A Line Based Visualization of Code Evolution

S.L. Voinea, A. Telea and J.J. van Wijk

Technische Universiteit Eindhoven

Abstract

The source code of software systems changes many times during the system lifecycle. We study how developers can get insight in these changes in order to understand the project context and the product artifacts. For this we propose new techniques for code evolution representation and visualization interaction from a version-centric perspective. Central to our approach is a line-based display of the changing code, where each file version is shown as a column and the horizontal axis shows time. We propose a version centric layout of line representations, and we describe a cushion based technique to enhance visualization with information about stable evolution areas. We demonstrate the usefulness of our approach on real- life data sets.

Categories and Subject Descriptors (according to ACM CCS): D.2.2 [Software Engineering]: Design Tools and Techniques; D.2.7 [Software Engineering]: Maintenance, Enhancement; H.5.2 [User Interfaces]: Evaluation, Methodology

1 Introduction

In the last decade, software visualization has become a popular research area with a craving demand for results from the industry. According to industry surveys, the maintenance costs associated with software systems exceeded 90% of the total system costs in the last decade [Erl00]. Software visualization tries to address the challenge of reducing these costs both by preventive and corrective measures. We focus here on the corrective perspective. Here, software visualization addresses later, more expensive development phases, such as debugging, maintenance, restructuring, and optimization [ESS92] [FD04].

Industry practice studies show that 50% of the maintenance time is spent on program understanding [Sta84]. Many visualization tools have been proposed to help revealing the structure of a software system by means of reverse engineering [TMR02] [TWSM94]. However, most of them focus on a fixed high-level structural view that does not present the changes the system has undergone. This information can be of paramount importance, especially during debugging. Error propagation detection, for example, is one of the issues that are poorly supported by only one snapshot of the system structure. Intensive runtime analyses are required in order to discover dependencies. Visualizing the structure evolution, however, could enable the user to make relevant correlations between modified and faulty code, reducing this way the analysis scope. Collberg et al. [CKN*03] tried to overcome this limitation of high-level structure visualizations by a graph based technique showing the temporal dimension of software structures and mechanisms evolution. However, their still to be validated approach does not seem to scale well on real-life data sets. Additionally, the higher-level focus fails to reveal lower-level system changes, such as the many, minute source code edits which are often the source of errors. An apart class of visualization tools for program understanding use a line-based approach to represent code: source files are regarded as sets of code lines; code lines are visually encoded by pixel lines [ESS92] [FD04]. Most such tools are useful for revealing structure and change dependencies between code fragments. However, they do not reveal changes in the global context of an entire project life span. CVSscan, a recent tool addressing this limitation by a line-based visualization of software evolution was proposed in [VTvW04].

In this paper, we introduce several visualization and user interaction mechanisms that support understanding a file's evolution in a multi-version project from the perspective of a given version. Out goal is to support queries such as "How has the code changed from a given version on?" and "What is the impact of a given version to the project's outcome?"

The structure of this paper is as follows. In Section 2, we briefly review line-based visualization tools for software evolution. In Section 3, we outline the data model we use for the source code to be visualized and describe in detail two novel techniques: interpolated layout and cushion-based visual detection of stable code areas. Section 4 summarizes the interaction technique that support the interpolated layout. Section 5 presents a number of use-cases that illustrate our approach for investigating the evolution of real-life files. Finally, Section 6 summarizes our contribution and outlines future research.

2 Related work

Line-based software-evolution has been addressed by several visualization tools. SeeSoft is the first tool that proposes a direct code line to pixel line visual mapping [ESS92]. Color is used to show the age of a code fragment and enables users to correlate fragments that change in the same time. Augur [FD04], a recent effort in the area, uses the same color encoding as SeeSoft, but combines within one visual frame information about both artifacts and activities of a software project at a given moment. UNIX's `gdiff` and its Windows version WinDiff visualize code differences between two versions of a given file by depicting line insertions, deletions, and modifications, as computed by the popular `diff` utility. However efficient for comparing pairs of files, these tools cannot deal with file evolutions of hundreds of versions. CVSscan, an attempt to address such evolutions, gives an overview of code evolution for the *entire* life span of a project [VTvW04].

The above tools are successful in revealing the line-based structure of software systems, and uncover change dependencies at given moments in time. However, none of them, except CVSscan, provides insight into the code attributes and structure changes made throughout an *entire* project duration. While addressing this issue, CVSscan faces the challenge of a larger number of user queries during a typical session. To efficiently handle these, any tool requires not only a rich set of representations, but also efficient ways to navigate and explore the data. The work presented in [VTvW04] addresses the above only partially. Several important questions must still be answered:

- What is the value of a given version in the context of the entire project?
- What is the project evolution from the perspective of a given version?
- What are the stable code chunks during development?

To efficiently and effectively address the above, we propose several new visual representations, as well as custom navigation and interaction techniques for line-based visualizations. These are described next.

3 Representation and interaction techniques

Similarly to other line-based software visualization tools, CVSscan builds on the assumption that developers are comfortable with visualizations that show code in the same spatial context in which they construct it [ESS92]. Since software maintenance is mainly done at code level, we use a 2D line-based approach to visualize the software evolution [ES92, VTvW04]. The main questions we next had to answer were how to layout the code lines in 2D, and how to use color for encoding attributes. Finally, we had to design interaction techniques that enable users to navigate and explore the data.

3.1 Data model

CVSscan is a tool that visualizes the evolution of single files. The data come from the CVS version control management system, as described in detail in [VTvW04]. Briefly, CVS maintains versions for each file of a software project. A version V is a tuple holding the unique version ID, the author who committed it to the repository, the time of commitment, and its source code. CVS uses `diff` to compare the source code of consecutive versions V_j and V_{j+1} to find inserted and deleted lines in V_{j+1} with respect to V_j. Lines not deleted or inserted in V_{j+1} are defined as constant (not modified). Lines reported as deleted *and* inserted in some version are defined as modified (edited). Using `diff`, we can find which lines in V_{j+1} match constant (or modified) lines in V_j. For every such line, we call the complete set of matching occurrences in all versions (i.e. the transitive closure of the above match relation) a global line *gl*. Next, we compute several attributes characterizing the code evolution, as follows.

The **global line position G** associates to every global line *gl* a unique number $G(gl)$. Let us denote by l_i the i^{th} line of a version. G has two main properties. First, if $l_i \in gl$, then $i \le G(gl)$. Second, for l_i and l_j in the same version and $i < j$, we have $G(gl_i) < G(gl_j)$, where gl_i and gl_2 are global lines and $l_i \in gl_i$ and $l_j \in gl_j$. In other words, G gives a unique label to all code lines of all versions of a file, keeps the partial line orders implied by the different file versions, and ensures that lines in different versions identified by `diff` as instances of the same global line have the same label. In [VTvW04] we detail a graph-based approach to build the global line position function. Figure 1 depicts the typical outcome of such a mapping. At the top, we show the code of three versions of a file. At the bottom, the same file versions are shown, this time aligned on the vertical axis by using the global line position. Insertion of the line `int h=3` and deletion of the line `int i=1` show up as empty spaces.

The **line status L** characterizes the global position $G(gl)$ of line l_i in version V. L is one of the following:

constant: l_j in V_{i-1} is identical with l_i and has the same global line position.

modified: l_j in V_{i-1} or V_{i+1} has the same global line position but differs from l_j.

deleted: there is no line that has the global line position $G(gl)$ assigned to it in V, but there is such a line in one of the previous versions.

inserted: there is no line that has the global line position $G(gl)$ assigned to it in V, but there is such a line in one of the following versions.

modified by deletion: l_i is *modified* and l_{i+1} is *deleted* or *modified by deletion*.

modified by insertion: l_i is *modified* and l_{i+1} is *inserted* or *modified by insertion*.

Additionally, we extract structural information, using a fuzzy parser with a customizable grammar, by parsing the source code: blocks, comments, preprocessor macros, and so on. This yields the **construct C** attribute which describes, for every line l_i in every version V, the grammar construct that line belongs to.

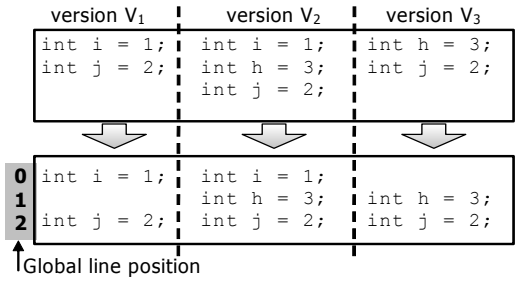


Figure 1: Global line position computation

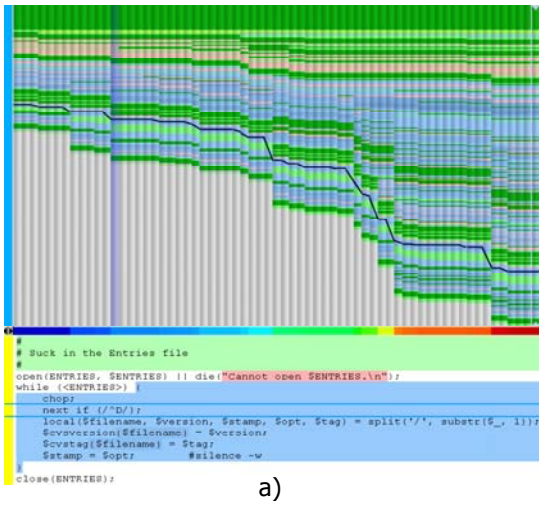
3.2 Interpolated layout

The driving idea in CVSscan was to enable users to easily and interactively perform analysis of software evolution. However, as discovered during the user studies done with CVSscan [VTvW04], these encodings address only part of the queries users have during software analysis. Evolution-related questions from the point of view of a given version (Sec. 2) are still hard to answer. To address these issues we propose a new layout technique that offers global insight in the code evolution from a version centric perspective.

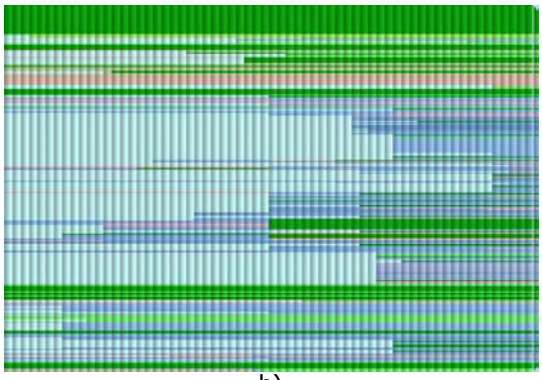
The CVSscan tool described in [VTvW04] offers a *file-based* and a *line-based* layout. Both use as *x* axis the version number. Each file is thus shown as a vertical stripe of horizontal pixel lines depicting codes lines colored by attribute values. Figure 2a shows a snapshot of CVSscan using the file-based layout. This layout uses as *y* coordinate the local line position and offers an intuitive 'classical' view on file organization and size evolution, similar to [ESS92].

The line based layout (Figure 2b) uses as *y* coordinate the global line position. It allows easy identification of code blocks that stay constant in time, or get inserted or deleted. Both above layouts use color to encode the construct attribute: dark green = comment, blue = nested statements, pink = strings. This is also visible in the lower part of Figure 2a, which shows the code under the mouse, interactively updated by a brushing mechanism. However insightful, these two layouts do not offer both an intuitive view of a *chosen* version (called the focus version) *and* a global overview of code deletion and insertion. We achieve this by a new type of layout: the *interpolated layout*.

Similarly to the file and line-based layouts, we map the version ID on the *x* axis. The main challenge we must address is how to use the *y* axis. We want to have a 'classical' view on the focus version with no empty spaces between lines, similar to the line based layout. In the same time we want to display all other versions so that it is easy to see inserted or deleted code. We need a layout scheme that maps a code line's evolution along a smooth curve on the screen, similar to the line-based layout. To address both objectives in one image, we propose a configurable interpolation scheme that combines the two layouts presented in [VTvW04]. We start from the bounding versions of the empty space interval with a line-based layout. Then we gradually decrease the *y* size of the empty spaces down to zero for the focus version (Figure 3). In this way the focus version appears as a contiguous stripe containing no empty spaces, just as in the file-based layout.



a)



b)

Figure 2: File-based (a) and line-based (b) layouts

In real-life software, a lot of code gets inserted and deleted during the project lifetime. The total *y* size of the focus version in the interpolated layout is considerably smaller than the sizes of the interval-bounding versions. The visual transition between their representations may thus become quite abrupt and difficult to follow. To make this transition smooth, we propose a number of complementary solutions.

First, we balance the representation by aligning the *y* midpoint of all versions with the image's *y* midpoint. The visual transition disruption caused by the vanishing empty spaces is now halved. Secondly, we use a configurable profile function to compute the size decrease of empty spaces, in order to distribute the visual transition disruption across the image's *x* axis. We use a weighted sum of exponential and hyperbolic tangent functions to compute the size of the empty spaces (Figure 4). Weight adjustment yields different visual disruption distributions.

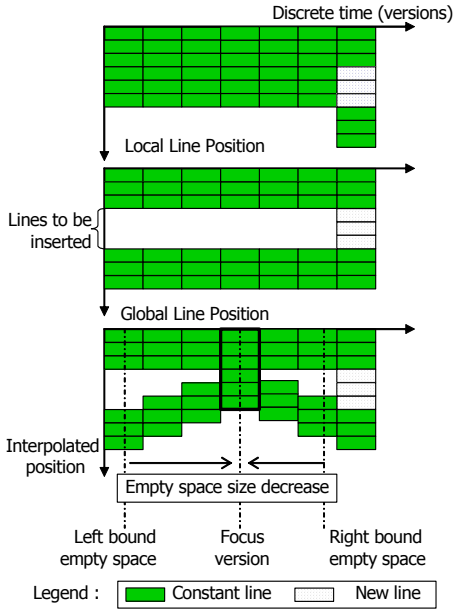


Figure 3: Line layout: file-based (top) line-based (middle) and interpolated (bottom)

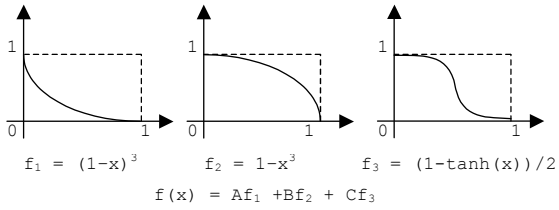


Figure 4: Profile function for empty space

The profile function is applied on the *x* distance between the version containing the empty space and the focus version. Its result is normalized such that it equals zero when the empty spaces are in the focus version and the height of a pixel line when the empty spaces are in an interval-bounding version (Figure 5).

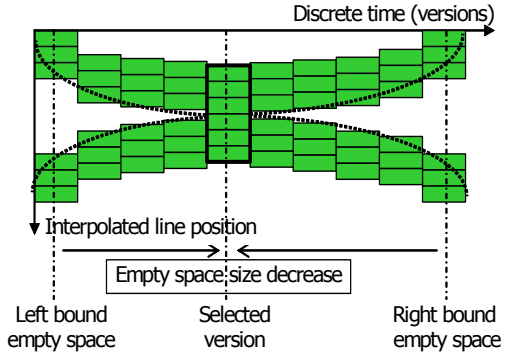


Figure 5: Balanced interpolated layout with asymptotical decrease of empty space size

Finally, an optional step is to balance the decrease in empty space size with a line height increase (Figure 6). This makes all versions have the same *y* size. However, the line height will differ from one version to another. This approach helps distributing the visual transition disruption on the focus version in the interpolated layout. However, as discovered in our user studies, it also affects the smooth visual navigation along the evolution of a code line. Therefore, the efficiency of this step must be further investigated.

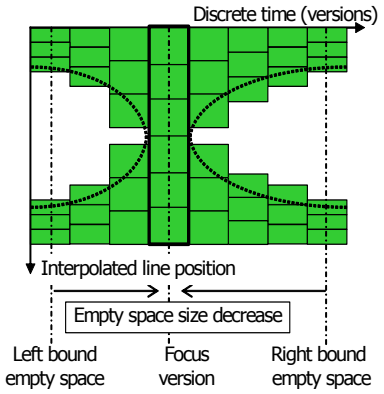


Figure 6: Compensated interpolated layout with sharp empty space decrease

Figure 7 shows the interpolated layout for the visualization of the evolution of a 3171 line C code file along 268 versions. Color shows line status: dark blue = constant, light blue = inserted, light red = deleted. We can easily see that most of the 'trash' code has been deleted before the focus version. Moreover, this version requires only few additions to reach the final form.

We next present a set of visual improvements for the interpolated layout that make navigation smoother. We also introduce a cushion-based technique to enrich evolution visualization with information about stable code fragments.

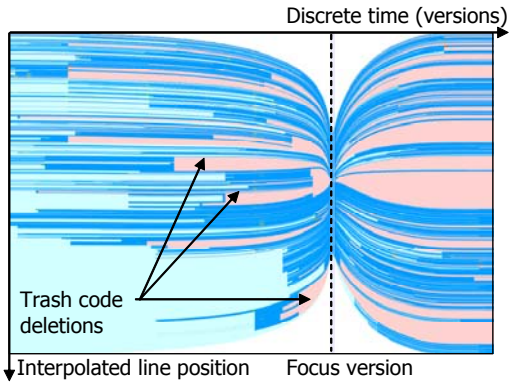


Figure 7: Interpolated layout, empty space reduction

3.3 Visual improvements

In the interpolated layout, the global line position (Sec. 3.1) does not map to a strait horizontal line on screen, as for the line-based layout, but to a curve. Encoding each code line as a horizontal pixel bar in a vertical version stripe causes a 'staircase' effect that disturbs the smooth visual navigation (Figure 8a). We solve this by skewing the individual line mappings. For this, we tilt the horizontal borders of each line rectangle such that the vertical border segments overlap when passing to the next version (Figure 8b).

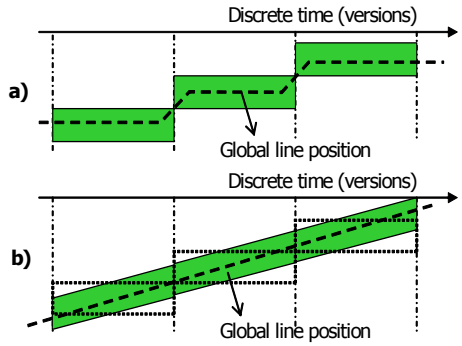


Figure 8: Interpolated layout without (a) and with skewing (b)

This replaces the 'staircase' effect with that of a poly-line. While still not perfect, this mapping removes the discontinuity feeling caused by the 'staircase' approach. Second, it is efficiently computed. Finally, the poly-line effect becomes negligible when version count exceeds 30.

A second interpolated layout improvement addresses the visualization of the evolution of large files, when we cannot fit the entire file on one screen, unless more lines share the same physical screen pixels. Given our aim to target real-life code, almost all files will fall into this class. The question is how to draw code lines that share pixels so that we get a consistent, comprehensible evolution image. We use a position based antialiasing algorithm that computes the color of overlapping lines using a weighted average. Two lines are overlapping when they share pixels on at least one of their vertical borders. The weight of each line is calculated according to the overlapping degree (0 for no overlap, 1 for complete overlap).

Figure 9 shows the antialiasing scheme for the interpolated layout of 100 versions of a 1350 line C code file. Color shows the line status attribute: dark blue = constant, light blue = inserted, and pink = deleted lines (see Sec. 3.1). Light blue and pink show

thus empty spaces in the layout. The rightmost version is in focus in both cases.

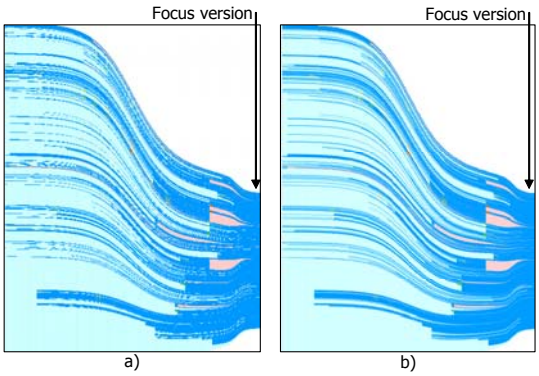


Figure 9: Interpolated layout a) without antialiasing b) with position-based antialiasing.

Position-based antialiasing preserves code structure over multiple zooming levels. An alternative would be to compute line weights using line attribute values. While this would help emphasizing lines based on their attributes, it may introduce structure inconsistencies when using different display magnification levels, so more research is needed to find out whether and/or how well this alternative works.

Finally, we enrich the evolution visualization with information on stable code fragments, i.e. code containing no insertions or deletions. The challenge is to emphasize these code blocks without modifying their layout and/or their colors that encodes attributes (line status, constructs, etc). We propose two methods for stable block detection: version-based and line-based. Both methods detect *contiguous* intervals in the version-line (x-y) space that contains no deletions and/or insertions and deliver a list of such intervals, sorted as follows. The line-based method maximizes first the number of lines (x axis) and is useful for revealing the long code fragments that are stable for a certain evolution time. The version-based method tries to maximize first the number of versions (y axis) and is useful for revealing the code blocks that are stable for long periods. To display the code blocks, we use parabolic cushions [vWvdW99], additively blended atop of the layout currently in use. This is our second use of cushions, the first being to show the versions themselves (see Figure 2, 11, 12). Size thresholds are used to interactively limit the search, e.g. to answer queries like "show all stable blocks longer than 100 lines or that have existed for more than 10 versions". Figure 10 shows the two methods on a code fragment of 50 lines followed along 65 versions. Color maps the line status attribute: dark green = constant, yellow = modified, light gray = inserted or deleted, red = modified by deletion, light blue = modified by insertion (Sec. 3.1).

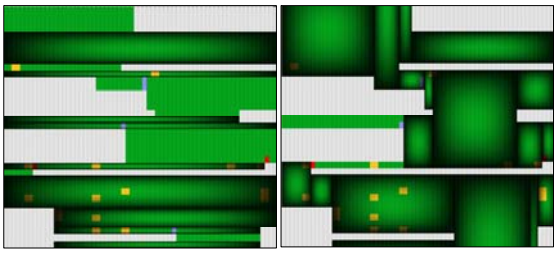


Figure 10: Detection of stable code fragments a) version-based b) line-based

Figure 10.a shows several wide blocks that have been stable for almost the entire evolution. Figure 10.b shows the longest (tallest) stable fragments. Color encodes line status, as explained above – the small yellow blocks visible in the lower part of the images indicate, for example, that only a few lines of code were edited. However, the large empty spaces in the layout indicate that many lines were inserted and deleted.

4 User interaction

Interactive file evolution exploration using the above techniques often leads to new, more specific questions about the data. We present a combined mouse and keyboard interaction scheme that enables users to easily navigate and explore the interpolated layout. From Shneiderman's perspective [Shn96], CVSscan offers a rich set of interactive exploration instruments. It gives an intuitive 2D **overview** on the evolution of files. It offers **zoom** and panning facilities to drill down to detailed representations. It has **filtering** mechanisms to remove irrelevant lines from the visualization and enables the user to **extract** specific evolution intervals for analysis. By means of an orchestrated set of **correlated** views CVSscan offers code level **details-on-demand**. Additionally, it enables the user to keep a **history** of his actions and lets him recover and reuse a specific visualization setting at a later time. All interaction instruments are designed to use a point-and-click approach, making the entire interaction possible only by the use of a mouse.

5 Use scenarios

We now present the outcome of using the described visualization and interaction mechanisms to visualize the evolution of a 450 lines Perl file along 65 versions, from the perspective of a focus version (Figure 11). In total, we target thus about 29000 code lines. The file comes from a public domain CVS repository, so it was not familiar to any of our visualization's users. Color encodes line status

(Sec. 3.1): dark blue = constant, yellow = modified, light blue = inserted, light red = deleted, light green = modified by insertion, dark red = modified by deletion. Line-based cushions (Sec. 3.3) show stable code areas.

This visualization helps us reason about the value of the focus version in the entire project's context. As Figure 11 shows, the focus version seems to contain a lot of code that gets deleted until the final version. At a closer examination we can see that most of this code (Figure 11, label C) is in the first half of the file. Also, the first half of the file still needs a large addition (Figure 11, label D) to reach its final form. This addition is quite fragmented – closer code inspection showed it involved rewriting already written code. This hypothesis is supported also by the relatively high number of changes in the later evolution. The second half of the file is different. Much of the trash code (Figure 11, label A) has been already removed from there before. The second half still needs quite a large amount of code to reach its final form. However, the inserted code is contiguous, so it probably does not interfere with already written code. This hypothesis is also supported by the small number of modified lines in the subsequent evolution of the file's second half – we confirmed this by looking at the source code itself. Overall, the focus version's second half is of better quality than the first half and contributes significantly to the final version.

Cushions complete the picture with useful information. The thick cushion spanning all versions (Figure 11, label F) shows that the file's beginning is stable throughout the whole evolution. The code block at the end of the file (Figure 11, label E) seems also to become stable soon after the first version. Finally, the focus version is close in time to a point (Figure 11, dotted line) from which much code becomes stable.

6 Conclusions

This paper presents a set of visualization and interaction techniques that support a version-centric examination of a file's evolution. Our goal is to extend previous experience obtained with the CVSscan software visualization tool [VTvW04], enhance its usability and effectiveness, and correct several problems found during CVSscan's validation. Our audience is the software maintenance community. Our goal is to provide support for program and process understanding in the context of code evolution. The first technique we propose is the interpolated layout, a new visual mapping of code lines for answering evolution related questions from the perspective of a given version. The second proposed technique uses cushions to emphasize code fragments that have a stable evolution. This enables one to easily identify and focus on relevant development areas. This technique is orthogonal to attribute color mapping, so it can be used for analyzing the evolution stability of any color-encoded attribute. While the interpolated layout helps answering version-centric questions, it also makes navigation more difficult. We correct this by a custom mouse + keyboard interaction scheme that constrains brushing to a given line or version. This technique can be used with any layout, e.g. the file-based and line-based layouts introduced in [VTvW04].

We validated the proposed techniques by implementing them as extensions of the CVSscan tool [VTvW04] and used the resulting tool to examine large files from real-life CVS repositories of hundreds of versions, such as Sourceforge as well as our own repositories. We present several scenarios using CVSscan to illustrate the efficiency of our approach on such data. We use CVSscan to navigate the data and answer version-centric questions on the file evolution in the context of the entire project. The complete software tool and several datasets can be downloaded from:

http://www.win.tue.nl/~lvoinea/soft/CVSscan_setup.exe

We would next like to extend our approach with higher-level overviews, such as whole-project, multi-file evolution visualizations, to enable evolution analyses on entire systems. Our final aim is to integrate CVSscan in code visualization and analysis toolset in order to make it effectively and efficiently available to the software development process.

7 Acknowledgements

This research was part of the ITEA project Space4U (http://www.win.tue.nl/space4u).

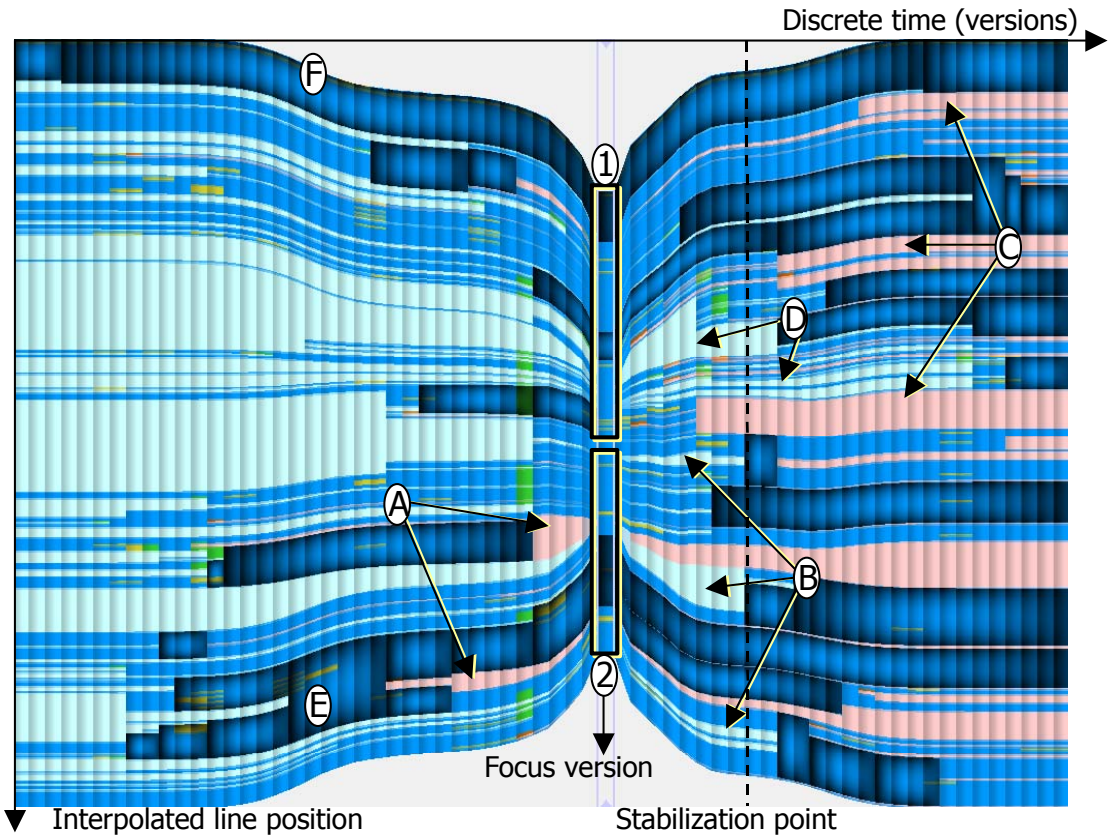


Figure 11: Version-centric visualization using the interpolated layout and the stable blocks detection

8 References

[CKN*03] COLLBERG C., KOBOUROV S., NAGRA J., PITTS J., WAMPLER K.: A System for Graph-Based Visualization of the Evolution of Software, *Proc. ACM SoftVis '03*, ACM Press, NY, 2003, 77 – 86.

[ESS92] EICK S.G., STEFFEN J.L., SUMNER E.E.: SeeSoft - A Tool for Visualizing Line Oriented Software Statistics. *IEEE Trans. on Software Engineering*, 18(11), 1992, IEEE CS Press, 957 – 968.

[Erl00] ERLIKH L.: Leveraging Legacy System Dollars for E-business. (IEEE) *IT Pro*, May-June 2000, 17 – 23.

[FD04] FROEHLICH J., DOURISH P.: Unifying Artifacts and Activities in a Visual Tool for Distributed Software Development Teams. In *Proc. ICSE '04*, IEEE CS Press, 2004, 387 – 396.

[Shn96] SHNEIDERMANN B.: The Eyes Have It: A Task by Data Type Taxonomy for Information Visualization. *Proc IEEE Symp. on Visual Languages (VL '96)*, IEEE CS Press, 1996, 336 – 343

[Sta84] STANDISH T.A.: An Essay on Software Reuse. *IEEE Trans. on Software Engineering*, 10 (5), Sep. 1984, 494 - 497.

[TMR02] TELEA A., MACCARI A., RIVA C.: An Open Toolkit for Prototyping Reverse Engineering Visualization. In *Proc. IEEE VisSym '02*, The Eurographics Association, Aire-la-Ville, Switzerland, 2002, 241 – 251.

[TWSM94] TILLEY S.R., WONG K., STOREY M.A.D., MULLER H.A.: Rigi: A visual tool for understanding legacy systems. In *Intl. Journal of Software Engineering and Knowledge Engineering*, Dec. 1994

[VTvW04] VOINEA L., TELEA A., VAN WIJK J.J.: CVSscan: Visualization of Code Evolution, submitted to *ACM SoftVis '05*, www.win.tue.nl/~lvoinea/cvss.pdf

[vWvdW99] VAN WIJK J.J, VAN DE WETERING H.: Cushion Treemaps: Visualization of Hierarchical Information. *In Proc. IEEE InfoVis'99,* IEEE CS Press, 73-78