ChronoTwigger:

A Visual Analytics Tool for Understanding Source and Test Co-Evolution

Barrett Ens, Daniel Rea, Roiy Shpaner, Hadi Hemmati, James E. Young, Pourang Irani
Department of Computer Science, University of Manitoba
Winnipeg, Canada
{bens, daniel.rea, roiy, hemmati, young, irani}@cs.umanitoba.ca

***Abstract*— Applying visual analytics to large software systems can help users comprehend the wealth of information produced by source repository mining. One concept of interest is the co-evolution of test code with source code, or how source and test files develop together over time. For example, understanding how the testing pace compares to the development pace can help test managers gauge the effectiveness of their testing strategy. A useful concept that has yet to be effectively incorporated into a co-evolution visualization is co-change. Co-change is a quantity that identifies correlations between software artifacts, and we propose using this to organize our visualization in order to enrich the analysis of co-evolution. In this paper, we create, implement, and study an interactive visual analytics tool that displays source and test file changes over time (co-evolution) while grouping files that change together (co-change). Our new technique improves the analyst's ability to infer information about the software development process and its relationship to testing. We discuss the development of our system and the results of a small pilot study with three participants. Our findings show that our visualization can lead to inferences that are not easily made using other techniques alone.**

***Index Terms*— Co-evolution, co-change, mining software repositories, information visualization, temporal data visualization, 3D visualization, visual analytics.**

I. Introduction

The relative changes between pairs of software artifacts over a system's development is known as *co-evolution* and can be facilitated through the mining of software repository data. The co-evolution of source code and test files in software is of particular interest to members of industry and academia [1]. For example, a project manager may compare how many test files changed compared to source files at the beginning of a project to see if the development team was properly testing from an early stage. *Co-change* is another concept that has been introduced in previous work on mining software repositories for software evolution [2] and software visualization [3]; co-change is a metric that estimates the level of coupling between various software entities based on the assumption that logically related files tend to be included in the same source repository commit. We believe that co-change can improve the analysis of source and test co-evolution by revealing periods of coupling between these artifacts to analysts, allowing them to infer information about the software development process and its relationship to testing.

In this paper, we present ChronoTwigger (Fig. 1), an interactive tool for exploring and understanding the co-evolution of source and test files. Following the approach of recent visual analytics systems [4], we apply information-visualization techniques to assist the understanding of large data sets that can result from mining a source repository. Our visualization is created automatically from data we mine from Git repositories. Using known effective temporal visualization methods [5], we improve on current co-evolution visualizations [1] that organize files spatially by their commit ID [2]. Our new temporal visualization visually organizes source and test artifacts using Beyer's co-change energy minimization function [3], which groups artifacts spatially by their level of coupling.

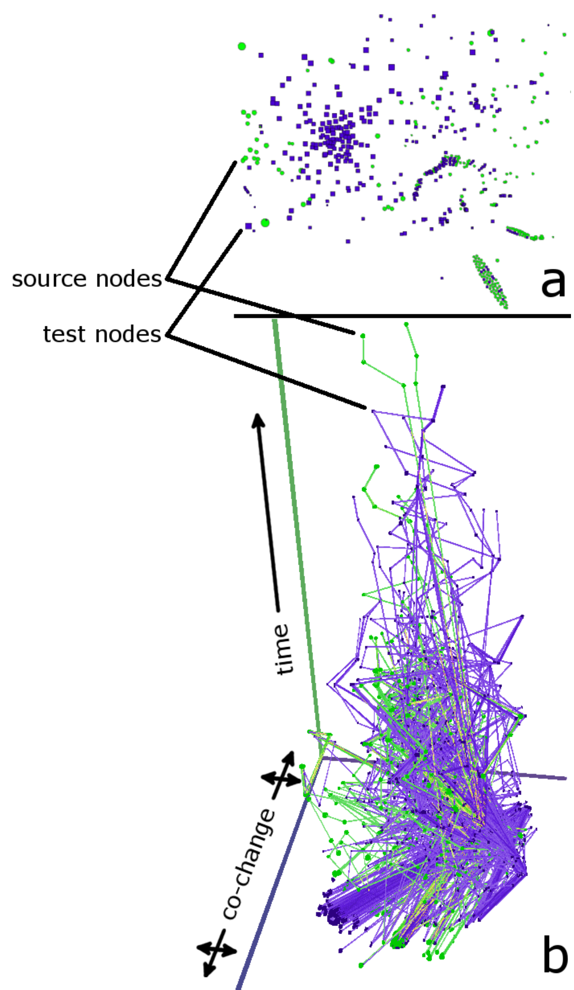

Fig. 1. Our ChronoTwigger visual analytics tools shows integrated 2D (a) and 3D (b) visualizations of mined repository data. The 3D visualization shows co-evolution of source and test files (green and violet nodes, respectively) over time on the vertical axis. Interaction is controlled via the 2D interface, which shows an overall co-change clustering [3].

For example, Beyer's visualization method groups files likely to be related closer to each other, while files that are probably unrelated are spatially far apart. However, unlike Beyer's visualization, which summarizes a complete project history in a single, static visualization, our visualization layers many time slices, allowing analysts to trace co-change over time.

Our visualization technique allows software analysts to identify items of interest that may not be readily visible with existing techniques. Some examples of such motivating patterns are shown in Fig. 2. For instance, existing temporal software visualizations (e.g. [1, 2]) may reveal the simultaneous creation of large groups of test or source files at the start of a project. However, our visualization of temporal co-change information provides additional details about subsequent coupling of between test and source files, as depicted in Fig. 2a. This may provide some evidence about the testing strategy (e.g., test driven development) and the extent to which it is used in a project. Temporal co-change information can also indicate instances of tight coupling among larger groups of loosely-coupled files (Fig. 2b) or draw attention to files that migrate between clusters (Fig. 2c).

Although these simple examples are shown in two dimensions, our 3D adaptation of Beyer's 2D co-change function is capable of handling data sets with much greater complexity. The user's comprehension is enhanced by her ability to view the 3D shape from different perspectives. To enable fine control over the viewing angle and the amount of data viewed at one time, we introduce features for interactive exploration of the software project in real-time. Using rotation and selection controls, along with temporal and artifact filters, analysts can gain insights about a project's co-evolution details by viewing any desired group of files over any span in the project's timeline. As a result, we derive the name ChronoTwigger from the Greek word for time, "chronos," and "twig," as our 3D visualization often resemble complicated branching structures such as twigs in a tree or bush.

To evaluate our system's ability to convey information to analysts, we conducted a small pilot study. We asked participants to explore software projects with ChronoTwigger and check if it allows them to answer questions such as "Can we determine periods of intensified development or testing?" or "Can we detect testing strategies such as test-driven development?" Participants were able to use ChronoTwigger's interactive features to find patterns in two open source projects. They were also able to clearly identify periods of concurrent activity in source and test files, as well as periods where testing activity tapered off or intensified at different points in the project. Many of the insights gained are either difficult or not possible to obtain using previous visualization tools alone.

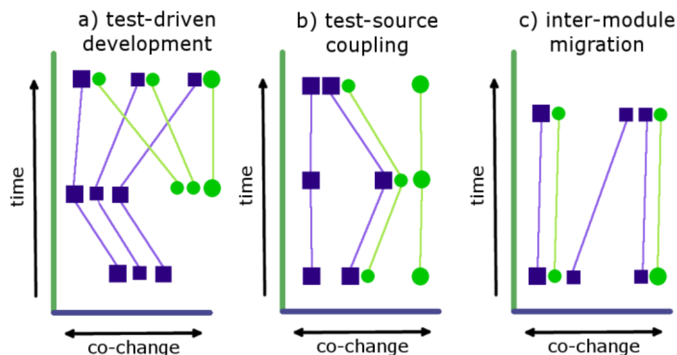

Fig. 2. Example cases where ChronoTwigger's amalgam of co-evolution and co-change provides information not readily available with existing software visualization: test files (violet squares) created during test-driven development subsequently couple with different source files (green circles) (a); tight coupling of source and test files among looser couplings (b); and migration of files between modules (c).

The result of our work is a visual-analytics solution for analyzing source and test file co-change and co-evolution. Our contributions include:

(1) a novel 3D visualization technique that displays the co-evolution of source and test files organized by temporal co-change. Our visual analytic tool also includes an extension of Beyer's system that explicitly differentiates source and test files and acts as a control for the linked 3D view
(2) an interactive prototype tool for both immersive 3D and desktop-based interfaces
(3) an initial, proof-of-concept user evaluation

II. RELATED WORK

We divide much of the related work into two primary areas of co-change and co-evolution. We flank these sections with a brief introduction of the overarching subject of software evolution and a closing discussion of temporal visualization.

A. Software Evolution

The study of software evolution is concerned with the detailed analysis of a software system's growth over time by leveraging the information contained in a source code repository. This helps project managers gain insight into, for example, the coupling of a project's subsystems, which can be used to improve software reliability and reduce development costs [6]. Researchers have also used repository data to understand the potential benefits and drawbacks of open and closed source development practices [7]. Others learned about a software team's efficiency by adding code documentation and email records to their analysis [8]. ChronoTwigger builds on these approaches as a visual analytics tool [9] by enabling the quick analysis of a software project's source and test file co-evolution by visualizing how potentially coupled files changed together (co-change) throughout the project history.

B. Co-change

Co-change is a measure of how often files change together in a project and can be used to extract logical relationships between code that are not apparent in the physical directory structure. It relies on the assumption that files committed together frequently are likely to be functionality dependent on each other, even if there is no explicit reference in the code itself. Co-change is known to be a reasonable heuristic for estimating the impact of code changes on an entire system [10], is correlated with the frequency of bugs in a system [11], and can be used to suggest files that may need to be changed to prevent bugs in the current commit [12]. A common theme in these works is that their results became stronger when people

assisted the analysis. Therefore, one of ChronoTwigger's intents is to present co-change information to users to assist analysis of a software system's co-evolution.

Co-change has previously been visualized to present module coupling in an intuitive way [13], but was focused on small software systems with a few files. Further work used an energy based clustering algorithm to produce an intuitive visualization of the relationships between software artifacts [3]. This work was later improved to an "animated" version [14] that showed change over short periods of time through a sequence of storyboards. The researchers chose to juxtapose static panels because the animation method makes analysis of long projects difficult as the user has to replay the animation many times. ChronoTwigger also shows co-change over time, but addresses the problem of analyzing long projects by using a static temporal visualization that enables the user to view an entire time series at once. We apply co-change visualization to the problem of co-evolution analysis, where we use its properties to help the viewer focus on relationships between test files and source files that were changed together frequently.

C. Co-evolution

Co-evolution is an idea that focuses on exploring how software and its associated tests evolve together. Since test and source code can change at different rates and times, observing their co-evolution can give insights into aspects how testing was used during development [1]. There have also been studies in co-evolution of other parts of a software system, such as source code, test, and build files [15].

Co-evolution can reveal aspects of a software project that help promote good development practices. In recent work, Hurdugaci and Zaidman [16] use co-evolution enforce test case code-coverage checks on commits to maintain test quality throughout development. In their work, co-evolution was derived by explicitly checking which tests ran on which code which they noted was time-consuming, while others have measured co-evolution using various association rules built on probabilities [1, 17]. These works highlight the benefits of understanding the co-evolution of a system, and so we target ChronoTwigger as a tool that will help developers analyze the test and source co-evolution their systems. We also propose that co-change would be an effective way to investigate co-evolution, as source and test files that are committed to a source repository together should be detectable by co-change.

Zaidman et al. [18] explored visualization for co-evolution based on a previous tool for software evolution by Van Rysselberghe and Demeyer [2], where files are organized by their date of creation in the project. By inspection of Zaidman et al.'s visualization, analysts can determine whether testing happens synchronously or in phases, locate areas where test-writing effort is increased, and identify testing strategies such as Test Driven Development. Our visualization will explore the use of newer visualization techniques, and extend previous co-evolution visualizations by organizing files by their relationships with each other as calculated by Beyer's co-change method [3]. This organization was lacking from previous work, and should help the user recognize additional aspects of co-evolution such as impact on test files after changes in source files because these files would be visually grouped together.

D. Using 3D to Visualize Time

Our work aims to reveal patterns of co-evolution by visualizing a project's temporal dimension. In visualization literature, there are two well-studied methods for representing time: One method is through animation and the second is to map time to one available spatial dimension [5]. An example of animations for showing different authors' influence on a project's development history is Caudwell's Gource [19]. Since animations require a span of time to watch, Beyer and Hassan [14], used storyboards as an alternative to animation to show important events in a project's co-change history.

ChronoTwigger is inspired by visual analytics systems [9] such as Jigsaw by Stasko et al. [4], which uses multiple interlinked views of a set of complex information to assist in analytical tasks. The analysis of a software project's co-evolution, with perhaps thousands of files modified over a series of thousands of commit events, is difficult to comprehend with a single visualization alone. For our project, we chose one of the views to be a 3D visualization to help us display co-change and temporal information alongside software repository data. 3D visualizations have previously been used to help understand large, complex software systems. Caserta et al. [20], for example, use a "software city" metaphor to create an intuitive graph showing relationships between subsystems. In our ChronoTwigger prototype, we use the third dimension to represent time. Known as a space-time cube, this well-used visualization technique displays time alongside two spatial dimensions, making it a good candidate for adding a time component to Beyer's 2D clustering visualization [3]. The space time cube originated in social science [21], and is often used for the visualization of geospatial data, though any temporal data with two dimensions can be used. Although the resulting 3D layout can increase visual complexity, a study by Kristensson et al. [22] provided empirical evidence that the space time cube can lead to faster analysis when considering complex questions.

III. System Design

To create our ChronoTwigger solution, we first looked at Beyer's work in CCVisu, a tool that takes file-relation (RSF) files as input and displays a visualization (Fig. 3). This visualization is the result of an energy-based clustering algorithm that groups code files, or other software artifacts, based on their relative amount of co-change; files that are changed together often are attracted to one another, and all others are repelled. (See Beyer's paper [14] for a full description of the clustering algorithm). We could see that this kind of visual representation would be beneficial in making conclusions about the co-evolution of source and test files. The main drawback of this visualization was the lack of information about the process of evolution throughout the project, limiting conclusions that can be drawn about development; CCVisu displays the final clustering based on co-change over the entire duration of the project, which gives a robust view of the software system but does not provide insight about specific

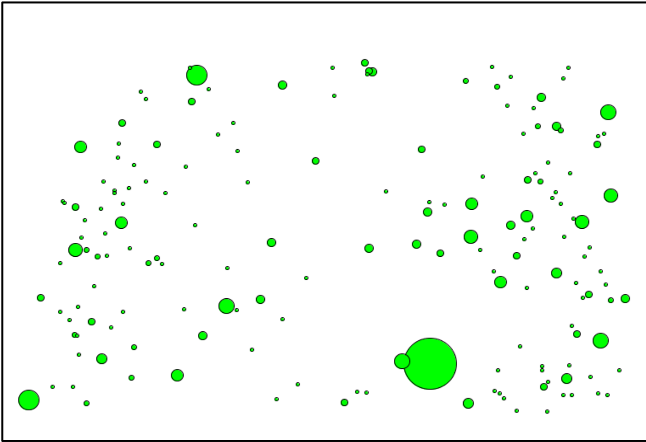

Fig. 3. Beyer's original CCVisu output [3]. Nodes closer to each other are likely to be related (changed together in commit logs). The size of the node shows its amount of change of the project's history (number of commits).

points in time. It also does not differentiate between source and test files.

We developed a 3D visualization that is based on Beyer's work, but spreads the project data over time in a space-time cube. In this way, we show the progression of file clustering and commit behaviour throughout the life of the system, or in particular time periods that are of important interest to the user. We chose a space-time cube as it is a well-researched way to show progression of data over time [22], and we hypothesized it could be useful if combined with the layout results from the CCVisu algorithm.

To promote user interactivity, we created a single interface to control both visualizations (improved 2D and 3D). We expanded the existing CCVisu interface with communications to the 3D application. This means that any command issued on the 2D interface will be transmitted to the 3D application, and create a simultaneous effect in that visualization as well. Due to the nature of the clustering algorithm, there is no direct correcpondance between the positions of nodes across the 2D and 3D visualizations, however the two views complement each other, offering different views on the same data.

We created a mining application to analyze the commit data of projects and create the necessary files which will be used to display the data. A diagram of our system architecture is shown in Fig. 4. We discuss each of the components in more detail in the next few sub-sections.

A. Mining Application

We created an application to mine open source Git repositories. The input is the output of the command `git log --name-status`, which returns each commit ID, the author of the commit, the time it took place, the commit comment, and all files in the commit and whether they were added, deleted, or modified. This output is automatically sorted by time.

We parsed this data into a format that CCVisu can understand: an RSF file. This is a general file format that is used to store relational information (such as a relational database). In this case, it is used to store the commits as a directed graph, where each edge connects a file with the

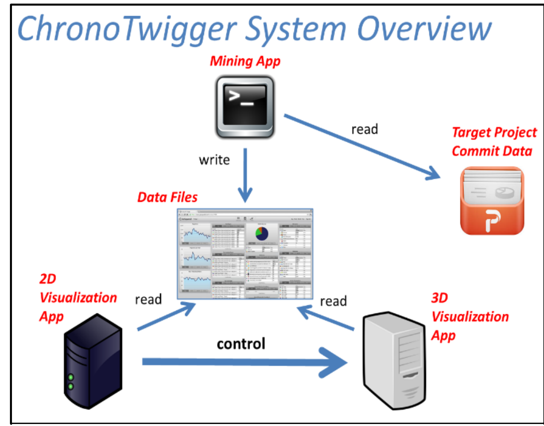

Fig. 4. ChronoTwigger system architecture.

commit it appears in, where the source of the edge is the commit node.

To display this data over time, we took the entire commit history and divided it into smaller time slices, grouping commits that occurred within the same slice. For proof of concept, we found that a granularity of eight equal slices adequate, although more slices would allow a more detailed analysis. Preprocessing could also be adapted to provide the user multiple levels of granularity; with the multiple resulting "layers", temporal relationships could be viewed at varying levels of detail using a temporal "zooming" functionality.

For each individual time slice, we applied Beyer's co-change minimization algorithm. However, because each slice was run without data from the previous commits, nodes common across slices (files included in multiple commits) had unpredictable and inconsistent mapping to spatial positions from one 2D slice to the next. This lack of common information across each time slice often resulted in the same node appearing in entirely different locations in different slices, making the temporal visualization difficult to interpret.

To maintain some consistency between the positions of nodes, we applied a sliding window to ensure that each adjacent slice contained a high number of common information. For each time slice, we enforced a substantial overlap with the previous window (Fig. 5). For proof of concept, we chose 75% overlap after trial-and-error, although the optimal overlap will require future investigation. We kept the size of the slices the same at one eighth of the commit history. The overlapping window results in the generation of many additional segments, providing a greater resolution for data exploration. This sliding window method has an effect analogous to interpolating between keyframes of an animation, smoothing the transition between them and reducing node movement due to different sets of files in each slice.

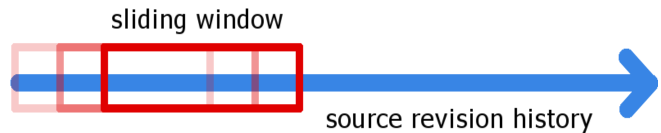

Fig. 5. Each time slice of the source revision history overlaps the previous slice by 75%. This sliding window method is necessary to create a set of change graphs that interpolate smoothly between consecutive states.

We ran each of the resulting RSF files through Beyer's algorithm. The output of the algorithm is a layout file that holds the final location of each node clustered by co-change, along with its degree, which represents the number of nodes that are connected to that node. The file also holds the name (including the file path) of each node, from which we can reliably determine whether the node corresponds to a test file (assuming the common practice of storing test files in explicitly named directories). This layout data is used by our 3D visualization, described in Section 3.3.

B. 2D Visualization

Once the input data is created by the mining application, we import it to the data visualization software. Our 2D visualization and control interface is a modified version of CCVisu [3]. In short, nodes in the resulting visualization are grouped by how often they are changed together in commits: nodes that are closer together are more likely to be logically coupled than nodes that are farther apart. Node size represents the total amount of change for that node throughout the project. The resulting visualization is a summary of co-change over the selected time period. The algorithm used to decide positions of nodes is the same as provided in CCVisu.

As noted previously, our aim was to produce a visualization that contrasts source and test files, enabling users to analyze co-evolution. To accomplish that we changed the color of test-related files in the project to violet, and changed their shape to squares instead of circles (Fig. 6). This simple but significant difference gives a much clearer view of the couplings and allows for immediate conclusions for the testing behaviour in the system (e.g., are source files and test files developed together? What are the logical test groupings in the system?).

We also added features to enable a closer look at a specific file's evolution with respect to files changed with it during its evolution. In our updated version the user can select a file node to show only the files that had changed with the target file. This change targets developer questions such as "Which test file should I modify after editing a certain source file?" and "Which source files should I investigate if a certain test fails?"

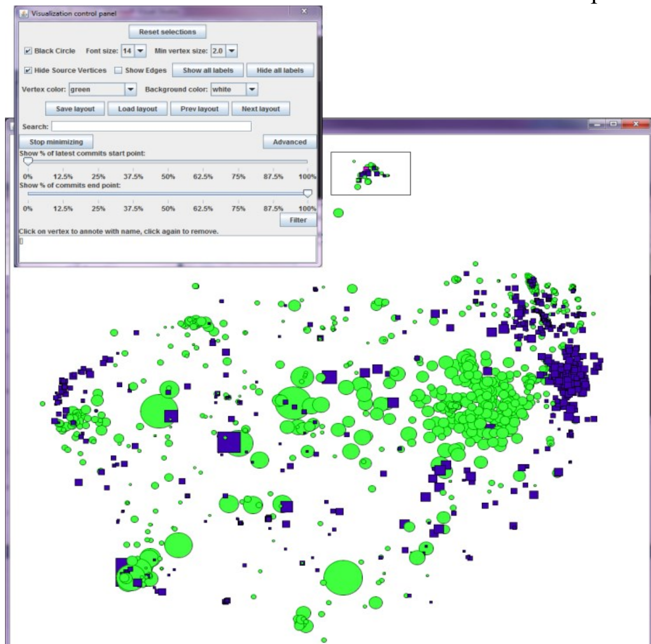

Fig. 6. Top: Control window display, controls both visualization simultaneously. Bottom: 2D visualization (violet squares are test-related files, green circles are source files).

Additionally, an important element for the project was the time aspect of the visualization. We wanted a way to have all visualizations display data for only a time period controlled by the user. We expanded the CCVisu control window to include sliders that control the start and end time for the commit data that will be displayed (Fig. 6). These controls enable the user to focus on the critical time points during development. When the user selects a time period, the 2D visualization shows the result of Beyer's algorithm using only the commits within that period.

The 3D and 2D interactions are synchronized over a network. We send all commands from the 2D interface directly to the 3D application, along with the IDs of nodes that should be displayed. This allows the 2D and 3D applications to display related data, and lets the user use both visualizations to analyze the data.

C. 3D Visualization

Our 3D visualization (Fig. 1b) uses a space-time cube representation to depict the co-change relationships between objects over time. Whereas the space-time cube is often used to represent geographical space, we instead use two spatial dimensions to represent the relative degree of co-change, precisely as is done in Beyer's static visualization (nodes that are likely to be logically coupled are drawn closer together). However, since we calculate the co-change function for each individual time slice (as described in Section III.A), changes in the degree of co-change between nodes over time are now visible. We use the vertical axis to represent time, with time flowing in the "up" direction. Node size represents the total change to the corresponding source or test file within a given time slice.

For a selected group of nodes and chosen span of time, the nodes are scaled to fill a 0.5 $m^2 \times 2$ m high volume for an immersive 3D environment. Users without such equipment can view it on a 2D display monitor, where the volume is scaled down to fill the display screen. The spatial dimensions are scaled such that the greatest distance between any two nodes spans the full cube width. Time slices are also scaled such that each slice is separated by a distance of $height/(n-1)$, where n is the number of available slices and *height* is the length of the time axis. Since one file object can exist in multiple layers, the corresponding entities are connected with line segments; each node is connected to the corresponding node (i.e. sharing the same file ID) in the previous layer. If a file does not undergo any change for a period of time, some levels may be skipped and the node is connected to the next counterpart found.

As mentioned in the previous subsection, interaction with the 3D visualization is done through the 2D visualization and control panel. Although interactivity would benefit from two-way communication, selection and control over 3D objects is a difficult problem and a topic of current research. Thus, we limit the control flow of our prototype to one direction (from the 2D

to the 3D visualization). Because there may be no obvious connection in the appearance the 2D and 3D structures, control mechanisms, such as “select node”, can also help users translate between the two views.

A summary of the control mechanisms available in our prototype is shown in Fig. 7. We give a detailed description of each as follows:

Select node – selecting a node on the 2D display (i.e. the magenta node in Fig. 7) causes the corresponding set of nodes and their connecting line to be highlighted (in magenta) in the 3D visualization.

Band select/zoom – a basic rectangle selection for selecting a set of nodes on the 2D display that causes the corresponding nodes to be shown on the 3D display. All other nodes are hidden and both 2D and 3D displays zoom to the selected set.

Show co-change – right-clicking on a node selects all nodes that are co-changed with that node at any point in the project history. The selected node is also highlighted in the 3D visualization.

Time filter – two sliders on the 2D control panel delineate the start and end points of the selected timespan. In the 2D display, any nodes that did not have any change events within this timespan are hidden. Meanwhile, the 3D visualization shows only those time slices that lie fully within the selected start and end points. These slices are then expanded to fill the entire height of the 3D volume.

D. Implementation Platform

Our 2D visualization and control panel were extended from CCVisu [3], which is written in Java and made freely available online by its creators. We modified the program into a network client which connects with a host program that updates the 3D visualization.

Our 3D visualization was implemented using the OpenGL framework. For view management and other utilities, we use additional frameworks: for the desktop implementation, we use the GLUT framework; for the immersive projected display, we use VR Juggler, which handles perspective mapping based on head-tracking input. The large display is a Visbox Viscube C2-HD. This consists of two pairs of projectors which display 4-metre wide stereoscopic renderings on a back-lit screen and on the floor. Standing on the projected floor while wearing the head-tracked 3D glasses provides a feeling of immersion, where the 3D visualization appears before the user. In this setting, 3D glasses do not impede the use of the 2D interface, which we run on a handheld tablet computer. While such immersive environments may be beyond reach of small companies, our system could easily be adapted for use with 3D monitors or wearable virtual reality systems, which are becoming available at low cost. Nonetheless, our 3D visualization does not require stereoscopic viewing equipment of any kind; ChronoTwigger can be displayed on a standard 2D monitor, in which case a sense of the 3D structure can be revealed by controlling its rotation.

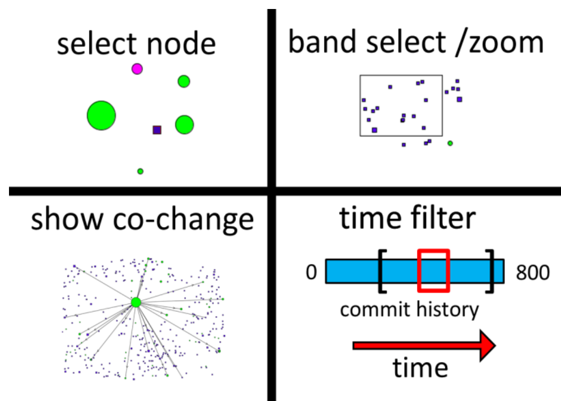

Fig. 7. A summary of the interactive features that allow control of the 3D visualization from the 2D display.

E. Example Visualizations

To demonstrate how ChronoTwigger works in practice, we compared two open source software projects. The first is Checkstyle, a tool for evaluating Java coding standards, written in Java. It has over 2500 files and a similar number of commits from 2001 to 2013. For the second example, we chose the mailnews component of the Mozilla project, which has nearly 8000 files and about 13000 commits from 2003 to 2012.

Fig. 8 shows all nodes from both projects after 100 iterations of Beyer’s energy function (the default value). Our modification for highlighting co-evolution shows some clear groupings of source and test files in the Checkstyle project (Fig. 8a). It appears there is strong coupling between a large group of source files on the bottom left side and a close but separate group of test files on the bottom right. The Mozilla project (Fig. 8b) has much less visible structure, which suggests lower cohesion of sub-groups and greater overall coupling throughout sections of the project. However, we believe this may be due to the large size of the project (approximately four times the number of files and commits from Checkstyle), which may require additional iterations of Beyer’s minimization algorithm to define a visible structure.

We look at some sub-groups of the projects’ files. Fig. 9 shows relatively large selections from both projects. In the 3D visualization, we again see more structure in the Checkstyle group (Fig. 9a) than in the Mozilla selection (Fig. 9b). However, we can see some immediate differences between the timelines of the two selections. In the Checkstyle project, there is a significant amount of development in the early stage of the selected files, which tapers off as the project progresses (the

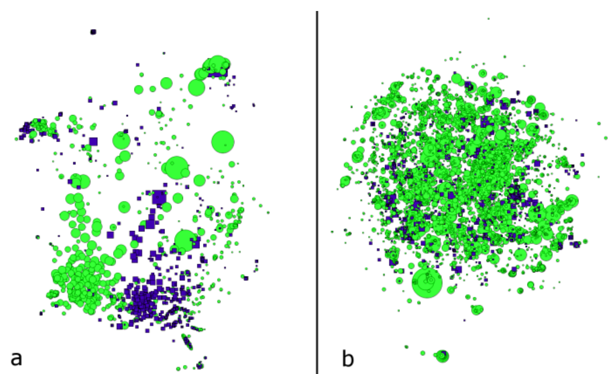

Fig. 8. All files from the Checkstyle (a) and Mozilla (b) projects as they appear in our modified 2D visualization, each after 100 iterations.

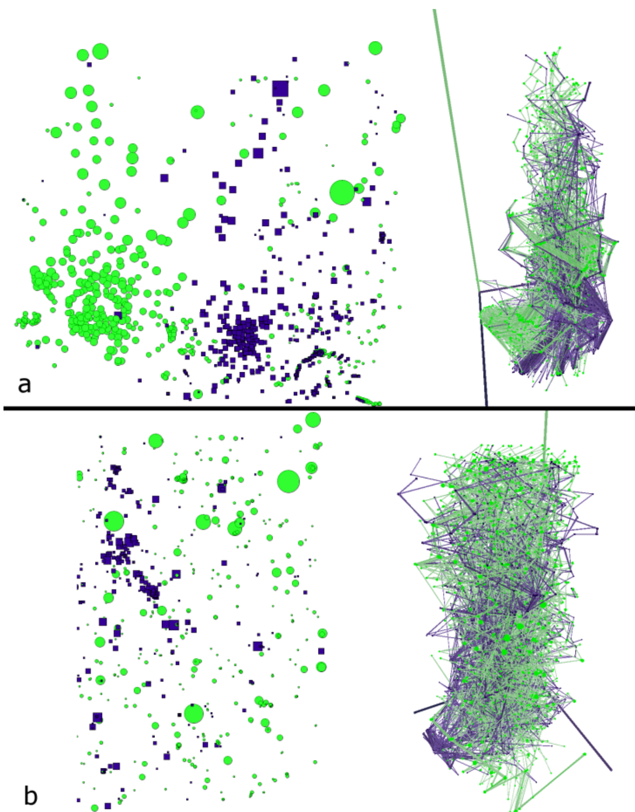

Fig. 9. A selection of a relatively large group of files from both the Checkstyle (a) and Mozilla (b) projects.

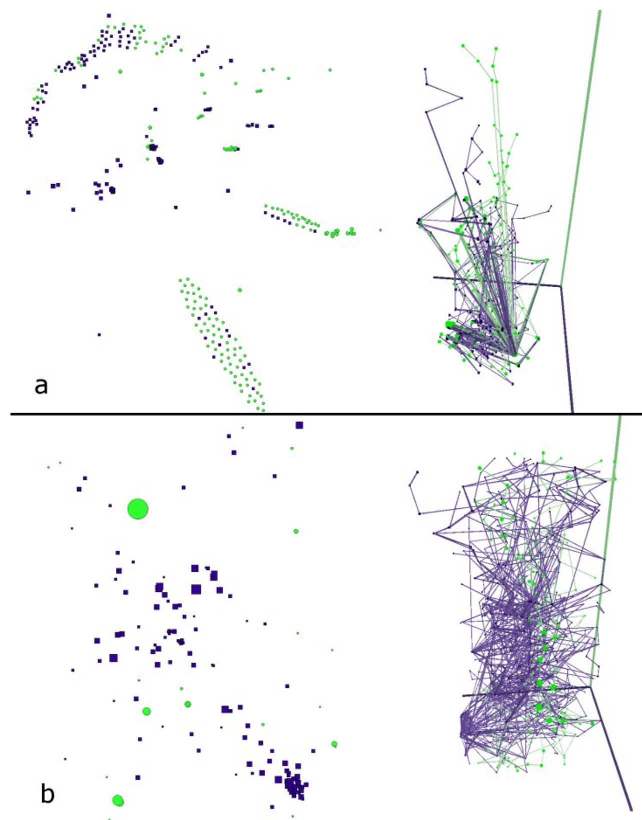

Fig. 10. A selection of a smaller group of files from both the Checkstyle (a) and Mozilla (b) projects.

majority of nodes disappear near the upper end of the timeline). In the Mozilla selection, on the other hand, development seems to remain fairly constant throughout, possibly with major refactoring near the middle, indicated by the larger source nodes in the central region of the 3D view. In the Checkstyle project, we can also see several thick strands in the 3D view, suggesting closely coupled files. No similar structure is visible in the Mozilla example. One common feature between both projects is the large number of test files created at the start of the project (bottom of the 3D views).

Our visualization also allows us to see features not visible in the 2D view alone. For example ChronoTwigger allows us to see how the clusters of source and test files evolve over time; while the 2D view shows a tight cluster of test nodes in both projects, the 3D view reveals that these couplings are not static throughout the project: in both projects we can see a group of violet test nodes diverge from a cluster near the bottom of the timeline, but they become mixed with the source files as development continues. This indicates that many of the test files were created together at the beginning of the project. These groups of test files disperse over time as they migrate closer to their (presumably) associated source files, indicating development focused on single features or bugs. This pattern cannot be extrapolated from the 2D visualization alone.

In Fig. 10 we see a smaller selection of files from each project (Checkstyle, Fig. 10a and Mozilla, Fig. 10b). Once again, a stronger organization is apparent in the Checkstyle project than in Mozilla. In the 2D visualization of Checkstyle, we see several clear subgroupings of source and related test files. As in the previous large selection, the 3D view tells us some additional information about the evolution of these groups that is not apparent in the 2D visualization: there is a tight grouping at the start of the timeline, as most of the files seem to have been created together. In later revisions, however, this large group diverges into several smaller tight groups, likely corresponding to the groups visible in the 2D graph. Again, we see that development tapers off at the end, where only a few nodes remain in the change logs.

In the Mozilla selection, we can see one large source node that was changed with many other files. The 3D view allows us to follow this file's history (the large green nodes along the center of the 3D view). Over time, we see many other source and test files, pulled repeatedly towards, and then away from, this file. Likely, there is more to the story, as these motions are affected by other files out of view. However, we can see that a majority of these concurrent changes happened in the first half of the project where the nodes are largest (like the 2D visualization, node size in the 3D visualization is proportional to the number of files changed with that node).

IV. RESEARCH QUESTIONS

We hypothesize that representing co-change (grouping source and test files that change together) in a clear co-evolution (temporal relations between source and test files) visualization will provide important information about the

software development process to the viewer. In particular, we ask the following research questions:

RQ1: Can we identify periods of intensified testing activity during the history of a project/subsystem?

RQ2: Can we infer testing patterns and determine how these evolved during the project's development?

V. EVALUATION

We performed a pilot user study with three participants who each used our system for approximately 30 minutes. These participants are proficient software developers (at minimum at computer science graduate student level); two males and one female. The 2D visualization was displayed on a tablet PC and the 3D visualization was displayed in an immersive 3D display. The system under scrutiny was the open source project Checkstyle, which has been used in previous works, including Zaidman et al.'s co-evolution visualization [18].

The study was performed in a workshop style; we explained the meaning of the 2D and 3D visualizations, and then asked participants to investigate any part of the software they wished while thinking out loud. We took notes on their insights, and present some qualitative findings below.

VI. RESULTS

RQ1: Can we identify periods of intensified testing activity during the history of a project/subsystem?

In Fig. 11a, a participant is inspecting a large subsystem of Checkstyle by zooming in on a large cluster of source and test files he noticed in the 2D visualization. Upon inspection in the 3D portion of ChronoTwigger, he noticed the large cluster of nodes at the beginning of the project, thinning out towards the top. He noted that testing and source changes occurred about uniformly throughout this intense development, but towards the end of the project where there is less development, test node changes outweigh the source file changes. He stipulated that this implies a significant testing phase at the end of the project, at least for this part of Checkstyle.

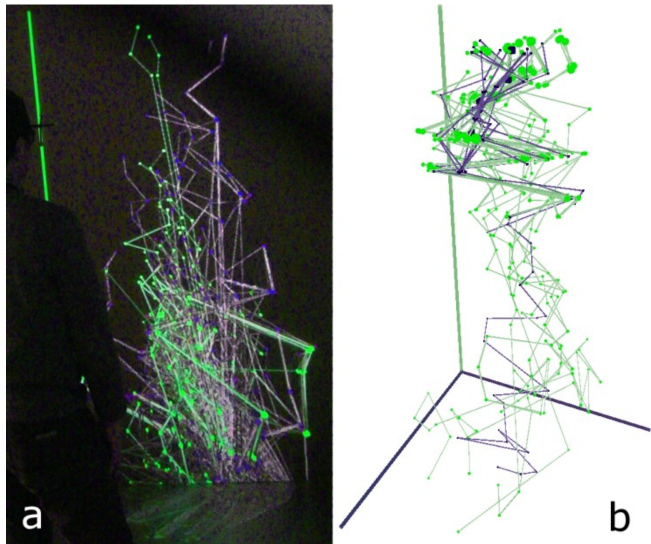

Fig. 11. A participant inspects a section of Checkstyle in an immersive 3D display (a). A group of tightly coupled, highly related files (b). Note how the lateral movements (co-change) match between source and test files, especially towards the top of the timeline. This implies the files were often being changed together, as they stick together despite moving in the co-change dimension, likely due to co-changes with files off-screen.

Our users picked out areas with unbalanced amounts of violet test nodes and green source nodes. Due to the co-change clustering related files together, they were able to also notice such periods on a smaller scale: in large projects, teams in charge of certain features may test better than other teams. This information would be available to project managers when they inspect projects with ChronoTwigger.

In Zaidman's temporal co-evolution visualization [1], this question can be clearly answered on a project-wide scale. However, Zaidman's visualization organizes files by ID, grouping modules as they are added throughout a project. This organization by file ID can make it extremely difficult to find relationships between closely coupled files that are added at different times (e.g. a source file and corresponding test file). ChronoTwigger makes answering this question easier on a per-module basis because of the structured nature of the visualization: as in Beyer's visualization, related nodes are spatially grouped, allowing the observer to easily identify modules. However, such a question cannot be answered by Beyer's co-change alone, due to its non-temporal nature.

RQ2: Can we infer testing patterns and determine how these evolved during the project's development?

In another inspection, one participant was inspecting Fig. 11b. The participant suggested this was, perhaps, a stub, being just a small data generation program (just a few files) that was later fully implemented about two-thirds into the project (as the related file count increases, shown by the increased node count). He also noted the continual testing visible throughout this subsystem's development, with a proportional increase of source and test files throughout the structure; which suggests an agile-like development methodology. He also noticed the proportion of test to source files, suggesting the test files likely hold tests corresponding to many source files: at the early phase of the project, where there are less files, he observed that the test file would move together with one source file at one point, and then move parallel to another source file at a different level, indicating that the test file covers multiple source files and may need to be refactored into multiple test files. While this conclusion is noticeable in Beyer's visualization (many source files by a single test file), ChronoTwigger makes such observations easier by explicitly marking test files in the 2D visualization, and more informative by including the temporal dimension which allowed our user to notice at what times (which commits) the test node moved closer to different source nodes, possibly aiding refactoring.

A different testing pattern was seen in Fig. 12. Upon inspection of the 2D visualization (Fig. 12c), it seemed the source and test files were related in some way, though the two types of files were clearly separated. The participant inspected each group separately (Fig. 12a, b). He noted that a similar structure appears in both source and test groups over time: a compact, strong branch at the beginning of the project moving

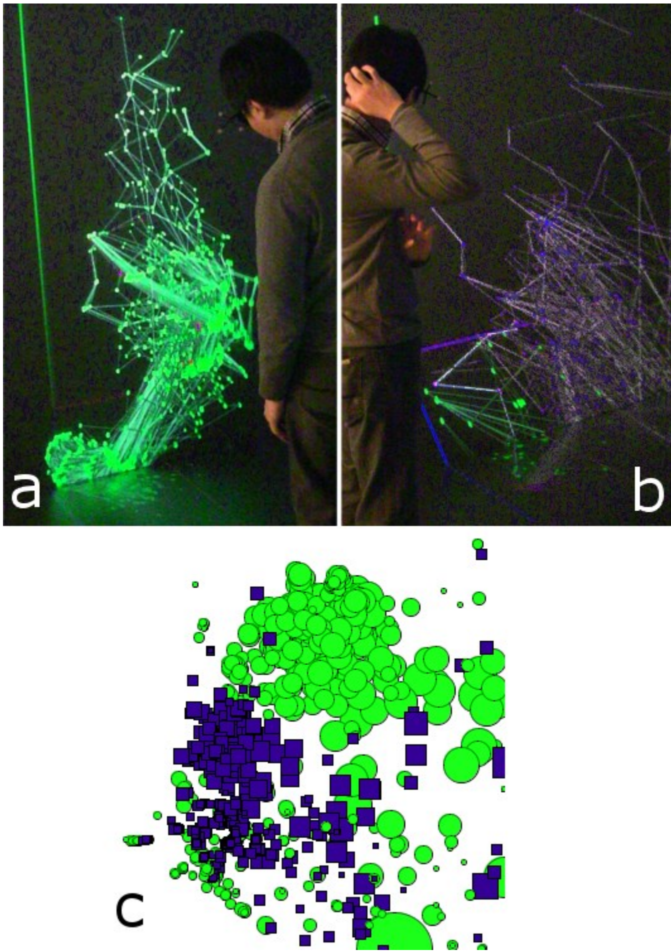

Fig. 12. A participant individually inspects large groups of source files (a) and test files (b). Both groups are shown together in the 2D view (c).

to a more dispersed, chaotic looking mid-project section, and ending in the thinner and spread out end of the project. The participant observed the beginning is probably due to many files being created at once, with the middle having different features being developed at different times by different authors, ending with smaller localized bug-fixes at the end. The participant concluded that it was possibly a test-driven development methodology due to the strong parallels in shape of both source and test nodes. They thought the source and test were separated possibly because the authors developed the tests and committed them before developing the appropriate source code.

Our users made clear deductions about general practices in project development styles, however we cannot confirm such speculations without detailed analysis of the source repository. For example, one participant deduced that test-driven development was used in the code visualized in Fig. 13, but also suggested that it was difficult to be sure: it could be agile development with a test-last methodology as well. The participant also noted it is hard to compare large groups of files on a detailed time schedule. It is important to restate that our 3D visualizations shows groups of commits (commits occurring in the same time slice), not individual commits, blurring the exact time of change. However, since, in our examples, source and test files appear together over long periods of time, we can say with relative certainty that this project mostly tested alongside source development.

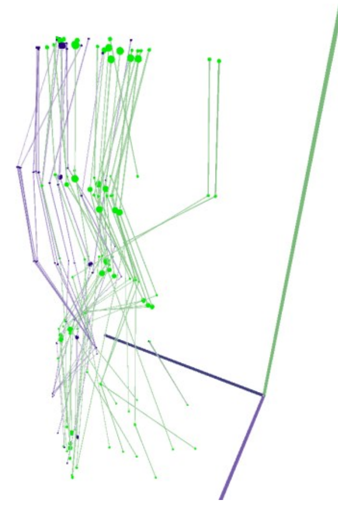

Fig. 13. A view of group of related source and test files, with some files branching off. This was a time filtered result, showing only the first 25% of commits containing these files.

As with the previous question, the organization by file ID in the visualization by Zaidman et al. makes it difficult to identify related files. Whereas this visualization can show when tests and source were written in relation to each other [1], it is difficult to tell how the relationship between source and test nodes changes over time. However, with ChronoTwigger's use of Beyer's co-change function, we can easily identify points where nodes converge or diverge during the project timeline. Again, Beyer's visualization shows only the sum of changes over the entire project, while ChronoTwigger's temporal nature allows Co-change to be observed over time.

VII. LIMITATIONS

While our participants used ChronoTwigger to infer properties of a software system's development, there are many factors that still need to be explored. For example, we noticed our visualization of Mozilla's mailnews component does not have as much structure as Checkstyle, suggesting that different sized projects (mailnews was about four times larger) may require more iterations of Beyer's algorithm to converge. In addition, the number of time slices for our 3D visualization (we picked eight) may affect what conclusions can be drawn from the visualization: more time slices would allow more detailed temporal analysis, but may impede the observer because the visualization would become cluttered. Additionally, our sliding window overlap method has a strong effect on the Beyer's layout result of each slice due to the inclusion or exclusion of files. Foresighted graph drawing techniques [23] may help mitigate this issue by initializing node positions to correspond with those in the prior layer. Likewise, the ideal amount of overlap between time slices requires further study. The overall user experience with our 3D visualization will benefit from additional features (e.g. text labels) and by allowing interaction

directly with the 3D structure as well as through the 2D view. We also require a larger user study to generalize our results. These studies should ideally include actual project managers analyzing their own projects with ChronoTwigger, to provide expert insight into the benefits and drawbacks of our system in real-world situations.

VIII. CONCLUSION

We have created a fully functional visual analytics system with interlinked 2D and 3D display views of mined Git repository data. Building on the work of Beyer [3] and Zaidman [1], we provide an interactive tool that can help find insights that are difficult to observer with previous tools alone because ChronoTwigger shows co-change as a function of time and allows users to filter by time or by connectivity of file artifacts. We ran a pilot user study to understand the potential benefits of our system and found that ChronoTwigger can help to:

(1) identify periods of intensified test or source development
(2) understand co-evolution at the level of logical subsystems
(3) understand aspects of a software project's general development style

While further, more rigorous user studies remain as important future work, we conclude that ChronoTwigger is a step forward in the visualization of co-evolution and in its current state can already show useful information to project managers and test engineers.

REFERENCES

[1] A. Zaidman, B. Van Rompaey, S. Demeyer , A. van Deursen, "Mining software repositories to study co-evolution of production & test code," In Proc. of Software Testing, Verification, and Validation, 2008, pp. 220-229.

[2] F. Van Rysselberghe and S. Demeyer, "Studying software evolution information by visualizing the change history," In Proc. of Software Maintenance, 2004, pp. 328-337.

[3] D. Beyer, "Co-change visualization," In Proc. of Software Maintenance (ICSM '05), Industrial and Tool Volume, 2005, pp. 89-92.

[4] J. Stasko, C. Görg and Z. Liu, "Jigsaw: supporting investigative analysis through interactive visualization," In Proc. of Visual Analytics Science and Technology (VAST '07), 2007, pp. 131-138.

[5] G. Andrienko, N. Andrienko, H. Schumann, C. Tominski, U. Demsar, D. Dransch, J. Dykes, S. Fabrikant, M. Jern and M.-J. Kraak, "Space and time," in Mastering the Information Age: Solving Problems with Visual Analytics, D. Keim, J. Kohlhammer, G. Ellis and F. Mansmann, Eds. Goslar, Germany: Eurographics, 2010, pp. 57-86.

[7] M. Godfrey and Q. Tu, "Evolution in open source software: A case study," In Proc. of Software Maintenance, 2000, pp. 131-142.

[8] D. German, "Using software trails to rebuild the evolution of software," Journal of Software Maintenance and Evolution, vol. 16, no. 6, 2004, pp. 367-384.

[9] D. Keim, J. Kohlhammer, G. Ellis and F. Mansmann, Mastering the Information Age: Solving Problems with Visual Analytics, Goslar, Germany: Eurographics, 2010.

[10] A. E. Hassan and R. C. Holt, "Predicting change propagation in software systems," In Proc. of Software Maintenance, 2004, pp.284-293.

[11] J. S. Shirabad, T. C. Lethbridge and S. Matwin, "Mining the maintenance history of a legacy software system," In Proc. of Software Mantenance, 2003, pp. 95-104.

[12] T. Zimmermann, A. Zeller, P. Weissgerber and S. Diehl, "Mining version histories to guide software changes," Transactions on Software Engineering, vol. 31, no. 6, 2005, pp. 429-445.

[13] N. Hanakawa, "Visualization for software evolution based on logical coupling and module coupling," In Proc. of Software Engineering, 2007, pp. 214-221.

[14] D. Beyer and A. Hassan, "Animated visualization of software history using evolution storyboards," In. Proc. of Working Coference on Reverse Engineering, 2006, pp. 199-210.

[15] S. McIntosh, B. Adams, T. H. D. Nguyen, Y. Kamei and A. E. Hassan, "An empirical study of build maintenance effort," In Proc. of Software Engineering, 2011, pp. 141-150.

[16] V. Hurdugaci and A. Zaidman, "Aiding software developers to maintain developer tests," In Proc. of Software Maintenance and Reengineering (CSMR '12), 2012, pp, 11-20.

[17] Z. Lubsen, A. Zaidman, M. Pinzger, M. W. Godfrey and J. Whitehead, "Studying co-evolution of production and test code using association rule mining," In Proc. of Mining Software Repositories (MSR '09), 2009, pp. 151-154.

[18] A. Zaidman, B. Van Rompaey, A. van Deursen and S. Demeyer, "Studying the co-evolution of production and test code in open source and industrial developer test processes through repository mining," In Proc. of Empirical Software Engineering, vol. 16, no. 3, 2011, pp. 325-364.

[19] A. H. Caudwell, "Gource: visualizing software version control history," In Object Oriented Programming Systems Languages and Applications Companion (SPLASH '10), 2010, pp. 73-74.

[20] P. Caserta, O. Zendra and D. Bodenes, "3D hierarchical edge bundles to visualize relations in a software city metaphor," In Proc. of Visualizing Software for Understanding and Analysis (VISSOFT '11), 2011, 1-8.

[21] T. Hägerstrand, "What about people in regional science," Papers in Regional Science, vol. 24, no. 1, 1970, pp. 7-24.

[22] P. O. Kristensson, N. Dahlback, D. Anundi, M. Bjornstad, H. Gillberg, J. Haraldsson, I. Martensson, M. Nordvall and J. Stahl, "An evaluation of space time cube representation of spatiotemporal patterns," Visualization and Computer Graphics, vol. 15, no. 4, pp. 696-702, 2009.

[23] S. Diehl and C. Görg. "Graphs, they are changing," In Proc. of Graph Drawing (GD '02), 2002, pp. 23-30.