Eurographics/ IEEE-VGTC Symposium on Visualization 2008
A. Vilanova, A. Telea, G. Scheuermann, and T. Möller
(Guest Editors)

Volume 27 (2008), Number 3

Code Flows: Visualizing Structural Evolution of Source Code

Alexandru Telea[1] and David Auber[2]

[1] Institute for Mathematics and Computer Science, University of Groningen, Netherlands
[2] LaBRI, University of Bordeaux, France

Abstract

Understanding detailed changes done to source code is of great importance in software maintenance. We present Code Flows, a method to visualize the evolution of source code geared to the understanding of fine and mid-level scale changes across several file versions. We enhance an existing visual metaphor to depict software structure changes with techniques that emphasize both following unchanged code as well as detecting and highlighting important events such as code drift, splits, merges, insertions and deletions. The method is illustrated with the analysis of a real-world C++ code system.

Categories and Subject Descriptors (according to ACM CCS): I.3.8 [Computer Graphics]: Applications D.2.8 [Distribution, Maintenance and Enhancement]: Restructuring, reverse engineering, and reengineering

1. Introduction

Analyzing the evolution of source code bases has become an important part of software maintenance. Code bases, stored in software configuration management systems such as CVS, Subversion or ClearCase, contain a wealth of evolution data, ranging from coarse-scale events, such as the addition or removal of a team member, to fine-scale events, such as the editings done to each line of code in every file. Several tools and techniques have been created to perform various types of analyses on code repositories. Such tools combine data mining components that extract the actual facts of interest from the repository with visualizations that let users see, navigate, and query the extracted facts to support specific tasks.

Although many software evolution visualization tools exist, they mostly support evolution analyses of the high-level structure of a project, *e.g.* analyzing the software architecture evolution, identifying the developer network, finding stable software releases, and monitoring software quality trends [Die07]. The main users of such tools are software architects, who manage large systems at medium or high abstraction levels, and are not interested in minute code details.

However, developers manage code at finer-grained levels. They need to understand how code blocks have moved around within or between files; whether a class or function was split, or whether code snippets were merged together; and how specific constructs (*e.g.* type declarations), evolve. Traditionally, such tasks are done with tools such as `diff` or WinDiff, which show a few versions of a given file, rendered as text, and highlight changed, added, or deleted lines. However, such tools have several limitations. First, a line-based approach is sensitive to layout changes or identifier renaming. Second, more complex change patterns such as code merge, split, or drift, important in maintenance, cannot be easily detected and/or shown. Finally, displaying files as text is not scalable.

Recently, a technique was proposed to visualize code structure changes at a syntactic level, by rendering syntax-matched code blocks in consecutive versions as icicle plots, and connecting these with straight tubes [CAT07]. In this paper, we extend this idea in several directions. First, we render code correspondences using textured splines connected to mirrored icicle plots, thereby simplifying the visual inspection. Second, we present structure tracking, a technique to detect and display code fragments which stay (near) constant during evolution, and separate them from highly changing code. Finally, we describe how to detect and visualize complex events of interest, such as code splits and merges. We illustrate our techniques for the analysis of a real-world C++ code base. All in all, our proposed technique can be used in several maintenance activities, such as understanding

low-level code changes, checking the presence (and persistence) of given small-scale code patterns, and overall bridge the gap between code and design or architectural inspection.

This paper is structured as follows. In Section 2, we review related work on visualizing source code evolution. Section 4 outlines our code matching method. Section 5 presents our visualization method. Section 7 demonstrates our method on a C++ code base. Section 8 discusses our method. Section 9 concludes the paper.

2. Previous Work

To understand software structure evolution at code-level, we need to combine code analysis and visualization techniques. Given two versions f_A and f_B of a file f, we can use clone detectors to find a set of code patterns $c_i^A \in f_A$ closely resembling a set of patterns $c_j^B \in f_B$. Several such clone detectors exist, *e.g.* [BYM*98, JMSG07, KFF06, WSvGF04, DN06]. However, most software evolution visualization methods work at either higher, or lower, levels than code *structure*. At architectural level, Beyer *et al.* mined and visualized graphs based on common source code changes [BN05]. Gall *et al.* visualize release histories as layered two-dimensional pictures to show system-subsystemŰmodule relations for versions over time [JRG99]. Baker and Eick [BE94] and Ball and Eick [BE96] use both static and animated dense-pixel visualizations of aggregated software metrics to observe the growth of software systems. Many authors visualized system structure evolution as a sequence of static layouts of *e.g.* call and inheritance graphs [CKN*03, Bey06]. Fischer and Gall visualized the evolution of dependencies between features [FG04]. Lanza proposed a powerful metaphor using matrices to show aggregated evolution data [Lan01]. Marcus *et al.* also show aggregated evolution metrics in 3D [MFM03]. A comprehensive survey of recent software visualization methods is provided by Diehl [Die07].

The above visualizations do not directly target code-level maintenance tasks. As an exception, Voinea *et al.* visualized the evolution of individual code lines of a file using dense pixel layouts [VTvW05], but their method cannot show structural change events, as it does not extract code syntax. Recently, Chevalier *et al.* proposed a method to detect and show similar structures in evolving C++ code bases using graph-matching techniques [CAT07]. In the following, we build upon the latter approach to support more precise and more insightful code evolution analyses.

3. Process Overview

Our method consists of several steps (Fig. 1). First, we use a C/C++ parser to extract an abstract syntax tree T_i (AST) from every version f_i of every file f of interest. Other parsers for other languages can be used, as available, as the next steps are fully generic in this respect.

Second, we use the code matching technique described in [CAT07] to detect correspondences in consecutive ASTs T_i, T_{i+1} (Sec. 4). Third, we propose a new technique, called code flows, to track and display stable code structures across the entire evolution, from the detected correspondence pairs (Sec. 5). Finally, we present a new way to detect and visualize events of interest, such as code merging and splitting, using icon sets (Sec. 6).

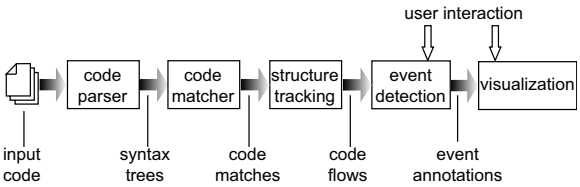

Figure 1: *Code structure visualization pipeline*

4. Code Matching

We first briefly sketch the code matching technique. As the full technique is quite involved, we refer to [ADDD07, CAT07] for details and pseudocode. Denote the N versions of a given source code file by $f_1, \ldots, f_N$, and their ASTs by $T_1, \ldots, T_N$. We call the nodes $m \in T(n)$ contained in the subtree $T(n)$ rooted at a node n the *descendants* of n, *e.g.* member-declarations are descendants of a class-declaration. In the following, 'type' denotes the syntax type of a code construct, *e.g.* class, function, variable, and so on, as defined by the analyzed programming language. For any two consecutive trees T_i and T_{i+1}, we identify correspondences between similar subtrees in a two-step approach. First, we group all nodes $u \in T_i$ and $v \in T_{i+1}$ into equivalence classes, using a distance metric $d(u,v)$

$$d(u,v) = [1 + d_{typ}(u,v)]\, d_{str}(u,v) \tag{1}$$

Here, d_{str} measures the *structural* difference between the syntax trees $T(u)$ and $T(v)$ rooted at u, v

$$d_{str}(u,v) = (\tilde{\delta}(u) - \tilde{\delta}(v))^2 + (\tilde{\nu}(u) - \tilde{\nu}(v))^2 + (\tilde{\sigma}(u) - \tilde{\sigma}(v))^2 \tag{2}$$

where $\delta(n)$ is the number of children of a node n, $\nu(n)$ is the number of nodes in $T(n)$, $\sigma(n)$ is the so-called Strahler number of $T(n)$, and $\tilde{\delta}$, $\tilde{\nu}$ and $\tilde{\sigma}$ are the values of δ, ν and σ normalized in $[0,1]$ over $T_i \bigcup T_{i+1}$. The Strahler number $\sigma(u)$, a well-known tree complexity measure in graph theory [Str52], is defined as

$$\sigma(u) = \begin{cases} 1, & u \text{ is a leaf} \\ \max_{1 \le i \le k}(\sigma(u_i) + i), & u \text{ has } k \text{ children } u_i \end{cases} \tag{3}$$

Further, $d_{typ}(u,v)$ denotes the *type* distance between two nodes, by default 0 if u,v have the same type and 1 otherwise. Different type distances can be user-defined, if we want *e.g.* to ignore specific syntax differences, such as the one between a C++ `struct` and a `class`. The distance d (Eqn. 1) hashes potentially similar code-subtrees in consecutive versions in the same equivalence class. Next, we build

correspondences between subtrees within the same equivalence class, using a top-down, recursive approach that finds the best matches between such subtrees. For details on the exact implementation, we refer again to [ADDD07, CAT07], which we fortunately can use as a black-box.

All in all, the matching process outputs a labeling $\alpha_i(u) = v$ for every node $u \in T_i$, which gives the node $v \in T_{i+1}$ that u was matched with, or *NONE* if no match was found. Matching all tree pairs T_i, T_{i+1} yields N such datasets α_i. The matched nodes and correspondences $\{\alpha_i : T_i \rightarrow T_{i+1}, \forall i\}$ form a directed acyclic *flow graph* G.

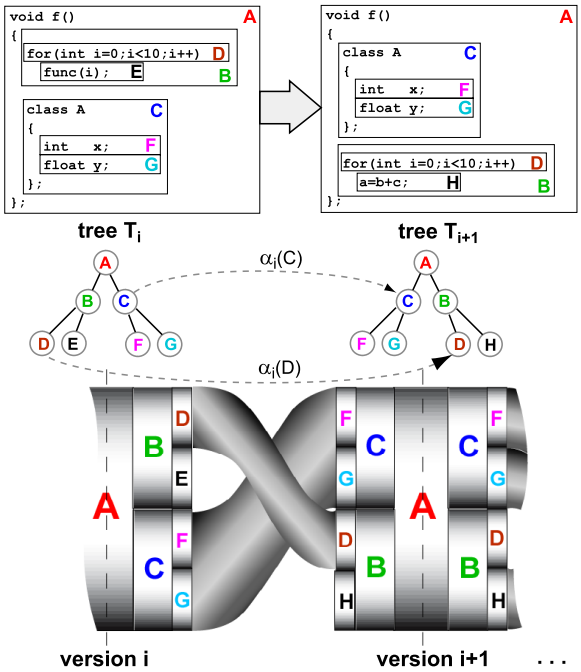

Figure 2: *Visualizing code evolution: source code (top), matched trees (middle), and visualization layout (bottom).*

Figure 2 illustrates the matching. For two code fragments, the matching finds two correspondences α_i (shown as dotted arrows) between their syntax trees T_i and T_{i+1}: the declaration of class `C` and the `for` loop. Given the top-down tree matching, correspondences are never nested. For example, since the nodes C in Fig. 2 are matched, as indicated by $\alpha_i(C)$, we do not store correspondences between their children F and G. Since we only match structure and types, we can transparently handle variable renaming, *e.g.* the class members `x` and `y` that become `u` and `v`, and code layout changes. Code swaps are handled implicitly, *e.g.* moving the class declaration before the `for` loop. Finally, the `for` loop body changes, so we have a deleted code fragment (node E), and a newly inserted one (node H), both unmatched.

5. Code flows visualization

5.1. Basic method

We now use the syntax trees T_i and flow graph G computed in the matching step (Sec. 4) to visualize the code evolution. We follow here [CAT07]: Every version i shows a layout of its syntax tree T_i, using a vertical icicle plot [BN01], which follows the order of code lines in the files (top to bottom). Every correspondence *i.e.* $\alpha_i(u) \neq NONE$ of two matched nodes $u \in T_i$ and $v \in T_{i+1}$ is drawn by connecting the nodes u,v by tubes. Thick tubes indicate large matched code fragments, thin tubes indicate small fragments.

To the above model proposed by [CAT07], we add several improvements. Figure 3 shows the original method and our enhancements. For an explanation of the colors, see Sec. 5.2 later on.

First, we draw each syntax tree T_i as a horizontally mirrored icicle plot of T_i instead of a simple plot (see Fig. 2 bottom). This lets us better see where each code fragment in version i went to the right in version $i+1$, and from where to the left (in version $i-1$) it came from, *e.g.* the code fragments marked A' and B' in Fig. 3 bottom-right. Second, we draw the correspondences as spline tubes, instead of straight cylinders. This produces easier to follow images, suggesting the actual 'code flow' metaphor. Third, we use a fading opacity texture having a Gaussian profile, opaque in the middle and transparent at the edges. This creates translucent flow tubes instead of opaque ones, compare *e.g.* Fig. 4 b with Fig. 4 a. Translucent tubes are better, as they leave a small white gap between neighbor tubes which enhances separation. Additionally, we draw a fully-opaque spline curve with a fixed width of 3 pixels at the tubes' centers. This has two benefits. First, correspondences between tiny code fragments, which can be crucial to see in *e.g.* debugging scenarios, are always visible. The enhancement is visible in both the overview (top) and detail (bottom) images in Fig. 3: The right images show some small-scale correspondences which are invisible in the left images (*e.g.* curve C). Second, the spline curve connects the centers of the matched nodes, thereby making more clear which code is exactly matched (*e.g.* Fig. 3 low-right).

In Figure 3 lower-right, we already see some facts. Yhere are not many insertions or deletions, *i.e.* icicle plot nodes to the right and left unconnected by tubes. Two relatively large code fragments stay unchanged (A', B'), while several small fragments drift, *i.e.* change place in the code.

5.2. Structure tracking

However, we still have a problem: How to follow the *entire* evolution of a given code fragment? Imagine a function which gets split in several code fragments as the software evolves. We want to visually follow these splits downstream the code flow, *i.e.* in future versions. A similar case holds

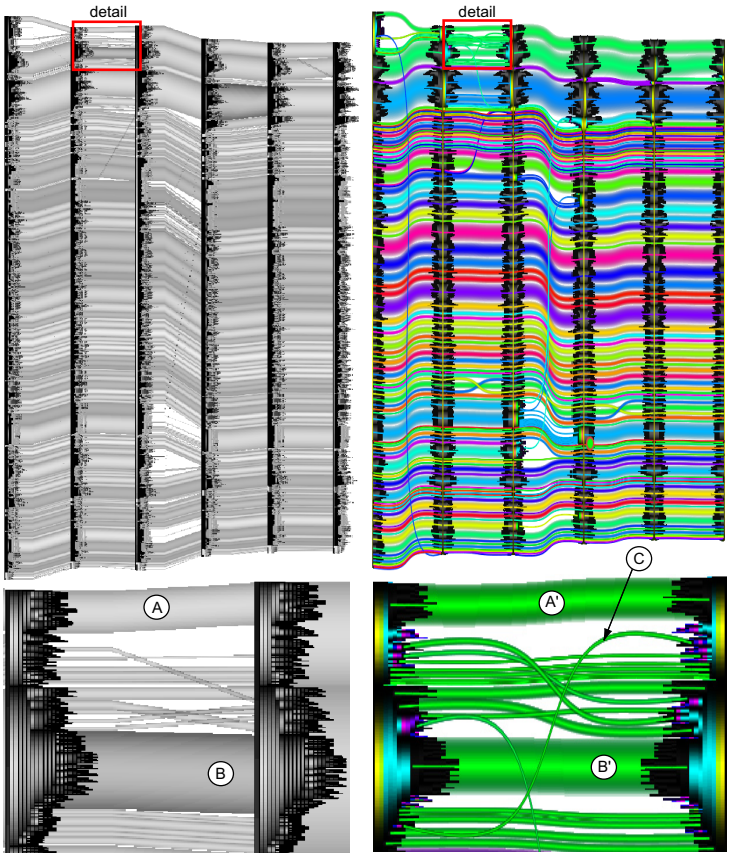

Figure 3: *Comparison: original method (left) and improved code flow visualization (right)*

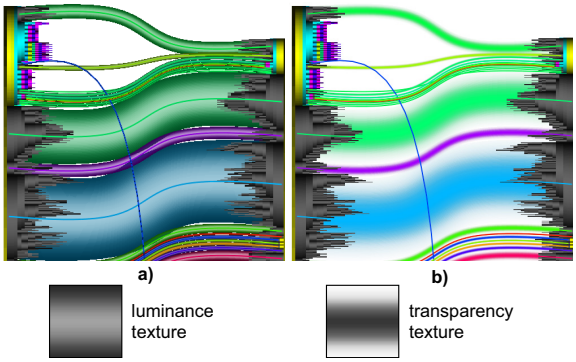

Figure 4: *Code-flow tube design: a) shaded; b) textured*

for code merging. We solve this using a technique called *code tracking*, as follows (see Fig. 5). First, we connect each matched node $n \in T_i, \forall i$ with all its children $m \in T_i$ which are also matched. This creates additional edges in the flow graph G (dotted lines in Fig. 5). Next, we decompose this enriched flow graph G in separate components, to obtain the evolving code fragments, as follows.

We assign a color $c(m)$ to each node $n \in T_i, \forall i$ which is not a match from version $i-1$, (A, B in Fig. 5, color=red). We can either pick colors cyclically from a small-size colormap, or map types to colors, *e.g.* functions=yellow, loops=green, variables=purple, etc. Next, we propagate the colors downstream in G in breadth-first order, until the final version T_N. We compute the color $c(m)$ of each visited node m by weighting the colors of its children in G by their AST tree sizes. A node $n \in T_i$ having several descendants matched in version $i+1$ encodes a code split event (*e.g.* C in Fig. 5 top, split into D', E' and F'). When $n \in T_i$ has several descendants matched in version $i-1$, it encodes a code merge event (*e.g.* C in Fig. 5 bottom which merges A and B). After the colors reach the final version, we execute the same propagation upstream, towards version 1. This mixes colors at code split nodes. We repeat the upstream-downstream propagation a few (*i.e.* 4-5) times, after which the colors converge.

The produced picture shows the evolution of distinct code fragments, or *code flow*, throughout all versions (*e.g.* Fig. 3 right). To help readability, we also added brushing which displays the actual code text under the mouse in a tooltip. We can now follow a code fragment upstream or downstream by its flow color. Although colors get mixed

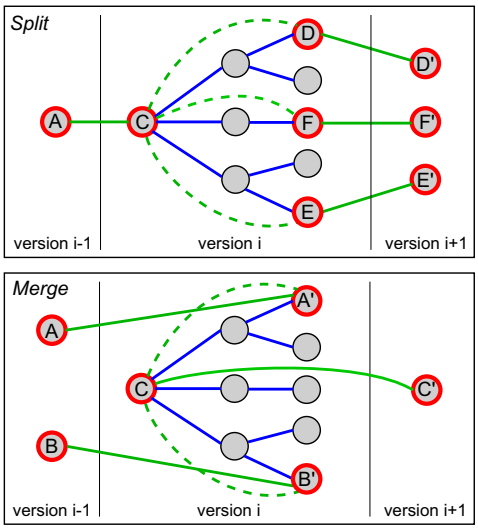

Figure 5: *Structure tracking. Flow graph edges are green, AST edges are blue. Red nodes indicate a tracked code flow.*

at merge/split events, the repeated propagation causes split or merged code-flows to retain similar colors before and after the event. The overall result, complete with the mirrored icicle plots and textured spline tubes, suggests a complex cable-and-plug wiring, which may be a good metaphor for the intricacies of fine-scale code evolution.

6. Visualizing events of interest

The code flow pictures give an overview of the structure evolution. Still, developers need to see specific evolution events, such as changes done to a given function, or find code split and merge events. We add these queries atop of the code flow visualization as follows.

We provide both lexical (text-based) and structural (AST based) queries to find specific structures. Lexical queries search nodes by their actual code text. Structural queries search tree patterns using regular expressions, *e.g.* "find all classes having a method which returns a float". The found nodes, and all their upstream and downstream matched nodes, are then highlighted in a search color. This outlines the path of the fragment of interest. Additionally, we add *event icons*, which are small custom-defined bitmaps, on the found nodes atop the code flow visualization. The icons attract the user's attention to the type of event being found. We also allow the user to highlight custom points of interest in the code manually, by clicking on them and adding the desired icons.

Insertion and deletion events already appear as white gaps between the correspondence tubes. We emphasize these further by coloring matched nodes in gray, to distract attention from them, and unmatched (inserted or deleted) nodes according to their syntax types.

We could detect code splits and merges using our structure tracking heuristic (Sec. 5.2). However, even small code changes can generate splits or merges, yielding too many such events. We improve this by letting the user specify the 'strength' of a split or merge as follows. For every syntax tree T_i, we define a labeling $\lambda_i(n) \rightarrow \mathbb{R}$ on all its nodes n as follows. First, we label all leafs l of T_i by increasing integers starting at 0, by traversing T in depth-first order, and also define their sizes $s(l) = 1$. Next, we label all non-leaf nodes n with the average of their children's labels weighted by the children sizes

$$\lambda(n) = \frac{\sum_{k=1}^{C(n)} s(c_k(n))\lambda(c_k(n))}{\sum_{k=1}^{C(n)} s(c_k(n))} \tag{4}$$

where $c_k(n), k = 1 \ldots C(n)$ denote all children of n. The size of non-leafs is the sum of their children sizes, *i.e.* $s(n) = \sum_{k=1}^{C(n)} s(c_k(n))$. Intuitively, $\lambda(n)$ are the y positions of the nodes' centres in the icicle plot, and $s(n)$ are their heights.

Next, we proceed as follows. Consider two nodes $n, m \in T_i$, matched to $n' = \alpha_i(n), m' = \alpha_i(m) \in T_{i+1}$. We say that n, m get split from version i to version $i+1$ if

$$|\lambda(n) - \lambda(m)| \leq k_{min} \frac{s(n) + s(m)}{2} \text{ and } |\lambda(n') - \lambda(m')| \geq f k_{min} \tag{5}$$

The parameter k_{min} says when the nodes n and m are considered 'in the same code entity', and is a fraction of their sizes. Hence, small code fragments, *e.g.* expressions, need to be closer to each other than larger code fragments, *e.g.* class declarations, to be considered in the same entity. Setting $k_{min} = 1$ requires n and m to be consecutive code fragments. The parameter f says how far apart must n and m drift to be considered split, as compared to their initial closeness. Higher f values detect stronger splits. Both k_{min} and f are dimensionless values. In practice, setting $k_{min} \in [1, 2]$ and $f \geq 5$ has given very good results on a variety of code bases. Merges are detected analogously, using T_{i-1} instead of T_{i+1}.

We visualize detected merge or split events by adding merge/split event icons on the merged, respectively split nodes, just as for the search-based events, and/or by highlighting the involved correspondences in color.

Finally, we detect code drift using the same distance metric as for merge/splits. For every correspondence $\alpha_i(n \in T_i) = m \in T_{i+1}$, we compute the distance $|l(n) - l(m)|$ between the current and next positions of the same node. If this distance exceeds $f k_{min} s(n)$, we have a code drift event for n. We visualize this event just as for the merge/splits, using icons or correspondence highlighting.

7. Application Examples

We have implemented the above visualization techniques using the Tulip visualization framework [Aub03]. We now illustrate the use of this tool in analyzing code evolution. In the following, we refer to Figure 6, a snapshot showing the

evolution of about 6000 lines of C++ using complex constructs (*e.g.* templates) over six versions. To make our test more difficult, we renamed identifiers in the different versions, inserted random comment and blank lines, and randomly changed the code layout and indentation. The analysis described below was performed by a seasoned C++ developer who had no knowledge of the involved code base, but was familiar with the visualization tool. During the analysis, heavy use was made of the code brushing to discover the actual code text. The results were verified by one of the code base's developers.

Task 1: Getting a first overview

The programmer wants first to get a global overview of the code changes. Figure 6 tells several things at a glance. The code shrinks gradually from version 1 to version 6. Also, large parts stay unchanged, as indicated by the several parallel tubes (g,h). Some interesting small-scale events occur, indicated by the upwards and downwards crossing tubes (*e.g.* c,d,j,i). These are interesting events, to be further analyzed.

Task 2: Finding drifts, merges, and splits

The user next performs the automatic event detection using $k_{min} = 1.5$ and $f = 5$ (Sec. 6). Matched nodes and correspondences *not* involved in the detected events are de-emphasized by coloring them gray. Inserted and deleted nodes and correspondences involved in the detected events are emphasized by coloring using distinct hues. Also, the detected split and merge events are marked by icons. Some non-trivial events become now visible. First, the method `f` at the beginning (top) of version 1 gets heavily changed (Fig. 6 a). About 50% of its code gets deleted. Two small fragments from its middle get split. One of them, containing a destructor call, drifts all the way to the bottom of version 2 (thin blue tube (b)). The other one is merged with the first and last fragments of `f` into a new instance of this method in version 2 (b). In version 2, `f` undergoes complex modifications. Some fragments surviving from version 1 (green tubes) drift and get redistributed over several parts of the code.

In version 3 (top), a small-scale merge is seen (indicated by the icon). Here, a small fragment coming from the method `f` in version 2 (green tube) gets merged with two other small fragments coming from a previous split in version 2 (purple and blue tubes). The merged fragment continues unchanged until version 5 (green tube).

Version 3 (bottom) shows also a split (k). A class gets refactored (the thick cyan tube coming from version 1 into icon (k)). Part of its code gets deleted. Two small fragments drift upwards to two different classes in version 4 (cyan and blue tubes), where they get merged with other pieces of code. Finally, several small-scale declarations in this class survive in its new instance in version 4 (thin parallel gray tubes in (k)).

Version 4 shows two salient merges. In the top one (d), two function calls, coming from the split, respectively merge in version 3, get merged into a newly inserted code fragment, in this case a complex `for` loop. This was actually a surprise for the developer, since those function calls were not supposed to occur in that specific bit of code. In the bottom merge (e), a new class gets declared, which absorbs several methods from existing classes, as shown by the blue tubes merging to the left of the icon. Next, this class continues unchanged till the end (thick blue tube at right of event (e)).

Task 3: Finding code swaps

Code swaps are interesting, as they could hint to small-scale refactoring occuring during debugging. A first swap is visible in version 2, shown by the crossing blue and yellow tubes (j). These fragments get swapped once more in version 3 (not marked in the picture), and once more in version 4 (l). Apparently, the developer changed his mind several times about the swap, and in the end he left the code as it initially was. We did not provide an automatic way to detect swap events. Still, they are easily visible as symmetrically crossing tubes.

Task 4: Checking design rules

The combination of search and correspondence highlighting can be used to find design violations. For example, we can search whether a specific code construct, *e.g.* a declaration, stays within (or without) a given target, *e.g.* function. For this, one searches for the source or selects it by browsing and clicking on it. The same is done for the target. Next, we highlight all the upstream and downstream correspondences of both fragments in two different colors, and check if these intersect or diverge. Figure 6 (m) is such a case. Here, two destructor calls were selected which the developer knew that should always stay together, and also be the last operations executed in the analyzed file. We see two thin, blue and red, parallel tubes running indeed in parallel at the bottom from version 2 onwards. However, the blue tube did not stay together with the red one in version 1, but drifted from the top of the file, when method `f` was split (a). Hence, our rule is violated in version 1 but holds from version 2 onwards.

8. Discussion

Many other scenarios are possible. We can reduce the level-of-detail at which code is displayed *e.g.* to show syntax elements only above some granularity level (functions, classes, etc). This enables showing more information, *e.g.* more files, in a single view, and also moves the code assessment at a higher level. This can be done simply by filtering the input correspondences as needed, or setting the matching algorithm's parameters. The syntax-based code flow visualization is quite powerful, as it can target all detail levels from a single identifier to entire functions only by a few parameter changes. Given a parser able to extract ASTs from large-scale code, our visualization is, hence, easily scalable.

Although understanding how to interpret and use the code

flows was not immediate (it took approximately one hour to the four users we presented it to), there are few tools that can provide this type of insight. Our previous experience showed that it is very hard, if not impossible, to detect and track patterns such as code swaps, drift, merge and split over more than two versions at a time using existing tools such as *diff*, WinDiff or CVSscan. We believe the code flow metaphor is a powerful aid for developers to comprehend code-level evolution changes, bridging the gap between line-level and high-level (*e.g.* class, file, or aggregate metrics) evolution visualizations.

A similar, yet independently developed, technique to code flows was proposed by Viegas *et al.* [VWD04]. Here, the history of wiki pages is visualized. Identical text fragments from consecutive versions are linked by thick bands. The result is quite similar to Fig. 3 left. Interestingly, they also suggest looking at software evolution in their future work. Compared to them, our technique uses splines instead of straight lines for a more effective following of the evolution; shows correspondences between elements in *hierarchies* instead of plain text blocks; and uses color to emphasize the identity of a fragment instead of the fragment's first author.

9. Conclusion

We have presented code flows, a set of visualization techniques for analyzing source-code structure evolution above the line level but below the file or aggregate metrics level. We propose propose several rendering and layout techniques to suggestively show how bits of code evolve over several code versions, visually track a given code fragment throughout its evolution with guaranteed visibility, and detect and highlight more complex events such as code splits, merges, and swaps.

Many extensions of the code flows are possible. One of them is to show code clone evolution across several versions. Next, we plan to add information such as code quality metrics, programmer IDs, or bug data, mined from code repositories to the code flows, to assess *e.g.* how code marked as buggy drifted during a project, who was responsible for it, and thereby detect potential problems. This is helpful in both corrective and preventive maintenance.

References

[ADDD07] AUBER D., DELEST M., DOMENGER J. P., DULUCQ S.: Efficient drawing of RNA secondary structure. *J. Graph Algo. & Appls. 10*, 2 (2007), 329–351. 2, 3

[Aub03] AUBER D.: Tulip : A huge graph visualisation framework. In *Graph Drawing Software - Mathematics and Visualization* (2003), Springer, pp. 105–126. 5

[BE94] BAKER M., EICK S.: Visualizing software systems. In *Proc. ICSE* (1994), pp. 59–67. 2

[BE96] BALL T., EICK S.: Software visualization in the large. *IEEE Computer 29*, 4 (1996), 33–43. 2

[Bey06] BEYER D.: Co-change visualization applied to PostgreSQL and ArgoUML. In *Proc. MSR* (2006), pp. 165–166. 2

[BN01] BARLOW T., NEVILLE P.: A comparison of 2D visualizations of hierarchies. In *Proc. IEEE InfoVis* (2001), pp. 131–139. 3

[BN05] BEYER D., NOACK A.: Clustering software artifacts based on frequent common changes. In *Proc. IWPC* (2005), pp. 259–268. 2

[BYM*98] BAXTER I., YAHIN A., MOURA L., SANT'ANNA M., BIER L.: Clone detection using abstract syntax trees. In *Proc. ICSM* (1998), pp. 368–377. 2

[CAT07] CHEVALIER F., AUBER D., TELEA A.: Structural analysis and visualization of c++ code evolution using syntax trees. In *Proc. IWPSE Workshop* (2007), ACM, pp. 90–97. 1, 2, 3

[CKN*03] COLBERG C., KOBOUROV S., NAGRA J., JACOB P., WAMPLER K.: A system for graph-based visualization of the evolution of software. In *Proc. ACM SoftVis* (2003), pp. 77–85. 2

[Die07] DIEHL S.: *Software Visualization: Visualizing the Structure, Behavior and Evolution of Software*. Springer, 2007. 1, 2

[DN06] DUCASSE S., NIERSTRASZ O.: On the effectiveness of clone detection by string matching. *Intl. J. on Soft. Maint. and Evolution: Research & Practice 18*, 1 (2006), 37–58. 2

[FG04] FISCHER M., GALL H.: Visualizing feature evolution of large-scale software based on problem and modification report data. *J. of Software Maintenance 16*, 6 (2004), 385–403. 2

[JMSG07] JIANG L., MISERGHI G., SU Z., GLONDU S.: Deckard: Scalable and accurate tree-based detection of code clones. In *Proc. ICSE* (2007). 2

[JRG99] JAZAYERI M., RIVA C., GALL H.: Visualizing software release histories: The use of color and the third dimension. In *Proc. ICSM* (1999), pp. 99–108. 2

[KFF06] KOSCHKE R., FALKE R., FRENZEL P.: Clone detection using abstract syntax suffix trees. In *Proc. WCRE* (2006), pp. 253–262. 2

[Lan01] LANZA M.: The evolution matrix: Recovering software evolution using software visualization techniques. In *Proc. IWPSE* (2001), pp. 109–118. 2

[MFM03] MARCUS A., FENG L., MALETIC J.: 3D representations for software visualization. In *Proc. ACM SoftVis* (2003), pp. 27–36. 2

[Str52] STRAHLER A.: Hypsomic analysis of erosional topography. *Bulletin of Geol. Soc. of America 63* (1952), 1117–1142. 2

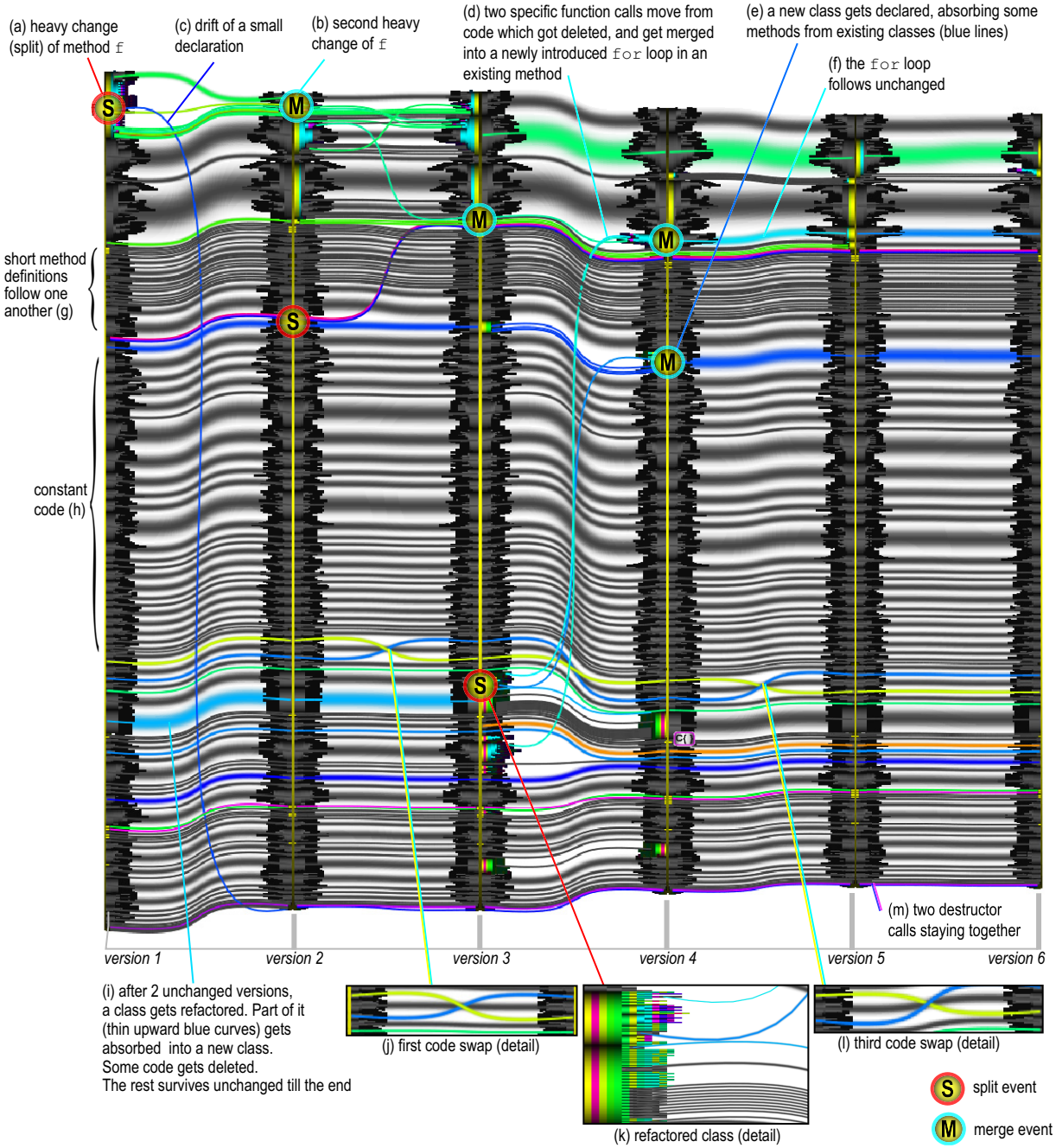

Figure 6: *A complex example of events of interest visualized using code flows*

[VTvW05] VOINEA L., TELEA A., VAN WIJK J. J.: Cvsscan: Visualization of code evolution. In *Proc. ACM SoftVis* (2005), pp. 47–56. 2

[VWD04] VIEGAS F., WATTENBERG M., DAVE K.: Studying cooperation and conflict between authors with history flow visualizations. In *Proc. CHI* (2004), pp. 575–582. 7

[WSvGF04] WAHLER V., SEIPEL D., V. GUDENBERG J. W., FISCHER G.: Clone detection in source code by frequent itemset techniques. In *Proc. SCAM* (2004), pp. 128–135. 2