A. Marcus, J. I. Maletic, and M.-A. D. Storey, editors. *Proceedings of the 3rd International Workshop on Visualizing Software for Understanding and Analysis, VISSOFT 2005, September 25, 2005, Budapest, Hungary*. IEEE Computer Society, 2005.

[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. E. Hassan and R. C. Holt. C-REX: An Evolutionary Code Extractor for C - (PDF). Technical report, University of Waterloo, 2004. http://plg.uwaterloo.ca/ aeehassa/home/pubs/crex.pdf.

[10] 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.

[11] C. Mesnage and M. Lanza. White coats: Web-visualization of evolving software in 3d. In Ducasse et al. [5], pages 40–45.

[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] A. Telea and L. Voinea. Interactive visual mechanisms for exploring source code evolution. In Ducasse et al. [5], pages 52–57.

[15] 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.