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 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 generates* YARN *"balls" (animations) that a viewer can unravel (watch). The animation is based on a static layout of the modules connected by animated edges that model the changing dependencies. The edges can be weighted by the number of dependencies or the importance of the change. We demonstrate our approach by visualizing the* PostgreSQL *DBMS.*

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. The visualization and comprehension of change over time is a problem that faces software evolution.

For a given system, a developer may have access to static "snapshots" of its software architecture and its internal dependencies. These snapshots may be hand-drawn by an expert or automatically extracted from the source code. However, in practice these approaches are not well suited to the task of understanding how the system has evolved. Hand drawn snapshots are often not maintained as the system ages, and automated architecture visualization tools tend to emphasize a static view 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. Thus software architecture needs to be extracted incrementally to reflect the historical changes. In our case, we establish a baseline architecture for the system, and then 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.

Animating the changes is an intuitive and natural way to compare changes visually over time. We start by showing the state of the architectural dependencies within the system at a chosen baseline version, and then show the user the subsequent changes progressively as animations of the changing architectural visual model. Change can be shown cumulatively. Cumulative views allow us to compare instances sequentially, this helps us to compare the dependencies at two different points in time, and animate the transition.

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's structure, allowing the user to concentrate on the interactions between changing dependencies over time and to observe the interactions between the system's components. 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 represent the dependencies: the cumulative addition of dependencies and the difference of edge weights between changes.

Our tool, *YARN* (Yet Another Reverse-engineering Narrative), generates animations of the changing dependencies (edges) between subsystems (vertices) of a project. These changes are animated via varying edge width and edge color, against statically placed vertices which represent subsystems of the project being studied. Figure 3 provides an annotated look at a *YARN* ball produced based on *PostgreSQL*.

Our primary motivation behind *YARN* was that we wanted to visualize the changing architecture and share these visualizations with others. We wanted to convey our

mental model of the software[10] via modular decomposition, yet we had to deal with a large amount of data which was hard to interpret in text form. *YARN* balls help us visualize change and share these visualizations with others. *YARN* balls could be useful for managers or newcomers just to summarize the concrete architectural state of a project, or show how the project has evolved.

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* on *PostgreSQL* 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 [11]. Telea et al. [15] used animation for interactivity rather than to represent time. Mesnage et al. [12] created web embeddable presentations of software evolution matrices using VRML. Beyer et al. [4] used animation and software evolution metrics to storyboard changes to files.

Our architectural views are similar to those of Rigi [13] and Shrimp [14]. Shrimps uses animation to support iterative navigation instead of representing change over time. Finally, we note that our architectural model of *PostgreSQL* was adapted from that of Dong et al. [5].

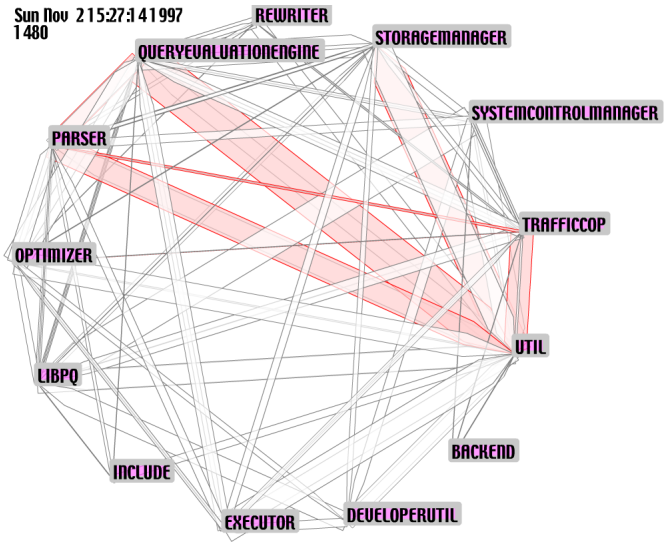

Figure 1. *PostgreSQL YARN* **Flip-book shot 1/6**

3. Tools

C-REX is used to extract the architectural information from the CVS repository of *PostgreSQL*. Once extraction is done, we run *HistODiff* (part of *YARN*), 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. *YARN* reads these dependency graphs and then produces *YARN* Ball animations that can be played back by the user.

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* [16], *RIGI* [13], 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 different programming language dialects, syntax errors, etc. In this situation, many 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 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. *HistODiff* uses these transactions to maintain its model of the system.

Within each changed entity, *C-REX* keeps tracks of changes in entity types, dependencies, and comments. 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. *C-REX*'s output is then processed by our tool, HistODiff.

3.2 HistODiff

HistODiff is used to analyze *C-REX* output, and create graphs for *YARN* to animate. HistODiff performs many tasks such as symbol mapping, architectural mapping, lifting and filtering.

Symbol Mapping: *C-REX*'s output consists of many identifiers and symbols per transaction, *HistODiff* resolves these symbols and identifiers to functions and macros.

Architectural Mapping: *C-REX* produces a list of transactions where entities such as functions, macros, and variables are added, deleted, and modified. *HistODiff* maps these transactions to architectural entities. These transactions are used to update the architectural dependency graph per each commit.

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. It makes the assumption that each file is associated with a single module within the module hierarchy.

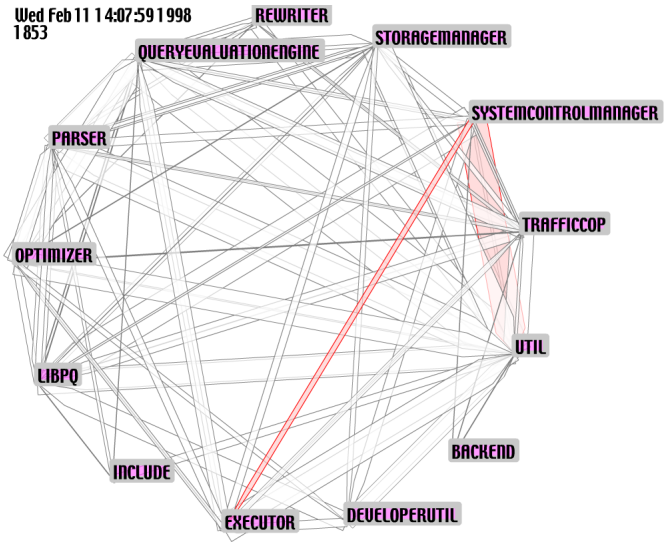

Figure 2. *PostgreSQL YARN* **Flip-book shot 2/6**

Filtering: In large projects such as *PostgreSQL* there are many 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 rule violation or an important feature addition.

Our filtering heuristics retain transactions that affect the number of dependencies between the top level subsystems 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 *HistODiff* output along with some animation parameters to generate *YARN* Balls (animations), which can be unraveled (watched) by the user, in order 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. In the animation, the thickness of the edges represents how many dependencies exist between two modules. In figures 1 through 8, we can see the modules don't change position. They are laid out in a radial pattern due to the high coupling.

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.

YARN animates edges in different ways:

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

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

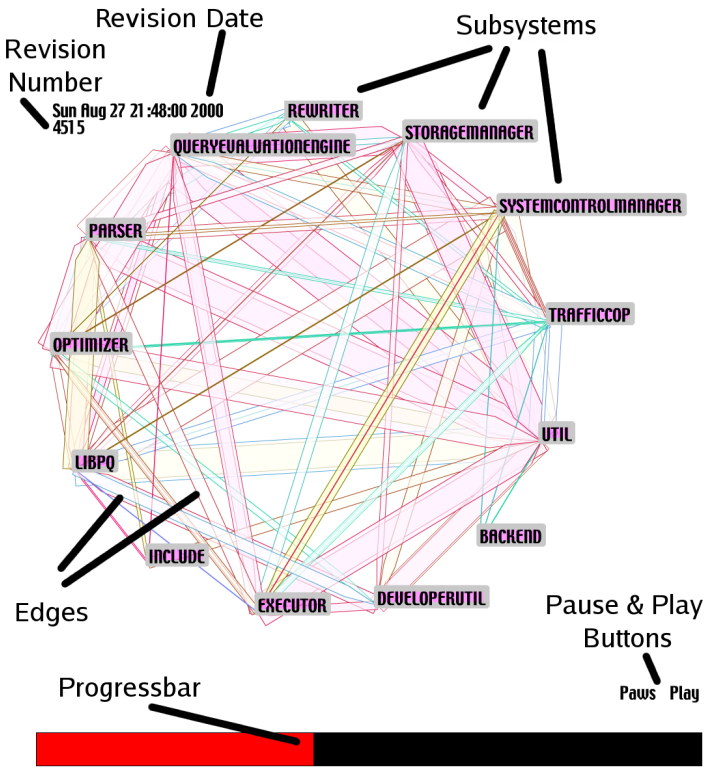

Figure 3. Screen-shot of *YARN* with *PostgreSQL*

All of the the following algorithms assume that the edges are growing and shrinking in width and that the nodes remain stationery while the edges are being animated.

YARN supports several coloring schemes. Each uses color in a different way, in the animation, to emphasize certain aspect of the changes over time. Three of these schemes are:

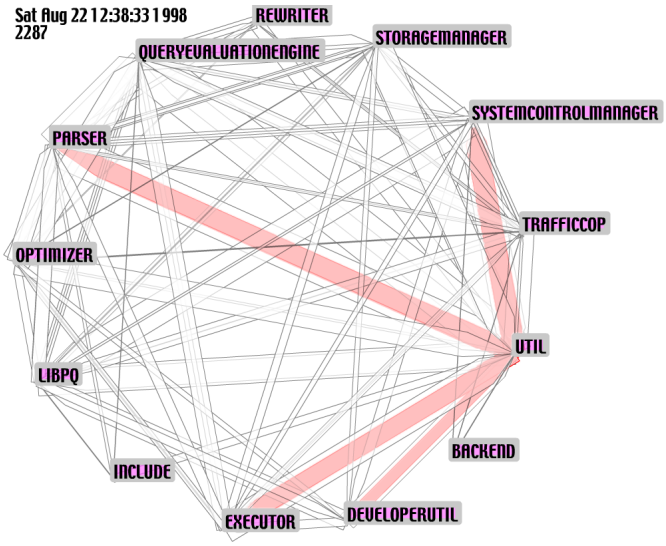

Figure 4. *PostgreSQL YARN* Flip-book shot 3/6

Color Changes on Modification: This algorithm changes the color of the edges, each time the edge is modified. This serves to emphasize edges that change. Per each revision a new color is assigned. This color changes gradually over time. When dependencies between two modules change, the edge is highlighted with the current color. This means edges that change frequently will flash with color. Frequent edges will be colored similarly while edges which are infrequently changed will have out-dated colors. In figure 3 we can see this algorithm in use, the edges that have a similar color were changed at the same time, where as edges with distinct colors have not been changed recently. This coloring scheme can be ambiguous; based on the colors chosen, edges getting brighter or darker might suggest decay rather than change to a user. The suggested use for this algorithm is to emphasize the rate of change and show all of the modifications 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. The flip book figures (1) uses this method.

Possible disadvantages are that 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 emphasize new changes, as 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 the highlight and decay algorithm except only the important changes are highlighted instead of just the new changes. Our algorithm has designated certain changes as important based on our importance metric. The flip book figures (1) are similar to this method. This algorithm is useful for highlighting changes that are flagged as important as it demonstrates how the frequency of "important" changes.

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 $weight(t, u, v)$ 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 shows the current state of the architecture. This algorithm shows the accumulation of dependencies up to the current time.

Decaying Edge Width: The older an edge gets the thinner it gets. Over time an edge shrinks (decays) until it reaches a minimum width. When a change occurs, the edge gets thicker again. 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.

Edge Width as Age: Instead of the number of dependencies, edge width indicates when the last change happened to the dependencies of between modules. This scheme reflects the frequency of change in dependencies between two modules by the edge width itself. This can be used to emphasize the dependencies that are frequently added.

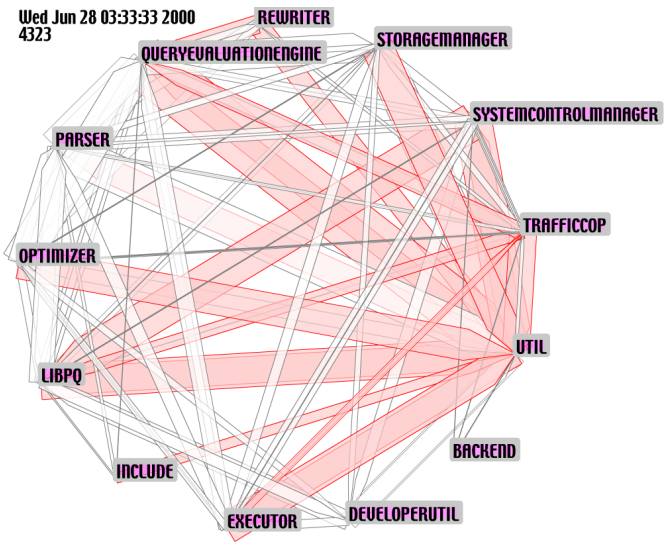

Figure 5. *PostgreSQL YARN* **Flip-book shot 4/6**

3.4 Implementation

The animated changes are transactions. One frame represents one transaction; *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 relatively when the transaction occurs. The time-line is interactive, it allows jumping to any part of the evolution of the project. The "Paws" (a reference to cats and yarn) and play buttons pause and play the animation.

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 layouts. The radial layout is useful for systems like *PostgreSQL*, where there are many dependencies.

Figures 1, 2, 4, 5, 6, and 8 depict 6 frames from a *YARN* ball of *PostgreSQL* (cropped) using the Highlight and Decay color function and the Cumulative Width edge function. Figure 3 depicts a screen-shot of *YARN* in action.

4. Case Study of *PostgreSQL*

PostgreSQL is a well known open source DBMS that is in wide use. *PostgreSQL* has a well defined layered architecture. The three layers are:

Client Interface Layer: This layer accepts input 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.

Using figure 5 going clockwise we'll describe the sub modules of the layers:

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

Storage Manager: is found inside of the data layer, it manages files and pages of the database.

System Control Manager: handles authentication and starts and stops `postgres` processes. **Traffic Cop:** handles the flow control of queries.

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

Backend: is a small module which stitches together the various modules to create the *PostgreSQL* server.

Developer Util: consists of code which end-users will probably not use, like the test suite.

Executor: executes the query plan (execution tree).

Include: consists of common include files shared amongst many modules, particularly those which use Util.

LibPQ: enables the client or user to query the RDBMS.

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

Parser: tokenizes and parses SQL queries.

Query Evaluation Engine: consists of submodules (Catalog, Command and Nodes) used to represent queries and meta-data about the databases.

4.1 Timeline of *PostgreSQL*

We examined the history of architectural changes of *PostgreSQL* by manually inspecting a few architecturally significant transactions. *HistODiff* flagged about 100 transactions that it considered to be architecturally significant; that is, they have either added or removed dependencies which have significantly increased or decreased the degree of coupling between subsystems.

We divided the history, consisting of a sequence of transactions, into three periods with an equal number of flagged changes. The time intervals of these periods varied considerably because the rate of occurrence of changes varied.

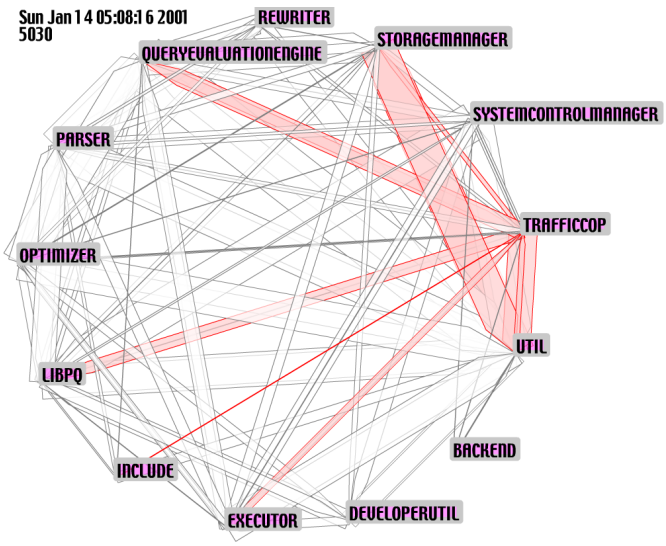

Figure 6. *PostgreSQL YARN* **Flip-book shot 5/6**

- 1996 to 1998: *PostgreSQL* was released as Open Source Software. Portability and reimplementation of features such as ODBC were included (see figures 1 and 2).
- 1998 to 2000: *PostgreSQL* was 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 maintained, as a reasonably mature system. Fewer features were added, more auditing and bug fixing occurred (see figures 6 and 8).

Figure 7 depicts the changes made to the system starting from 2001 to 2005; larger changes are highlighted. This was a notable period for *PostgreSQL* where large changes to the source code 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 included fixing flaws related to interrupt handling and critical sections.

5. Visual Story of PostgreSQL

There were many significant changes from 1996 to 2005; we have shown a few and have provided screen-shots of the revisions. These screen-shots help highlight the progression of architectural dependencies over time in *PostgreSQL*. As well, they highlight the architecturally significant changes that occurred over time.

The first notable change (figure 1), shows that dependencies changed between Parser, Query Evaluation Engine, Storage Manager, Traffic Cop and Utils. Investigation into the revisions revealed that this change was the reimplementation of the ODBC driver by Insight, and it was highlighted because of the doubling of dependencies between Parser and Utils.

The second notable change is from figure 2. We can see that the two corresponding dependencies that changed during this time are highlighted, namely those between the System Control Manager and Util. Although *YARN* does not tell the full story, the revisions around this time revealed that the embedded language PL/TCL was added. PL/TCL is an 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 (color-enhanced). By investigating the revisions, we found this was the addition of the PL/PgSQL embedded language: a language for stored procedures in *PostgreSQL* which wasn't TCL. It is important and noteworthy because the inclusion of the language crosscut multiple modules and introduced more coupling.

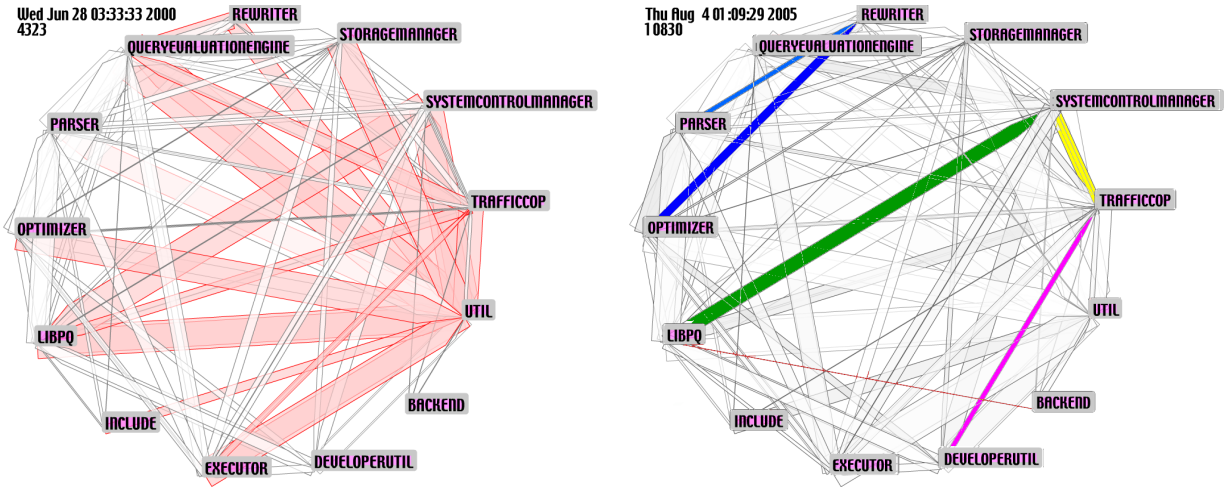

Figure 7. Important architectural changes done during the last 5 years

The fourth change (figure 5) we show is actually both an 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 Control Manager has been reduced. Most of the other changes seem to show an increase in coupling. Investigation of the related revisions revealed that a memory management submodule was rewritten. This change also included several bug fixes and removed some dead code.

From the fifth change (figure 6), we see new coupling from Parser and Optimizer to the Rewriter, Include to Traffic-Cop, Include to LibPQ and LibPQ to Executor. These changes were made to *PostgreSQL* due to the advice given by "white hat hackers" regarding security holes in the interrupt and issues with some of the critical sections.

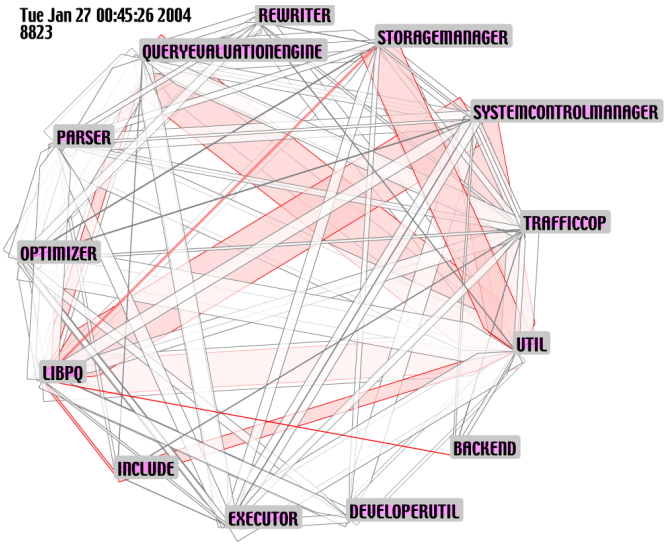

Figure 8. *PostgreSQL YARN* **Flip-book shot 6/6**

The sixth change (figure 8) was important due to the changes in dependencies that occurred between various modules and LibPQ and Util. The related revisions revealed that there was a change to the Win32 signal-handling code. This change made signal handling more portable via abstractions of the `kill` and `sigsetmask` syscalls.

6. Survey

In order to evaluate *YARN* , we created an informal user survey to see how intuitive the visualization was for users and if they were interested in using *YARN* . The survey could be answered over email or live over chat. Ten 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 worked intuitively. 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.

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 of the PostgreSQL architecture; one respondent said they were not surprised since they had experience with the architecture of the Linux kernel .

Some users said they would use the *YARN* Balls if they were available but were not actively going to create *YARN* Balls. Survey respondents suggested improvements such as making the visualization 3D or projected onto a sphere, or the use of other metrics. Some said more data should be displayed like the change log and a legend was needed. Some suggested tweaking parameters like color decay speed etc. One suggested coloring the edges to indicate which module was being depended on.

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 hierarchical navigation of sub modules. Different views of the architecture are planned, such as a source control view which shows which modules are coupled together per commit. Other work includes formally evaluating the use of animation for maintenance work.

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 include: an approach for animating 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 commit-level 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 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. Baecker and D. Sherman. Sorting out sorting. 30 minute colour sound film, 1981.

[4] D. Beyer and A. E. Hassan. Animated visualization of software history using evolution storyboards. In *Proceedings of WCRE 2006: International Conference on Reverse Engineering*, pages 199–208. IEEE Computer Society, October 2006.

[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. 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] R. Holt. Software architecture as a shared mental model. In *Proceedings of the ASERC Workhop on Software Architecture*, University of Alberta, Aug. 2002.

[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] C. Mesnage and M. Lanza. White coats: Web-visualization of evolving software in 3d. In *Proceedings of the 3rd International Workshop on Visualizing Software for Understanding and Analysis, VISSOFT 2005, September 25, 2005, Budapest, Hungary*, pages 40–45. IEEE Computer Society, 2005.

[13] 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.

[14] 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.

[15] A. Telea and L. Voinea. Interactive visual mechanisms for exploring source code evolution. In *VISSOFT*, pages 52–57. IEEE Computer Society, 2005.

[16] 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.