Vision, Modeling, and Visualization (2012)
M. Goesele, T. Grosch, B. Preim, H. Theisel, and K. Toennies (Eds.)

# Visualizing Dynamic Call Graphs

M. Burch, C. Müller, G. Reina, H. Schmauder, M. Greis and D. Weiskopf

Visualization Research Center, University of Stuttgart, Germany

**Abstract**

*Visualizing time-varying call graphs is challenging due to vast amounts of data at many dimensions to be displayed: Hierarchically organized vertices with attributes, directed or undirected edges with weights, and time. In this paper, we introduce a novel overview representation that shows dynamic graphs as a timeline- and pixel-based aggregated view targeting the preservation of a viewer's mental map by encoding the time-varying data into a static diagram. This view allows comparisons of dynamic call graphs on different levels of hierarchical granularity. Our data extraction and visualization system uses this overview as a starting point for further investigations by applying existing dynamic graph visualization techniques that show the graph structures and properties more clearly. These more task-specific visualizations show the dynamic graph data from different perspectives such as curved node-link diagrams or glyph-based representations combined by linking and brushing. Intermediate analysis steps can be stored and rebuilt at any time by using corresponding thumbnail representations.*

Categories and Subject Descriptors (according to ACM CCS): E.1 [Data]: Data Structures—Graphs and Networks

## 1. Introduction

The need for visualizing and analyzing time-varying graphs in general is supported by various application domains. In social networking, software development, and bioinformatics, e.g., researchers are typically confronted with vast amounts of graph data that is changing over time and where the vertices can be hierarchically organized.

An intuitive concept to represent dynamic graphs is by a natural time-to-time mapping, i.e. by an animated sequence that shows each graph as a node-link diagram and smoothly transforms it into the next one in the sequence [FT08,DG02]. The problem that we are facing by using this visual metaphor is the algorithmic complexity that is needed to produce aesthetically pleasing graph layouts with respect to good dynamic stability. This again supports viewers to preserve their mental map in the animation and hence, reduces cognitive efforts. Animated node-link diagrams are problematic for the application of interactive features and an attachment of an additional hierarchical organization of the vertices. Furthermore, from a perceptual point of view, it is hard to compare graph patterns over longer time intervals and on different levels of hierarchical granularity.

Another drawback from which node-link diagrams suffer, in particular, is the problem of visual clutter [RLMJ05] caused by many link crossings. Layout algorithms are needed to produce readable and understandable graph layouts. If the graph becomes dense, node-link diagrams are not considered the medium of choice for displaying graphs [GFC04] and matrix representations might be used.

For this reason, we introduce a timeline- and pixel-based overview technique in our dynamic call graph extraction and visualization system that has several benefits:

- The pixel-based representation can be used in an aggregated form that allows good scalability in both vertex and time dimension.
- The algorithmic complexity is reduced compared to an animated node-link diagram.
- It allows graph comparisons over longer time intervals in a static diagram.
- Interaction techniques can be applied easily.
- A hierarchical organization of the vertices can be attached in an aligned way.
- The static pixel-based diagram serves as an overview and gives the viewer a starting point to inspect the displayed data in more detail.

In our system, we rely on the Visual Information Seeking Mantra [Shn96]: Overview first, zoom and filter, then details on demand. An analyst is supported by various existing visualization techniques for dynamic graphs, whereas the single views can be combined by linking and brushing.

© The Eurographics Association 2012.

delivered by EUROGRAPHICS DIGITAL LIBRARY www.eg.org diglib.eg.org

DOI: 10.2312/PE/VMV/VMV12/207-214

Furthermore, we allow undo and redo functions. Snapshots of intermediate analysis steps can be stored and are easily indicated by thumbnail representations.

Our system is divided into two separate components: data extraction and data visualization. Once data is extracted, it can be visually analyzed and explored while another data source is extracted simultaneously. The usefulness of our dynamic graph visualization system is illustrated by means of dynamic call graphs extracted from software archives. Understanding call relations combined with the hierarchical organization of complex software systems can give a software developer many insights and can help uncover problematic inter-component dependencies. The additional time-series information can tell when such a specific code coupling occurred or for how long it existed.

## 2. Related Work

Although the field of graph drawing and graph visualization was researched in the past and many graph visualization tools, techniques, and algorithms for static graph data have been developed, less progress can be seen in the domain of dynamic graph visualization. A recent approach has been developed by Burch et al. [BVB*11] known as parallel edge splatting and a more scalable variant using the concept of Rapid Serial Visual Presentation [BBV*12]. Each graph of a sequence is visually encoded in a small vertical stripe. The vertex set is copied and each copy is placed on the vertical boundings of the stripe. All vertices follow the same order which allows one to put the corresponding edges as straight links in a left-to-right reading direction. The edge coverage information is used to compute a density field that makes the link structure visible again. In the work of Brandes and Corman [BC03], a layered approach is used for showing the dynamics of a network in a stacked 3D node-link diagram that suffers from occlusion problems.

The aforementioned concepts are based on the node-link visual metaphor that typically suffers from visual clutter [RLMJ05] when the graphs become denser. To mitigate this situation, matrix-like representations have been developed for static graphs that avoid visual clutter but are not well suited for solving path-related tasks [GFC04]. These matrix representations have also been extended to visually encode dynamic graphs [BBD08]. NodeTrix [HFM07] illustrates a hybrid representation combining the benefits of both worlds, i.e. matrix and node-link diagrams, but it is only implemented for visualizing static graphs.

Apart from techniques using small multiples or stacking, graph animation is also used as an intuitive concept and as a natural time-to-time mapping instead of a time-to-space mapping to show the evolution of a graph over time. Online approaches [FT08], which do not know the whole graph sequence before the layout phase and offline approaches [DG02], which can take the entire graph sequence into account for a good layout, have been designed. The general problem for such layout strategies is to keep the layout as stable as possible over the evolution of the graph, which is referred to as dynamic stability. This again supports viewers to preserve their mental map of the dynamic graph [MELS95] and to reduce their cognitive efforts. Animated node-link diagrams still suffer from many problems [TMB02] even if the graph is laid out with respect to the formerly mentioned criteria. In the work of Robertson et al. [RFF*08] the negative effects of animation in trend visualization are investigated that are also of interest for dynamic graph visualization.

In this paper, we display the dynamic graph data in a static matrix-like diagram. To start a data exploration process a pixel-based overview representation is first provided with the further goal to scale in both vertex and time dimensions by using aggregation and overplotting. Edge metrics are visually encoded by color-coded timeline-based pixel representations. We base our data exploration on the Visual Information Seeking Mantra [Shn96] and hence, have to provide a rough overview of the data first.

The usefulness of the novel approach is illustrated by means of dynamic call graphs extracted from software archives. A lot of data sleeps unused in software archives and consequently, various analysis and visualization tools were developed over the years focusing on the processing of different types of data: the structure, behavior, and evolution of software [Die07]. SeeSoft [ESS92] is a software visualization tool from the very first beginning focusing on line-oriented and metric-based visualization of source code, but the visualization of dynamic call graphs was not investigated by this pixel-based representation. The same holds for CVSScan [VTvW05], which uses line-oriented techniques combined with several metrics to show the evolving source code and the involved developers.

Ogawa and Ma used an animated version to represent the dynamic file-developer relationships in their codeswarm visualization [OM09]. In recent research, they have proposed evolution storylines [OM10], a static diagram with crossing color-coded timelines that show the developer activity. Especially for the visualization of call graph data, the Gevol system [CKJ*03] was developed showing the changing graph structure and the developer activity by color coding and fade-out effects. However, it lacks overview for exploring the dynamic graph data on a comparison-based strategy because it is based on animation, although the vertex positions remain fixed over time.

Although some dynamic graph visualization tools exist, there is no scalable overview representation to this end that is able to show large dynamic weighted compound digraphs in a static diagram. This work is a step toward closing this gap in dynamic graph visualization.

© The Eurographics Association 2012.

## 3. Preprocessor

The visualization system is composed of two components. First, an analysis session can be started in which dynamic graphs and statistical as well as meta data about a user-defined project are extracted. Second, an already analyzed project can be selected and the extracted and preprocessed data can be visually represented to support explorative tasks.

### 3.1. Data Extraction

The first phase of data extraction is integrated into our visualization system by means of a plugin that allows for adding support for new programming languages and types of graphs or metrics. Currently, our tool has data extraction modules based on the *Phoenix* framework from Microsoft Research [Mic]. *Phoenix* is an extensible system for reading and writing native *Windows* binaries and *.NET* assemblies presented in the form of a manipulable intermediate representation that can be accessed programmatically. We use the *Phoenix* SDK to implement analysis plugins for C#.

When starting a new analysis with the system, the user must first select a repository plugin, which allows the system to retrieve a single revision from a source code repository. To this end we support the analysis of remote and local Subversion archives for which the user must specify the connection information, i. e. path and user credentials. By providing plugins for additional version archives other than Subversion, the system is easily extensible. Furthermore, the extraction can be limited to a specific range of revisions. Otherwise, the extraction process starts at the very first revision and ends at the head revision.

The analysis is performed by incrementally retrieving and compiling each of the revisions and storing the results to the user-defined location of the analysis project. While the analysis is running, the currently performed activity and the overall progress with respect to the selected range of revisions are displayed. At any time during the preprocessing step, the operation can be canceled and resumed later.

As our call graph generator uses the *Phoenix* SDK to extract call graphs for all methods and an inheritance graph for all classes in each revision, we need a *.NET* assembly in its binary form, which is the reason for a compilation step in the pipeline. As for the repositories, it is possible to provide different kinds of compilation methods for different programming languages by means of the plugin system. For our analyses described below, we use an MSBuild-based compiler plugin that can process projects used by the *Visual Studio* IDE. When the user specifies the project file in the repository to be analyzed, any plugin may also prompt for additional settings like the target configuration to be compiled in case of the MSBuild plugin.

The actual call graphs and the inheritance hierarchies are constructed by separate distiller plugins. For constructing the former, we iterate over all function symbols in use and retrieve the call graphs for them using *Phoenix*. During the construction of a call graph, plugins can also compute metrics that can be used to annotate and weight the edges. For example, our call graph extraction plugin determines whether a call spans one or more levels of the namespace hierarchy, how many parameters the method has, and how often the call is made, i. e. whether the edge is a multi-edge.

### 3.2. Data Format

All data extracted by plugins is stored in a custom, file- and directory-based graph format. The graphs of most revisions are only stored as differences to their previous revision, which drastically reduces the amount of storage required for a large time series on the one hand, but also enables the visualization to easily retrieve and display additive and subtractive graphs that only show the vertices and edges that have been added or removed from one revision to the next, on the other hand.

Plugins can use a data access API provided by the tool to write these graph files, which also serve as means for transferring data between the pre-processing step and the visualization tool. They can also rely on the system for aggregating metrics and assigning unique identifiers over all revisions, which allows for construction of a dynamic graph over time. While the destillation plugins are responsible for retrieving a graph per revision, the visualization then displays a series of those as static visualization of a dynamic graph. The finalization step that the visualization tool performs after the plugins have retrieved a graph per revision also includes the generation of a hierarchy based on the namespaces of the classes, which is used in the visualization to support navigation in the graphs.

## 4. Dynamic Call Graph Visualization

Our visualization techniques are based on static diagrams for displaying dynamic graphs. The most important feature of a static representation for time-varying data is the fact that dynamic patterns can be explored better and more easily when inspecting a subsequence of the evolving data on screen. This stands in contrast to animated representations where only one image at a time is visually represented to the viewer and comparisons to other data points of a longer sequence have to be made in visual working memory [TMB02]. In this section, we first illustrate the timeline and pixel-based overview representation that is used by our visualization system as a starting point for further data analysis.

### 4.1. Timeline Representation

Each row of the matrix expresses the time series data of a specific vertex of the project hierarchy. All vertices are hierarchically organized and the order of the matrix rows is deduced thereof by using a flat lexicographic order in each

© The Eurographics Association 2012.

subhierarchy in a recursive manner. Each column represents a specific point in time whereas the time axis starts at the left hand side and points to the right hand side, resulting in a left-to-right reading direction of the diagrams. The color-coded matrix cells either encode the number of incoming or outgoing edges from the corresponding node and point in time. To achieve a scalable version of such a timeline-based pixelmap representation we use aggregation and overplotting modes in both the vertex and time dimensions and pixel- or subpixel-based representations. Figure 1 shows an example of an overview. The hatched columns indicate that the corresponding revisions could not be tranformed into a call graph.

For reasons of good scalability and overview, the diagram can only depict the time-varying number of incoming or outgoing edges by special color codings, but the actual graph structure has to be explored by different visual metaphors as demonstrated in the next sections.

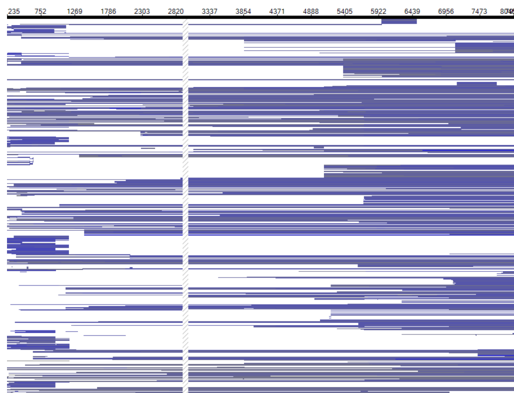


**Figure 1:** *The pixel- and timeline-based matrix view for the evolving graphs serving as an overview representation.*

### 4.2. Curved Node-Link Diagram

If all graphs in the sequence belong to the class of sparse graphs, i. e. only a few links are present, node-link diagrams may be used to show the directed relationships among the objects. We use one-dimensional vertical lines to position the graph nodes and a side-by-side representation for a subset of graphs in the sequence. To display the directed graph edges we use curved links on both sides of each vertical axis, i. e. upward and downward edges as also applied in the TimeArcTrees visualization technique [GBD09]. Figure 2 demonstrates the TimeArcTrees approach applied to a small time-varying directed graph.

The additional hierarchical organization of the graph nodes can be depicted in two major ways: as a node-link diagram and as an Indented Pixel Tree Plot [BRW10]. The node-link tree diagram can be represented as traditional and orthogonal style.

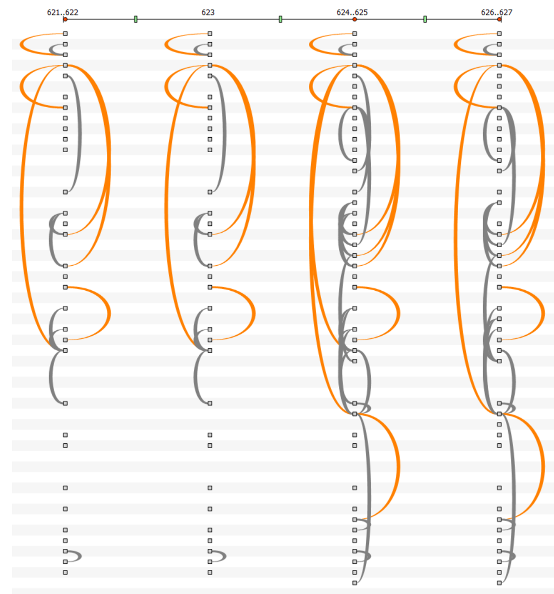


**Figure 2:** *Dynamic graphs shown as TimeArcTrees*

### 4.3. Radial Space-Filling Diagram

If we have to deal with dense graphs, node-link diagrams are not useful anymore because of a vast amount of visual clutter [RLMJ05] that is typically caused by many link crossings. Even sophisticated layout algorithms cannot produce aesthetically pleasing node-link diagrams and moreover, suffer from high runtime complexities causing problems for smooth interactions. If the graph is not static but evolves over time, further layout and visualization challenges occur. To understand the dynamic behavior of time-varying weighted and directed multi graphs we use the space-filling matrix-like and radial TimeRadarTrees technique [BD08], see Figure 3 for an illustrative example. Instead of explicit links, the relations are visually encoded by color-coded circle sectors with different shapes, sizes, orientations, i.e. distinguishing features to explore graph relations. For more details on the technique we refer to [BD08].

### 4.4. Source Code Visualization

The visualization tool also supports various details-on-demand features. The most prominent for software development processes is to allow the inspection and comparison of source code fragments. If a node is selected that corresponds to a part of the source code, e.g., a class or a method, the textual information is given in a separate frame.

## 5. Case Study

The usefulness of our visualization tool is illustrated by means of an application scenario where the ultimate goal is

© The Eurographics Association 2012.

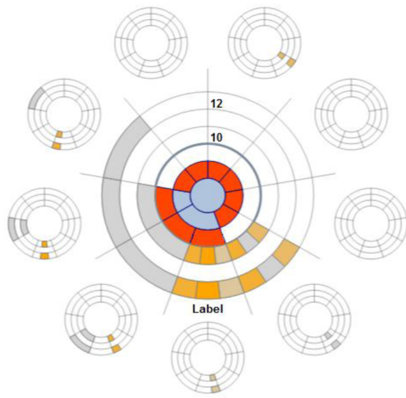


**Figure 3:** *The TimeRadarTrees technique uses radial diagrams to visually encode dynamic graphs.*

to derive and demonstrate insights from a large and time-varying dataset that contains many different types of data. The analysis process is divided into four stages:

- The data is first extracted from a software archive.
- It is preprocessed and dynamic call graphs are generated.
- The abstract datasets are visually depicted and combined by interactive features.
- Finally, already analyzed portions of the data can be stored to further explore it later on.

To demonstrate the usefulness of our tool we analyze the *GraphBox* project, which is a library actually used by our tool for representing graphs. The analyzed project was developed from November 2010 until October 2011 and contains 5,976 lines of code. The analyzed and displayed part of the project consists of 358 revisions (starting at revision 400 and ending at 757 inclusively) and 459 hierarchically organized software artifacts.

Figure 4 shows the graphical user interface (GUI) of our system with a timeline-based representation for the call relations of software artifacts of this software project. The GUI is actually divided into three different views:

- The hierarchical organization is shown at the left hand side by means of an Indented Pixel Tree Plot that can be scaled down to pixel- and even subpixel-based size and clearly reflects the hierarchical structure.
- The timeline view is represented in the center and shows a color-coded representation of the number of time-varying call relations, i.e., either incoming or outgoing calls aligned with the hierarchical structure.
- The graph bar is displayed on the right hand side and can be used to archive several views that need further investigation as a thumbnail representation for an analysis starting later on.

### 5.1. Overview First

The visualization tool is built on the Visual Information Seeking Mantra [Shn96] and hence, we first provide a coarse overview of the time-varying data. Even in this early analysis stage, we are able to detect some trends by inspecting the continuous and parallel colored lines. Furthermore, outliers can be uncovered by exploring the evolving data for short interrupted lines in the horizontal direction meaning that call relations also occur from time to time and do not exist over the whole period of the evolution process.

By this static timeline diagram combined with the hierarchical structure, we are able to easily explore the evolving graph data in different time intervals. All represented time series can be compared side by side for similar characteristics, trends, countertrends, periodicities, temporal shifts, and/or anomalies. Inspecting the timeline representation vertically, one can find some hatched columns expressing that the corresponding revision could not be used to transform it into a call graph, maybe because it was not compilable.

### 5.2. Zoom and Filter

As a next step, by zoom and filtering functions the user can select a rectangular region from the timeline-based pixelmap representation. The selected area is again displayed as a timeline-based pixelmap. This operation can be executed several times until the portion of the data is small enough to allow further and clearer investigation. Furthermore, each view can be put to the thumbnail list and all of them can be opened later on to start the exploration of the selected part from this point. Figure 5 shows the evolving call relations for the selected class *TimeRadarTree* of the analyzed project.

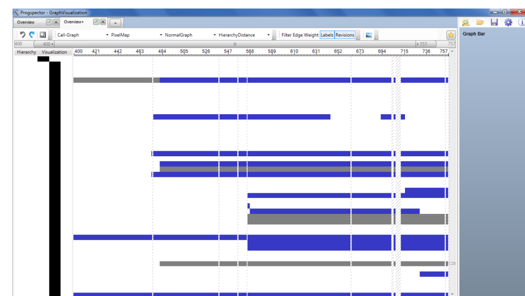

**Figure 5:** *A selected region by applying zoom and filtering.*

In Figure 5, there is an interruption in one of the timelines and a details-on-demand request reveals that the method *createShapes* is involved in this phenomenon. By inspecting the color coding we can find out that the metric value of the metric *HierarchyDistance* increases for the calls of *addHierarchyLeaves* from revision 481 to revision 482. Furthermore, a call relation occurs in revision 566 which only remains for the period of two revisions. Right below this call relation, another relation starts and remains much longer.

© The Eurographics Association 2012.

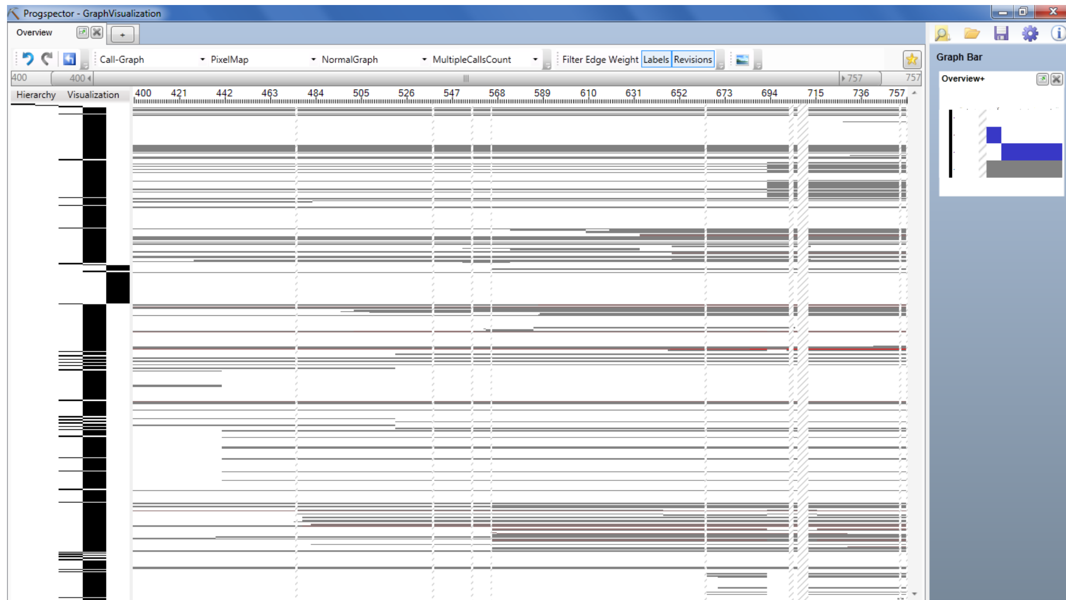


**Figure 4:** *The GUI showing an overview for the analyzed and visualized part of the project as a timeline-based pixelmap.*

Since this seems to be an interesting insight, we store this view and append it to the thumbnail view for future explorations. By using details on demand provided by a tooltip we suggest that this phenomenon is caused by single renamings, i.e., the starting letter of the method name is capitalized.

After this step, we return to the overview to obtain further insights. Now we open the class *PixelMap* as a zoomed and filtered timeline representation. A crosshair function can be used to better follow the focused row and column in the view. By this function and a details-on-demand request provided as a mouse tooltip we uncover breaks in the calls for the methods *imageScroller_MouseMove* and at the same time the first occurence of calls of the method *effectImage_MouseMove*. We suggest that some functionality was moved to another directory.

The zooming function is used to obtain more details about this phenomenon. To have a different point of view to the data we apply the TimeRadarTrees visualization technique here. The pixelmap representation benefits from good scalability, but only incoming or outgoing call relations can be explored one after the other and not at the same time. To fully understand an evolving graph structure we need both origin and target vertices involved in each relation.

For this reason, we provide various different visualization techniques that allow easy explorations of the dynamic graph data on different levels of hierarchical granularity. A TimeRadarTree can be used if we have to deal with dynamic directed compound graphs with the additional property that the graphs belong to the class of dense graphs, i. e., many relations exist that would cause a high degree of visual clutter in node-link diagrams.

Another feature would be to return to the former view by selecting the stored thumbnail from the list at the right hand side of the GUI and have a look at the TimeArcTree representation that uses curved node-link diagrams, see Figure 6.

In this figure, we can see that the representative node of the method *effectImage_MouseUp* is present in only two revisions and that it is not existing any longer. But directly after its existence the node for the method *EffectImage_MouseUp* occurs. By using interactive features and details-on-demand requests again, the outgoing edges of both vertices can be highlighted and we can detect that both have the same set of outgoing edges. Consequently, we can conclude that the developer of this method first named it the wrong way, but after some revisions he recognized the bug and corrected it in the next revision.

To further investigate distinctive features we look at the provided informations about the analyzed project. For the call graph, it is shown that the metric *MultipleCallsCount* is at a maximum of 54, but the average value for all call counts is roughly at 1.6, which is somewhat abnormal. We return again to the overview representation and change the displayed metric to *MultipleCallsCount*. After updating the view we see that the blue maximum value is hard to distinguish from the gray colored average values. Instead, we use a gray-to-red color coding and the outlier becomes obvious, see Figure 4.

© The Eurographics Association 2012.

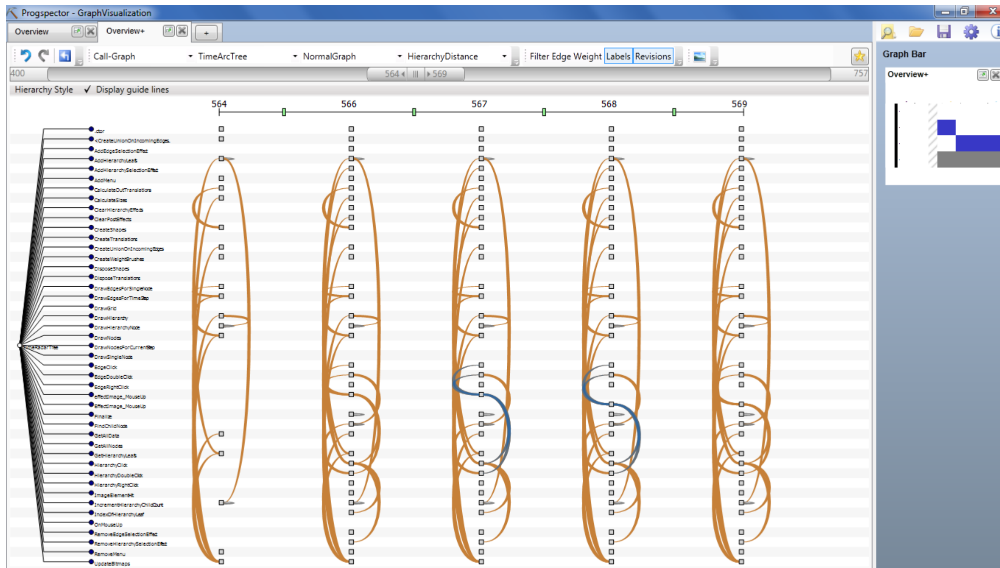


**Figure 6:** *A TimeArcTree for inspecting sparse time-varying graph structures by using curved node-link diagrams.*

### 5.3. Details on Demand

To further analyze this phenomenon we first zoom in the corresponding region of the overview and then change the time interval by selecting revisions 692 until 702. A TimeArcTree is used to inspect the time-varying call relations. We immediately detect the anomalous edge with the abnormally high metric value by its color and its orientation. In revisions 693 and 701, the edge has disappeared. For this reason, we aggregate this subregion that is not of special interest anymore and hence, obtain more space for the interesting part.

Furthermore, we switch the representation to a subtractive graph and we uncover that the edge with this high value is deleted in revision 693. By switching to the additive graph we can find out that it is added again in revision 702. By comparing subtractive and additive graphs we can conjecture that the project was reverted in revision 702 to the state of the former revision 692. We base this conjecture on the fact that in both revisions the same visual edge patterns occur.

By clicking on the node with the outgoing edge in revision 692 and by another clicking on the node of the corresponding edge in revision 702 we can inspect the source code for this node. By comparing both source code fragments we can see that both revisions are identical. A software maintainer may now inspect the code between the subsequent revisions to uncover why the transformation to a method with fewer calls has not worked before reimplementation.

This case study covers just a small set of features provided by our call graph extraction and visualization system. Explaining all details, views, data types, and interactive features of the system would go beyond the scope of this paper.

## 6. Discussion

There are several issues that are still not solved by our visualization system. The hierarchical organization of the vertices is represented as a 1D indented plot. The order of the vertices on the line is hierarchical, but we are aware of the fact that each subhierarchy may be rotated without destroying the hierarchical organization. To obtain a more scalable variant of the pixel-based timeline representation for the time dimension the GUI might use a scroll bar but also the concept of Rapid Serial Visual Presentation might be integrated. To better understand the strengths of our system it may be applied to datasets stemming from different application domains than software development.

In the visualization variant presented in this paper, we keep the same positions of all vertices in each of the views in the dynamic graphs because we base our visualization on the concept of mental map preservation that is best achieved when the node positions are fixed. This contradicts the concept of graph animation where vertices or vertex groups are smoothly moving around keeping a high degree of dynamic stability. We also support added or removed nodes and edges in our system at a specific point in time. These are represented as a timeline starting or ending at exactly this point in time in the view which is also difficult to achieve in animated graph sequences preserving the mental map. A special color coding might be used to indicate that the timeline for a node is starting or ending exactly there.

## 7. Conclusion and Future Work

We described a system for extracting, generating, visualizing, and analyzing time-varying data for dynamic call

© The Eurographics Association 2012.

graphs. The focus of this paper is on illustrating how the system can be applied to obtain interesting insights in the evolution of such call graphs by first providing an overview representation. We use a static pixel-based timeline representation with the goal to have a high degree of dynamic stability and to preserve a viewer's mental map. The additional hierarchical organization of a software system can be attached very easily to the representation and the viewer can apply interactive features to the overview and open additional views to inspect the data from different perspectives.

We demonstrated a case study where the novel pixel-based timeline representation can be used as an overview and starting point of an analysis process. By this overview representation the user is able to uncover anomalies that are otherwise hard to find, i.e, without a visualization tool. We illustrated a small set of supported additional visualization techniques such as TimeRadarTrees and TimeArcTrees. Apart from exploring dynamic call graphs, the tool can also be used to analyze various other types of data stored in a software archive. The tool should be extendable for representing dynamic graphs from different application domains apart from call graphs extracted from software archives. In future, we plan to extend the dynamic graph visualization by a more scalable variant for the time dimension with the goal to explore and compare longer graph sequences.

## Acknowledgment

We would like to thank the team of software engineering students that developed the *Progspector* system as a student project: Leonard Bruder, Christian Buchgraber, Daniel Exner, Stefan Gerzmann, Robin Goldberg, Miriam Greis, Jessica Hackländer, Severin Leonhardt, Nils Rodrigues, Hansjörg Schmauder, Christoph Schmid, and Benjamin Schmidt. For more on large-scale student projects we refer to [MRBW12].

## References

[BBD08] Burch M., Beck F., Diehl S.: Timeline Trees: visualizing sequences of transactions in information hierarchies. In *Proceedings of Advanced Visual Interfaces* (2008), pp. 75–82.

[BBV*12] Beck F., Burch M., Vehlow C., Diehl S., Weiskopf D.: Rapid Serial Visual Presentation in dynamic graph visualization. In *Proceedings of IEEE Symposium on Visual Languages and Human-Centric Computing* (2012).

[BC03] Brandes U., Corman S. R.: Visual unrolling of network evolution and the analysis of dynamic discourse? *Information Visualization 2*, 1 (2003), 40–50.

[BD08] Burch M., Diehl S.: TimeRadarTrees: Visualizing dynamic compound digraphs. *Computer Graphics Forum 27*, 3 (2008), 823–830.

[BRW10] Burch M., Raschke M., Weiskopf D.: Indented Pixel Tree Plots. In *Proceedings of International Symposium on Visual Computing* (2010), pp. 338–349.

[BVB*11] Burch M., Vehlow C., Beck F., Diehl S., Weiskopf D.: Parallel edge splatting for scalable dynamic graph visualization. *IEEE Transactions on Visualization and Computer Graphics 17*, 12 (2011), 2344–2353.

[CKJ*03] Collberg C., Kobourov S., J.Nagra, Pitts J., Wampler K.: A system for graph-based visualization of the evolution of software. In *Proceedings of International Symposium on Software Visualization* (2003), pp. 77–86.

[DG02] Diehl S., Görg C.: Graphs, they are changing. In *Proceedings of International Symposium on Graph Drawing* (2002), pp. 23–30.

[Die07] Diehl S.: *Software Visualization - Visualizing the Structure, Behaviour, and Evolution of Software*. Springer, 2007.

[ESS92] Eick S., Steffen J., Summer E.: Seesoft - a tool for visualizing line-oriented software statistics. *IEEE Transactions on Software Engineering 18*, 11 (1992), 957–968.

[FT08] Frishman Y., Tal A.: Online dynamic graph drawing. *IEEE Transactions on Visualization and Computer Graphics 14*, 4 (2008), 727–740.

[GBD09] Greilich M., Burch M., Diehl S.: Visualizing the evolution of compound digraphs with TimeArcTrees. *Computer Graphics Forum 28*, 3 (2009), 975–982.

[GFC04] Ghoniem M., Fekete J., Castagliola P.: A comparison of the readability of graphs using node-link and matrix-based representations. In *Proceedings of IEEE Symposium on Information Visualization* (2004), pp. 17–24.

[HFM07] Henry N., Fekete J.-D., McGuffin M. J.: Nodetrix: a hybrid visualization of social networks. *IEEE Transactions on Visualization and Computer Graphics 13*, 6 (2007), 1302–1309.

[MELS95] Misue K., Eades P., Lai W., Sugiyama K.: Layout adjustment and the mental map. *Journal of Visual Languages and Computing 6*, 2 (1995), 183–210.

[Mic] Microsoft Research, Phoenix Compiler and Shared Source Common Language Infrastructure, http://research.microsoft.com.

[MRBW12] Müller C., Reina G., Burch M., Weiskopf D.: Large-scale visualization projects for teaching software engineering. *Computer Graphics and Applications 32*, 4 (2012), 14–19.

[OM09] Ogawa M., Ma K.-L.: code_swarm: A design study in organic software visualization. *IEEE Transactions on Visualization and Computer Graphics 15*, 6 (2009), 1097–1104.

[OM10] Ogawa M., Ma K.-L.: Software evolution storylines. In *Proceedings of International Symposium on Software Visualization* (2010), pp. 35–42.

[RFF*08] Robertson G. G., Fernandez R., Fisher D., Lee B., Stasko J. T.: Effectiveness of animation in trend visualization. *IEEE Transactions on Visualization and Computer Graphics 14*, 6 (2008), 1325–1332.

[RLMJ05] Rosenholtz R., Li Y., Mansfield J., Jin Z.: Feature congestion: A measure of display clutter. In *Proceedings of SIGCHI Conference on Human Factors in Computing Systems* (2005), pp. 761–770.

[Shn96] Shneiderman B.: The eyes have it: A task by data type taxonomy for information visualizations. In *Proceedings of the IEEE Symposium on Visual Languages* (1996), pp. 336–343.

[TMB02] Tversky B., Morrison J. B., Bétrancourt M.: Animation: Can it facilitate? *International Journal on Human-Computer Studies 57*, 4 (2002), 247–262.

[VTvW05] Voinea L., Telea A., van Wijk J.: CVSscan: Visualization of code evolution. In *Proceedings of International Symposium on Software Visualization* (2005), pp. 47–56.

© The Eurographics Association 2012.