Structural Analysis and Visualization of C++ Code Evolution using Syntax Trees

Fanny Chevalier, David Auber, Alexandru C Telea

Structural Analysis and Visualization of C++ Code Evolution using Syntax Trees

Fanny Chevalier
LaBRI, Bordeaux
France
chevalie@labri.fr

David Auber
LaBRI, Bordeaux
France
auber@labri.fr

Alexandru Telea
Eindhoven University of Technology
Netherlands
alext@win.tue.nl

ABSTRACT

We present a method to detect and visualize evolution patterns in C++ source code. Our method consists of three steps. First, we extract an annotated syntax tree (AST) from each version of a given C++ source code. Next, we hash the extracted syntax nodes based on a metric combining structure and type information, and construct matches (correspondences) between similar-hash subtrees. Our technique detects code fragments which have not changed, or changed little, during the software evolution. By parameterizing the similarity metric, we can flexibly decide what is considered to be identical or not during the software evolution. Finally, we visualize the evolution of the code structure by emphasizing both changing and constant code patterns. We demonstrate our technique on a versioned code base containing a variety of changes ranging from simple to complex.

Categories and Subject Descriptors

I.3.8 [**Computer Graphics**]: Applications; D.2.8 [**Distribution, Maintenance and Enhancement**]: Restructuring, reverse engineering, and reengineering

Keywords

code structure evolution, syntax trees, software visualization

1. INTRODUCTION

Software configuration management (SCM) systems, such as CVS, Subversion, ClearCase and Perforce, are an important part of the management of large-scale software development projects. SCM systems support tasks such as checking in and out different file versions in a software project, managing parallel development branches, release scheduling, bug management, and collaborative teamwork. At the lowest level, a SCM system supports these functions by maintaining a history of *changes* of the software artifacts. Typically, these artifacts are source code (text) or binary files, and the changes are recorded and managed by the SCM system at line or character (byte) level.

IWPSE '07, September 3-4, 2007, Dubrovnik, Croatia.

In the recent years, several tools and techniques emerged to augment the above basic functionality with more complex features. These tools can be loosely classified into *data mining* tools, which extract various facts from code repositories, and *visualization* tools, used to present and navigate both raw data and extracted facts. An overview of such tools is presented in [17]. Such tools support a wide range of tasks, such as identifying the developer network, finding stable software releases, monitoring software quality and debugging activity, and analyzing the software architecture evolution. The primary users of such tools are software architects, who must manage large systems at medium or high abstraction levels, and are not interested in all the minute code details.

In this paper, we focus on a subset of these activities. We are interested to understand how the detailed structure of source code changes during the evolution of a code project consisting of hundreds of source code files, changed by tens of developers over several years. Developers are interested to see which parts of the code changed, and how. At a fine level-of-detail, this can be done by visualizing the evolution of each code line in one, or a few, files of interest across all versions. Tools such as WinDiff and CVSscan support this task, using the `diff` operator of the underlying SCM system to tell which lines have been changed, added, or deleted [18]. However useful to see small-scale, minute code changes, this approach has several limitations. First, one cannot detect and show more complex code evolution patterns located at coarser levels of details, such as the moving of a whole code block from one function or class to another. Second, the line-based code evolution analysis is sensitive to low-level changes such as reformatting, identifier renaming, expression rewriting, or declaration order change. Such changes are irrelevant to activities such as code refactoring, rearchitecting, and generally when one works with software source code at higher levels.

In this paper, we present a set of techniques to detect and visualize structural code evolution patterns for C and C++ code repositories. We use a C++ parser [13] to extract a full abstract syntax tree (AST) from every version of every file of interest in the repository. Next, we augment an existing technique for detecting structurally similar subtrees in a larger tree to detect structurally similar code fragments in ASTs corresponding to consecutive versions. At the core of our technique, we have a user-specified similarity metric which combines semantic and structural information in order to tell what is constant and what has changed. Finally, we propose a visualization method to present the code structure evolution and emphasize both constant and changing patterns.

This paper is structured as follows. In Section 2, we review related work on detecting similar (evolving) code patterns. Section 3 details our similar code pattern detection method. Section 4 presents our visualization method for code structure evolution. Sec-

tion 5 demonstrates our method on a C++ code base. Section 6 discusses our method. Section 7 concludes the paper and outlines future work directions.

2. PREVIOUS WORK

Showing constant and changing patterns in evolving code inherently relates to clone detection and clone tracking methods. Given two files f_A and f_B, which can be totally unrelated or two versions of the same file f, clone detection methods find a set of code patterns $c_i^A \in f_A$ which closely resemble a set of patterns $c_j^B \in f_B$. Several such methods exist. Baxter *et al.* extract abstract syntax trees from the code, determine a hash code from the entire tree structure, and compare same-hashcode trees using a bottom-up matching algorithm [6]. Jiang *et al.* compute fixed-length vector descriptors of syntax tree nodes, recording the number of occurrences of each node type, and hash similar subtrees based on the Euclidean distance between vectors [11]. Koschke *et al.* use a suffix token tree approach, comparing syntax trees by serializing the tree node types to strings, thereby combining the speed of string approaches with the precision of tree-based approaches [12]. Wahler *et al.* use an XML-based syntax trees and database queries to find code clones as frequent item-sets [19]. Ducasse *et al.* advocate a string-based clone detection, thereby removing the need for heavyweight parsers [9]. Recently, Ekoko and Robillard proposed a method to track code clones across several versions of a code base, by reusing the SimScan clone detector atop of a lightweight clone representation combining structural and lexical clone information [8]. Clone detection methods are also implemented in widely-used clone detection software, such as the well-known *CCfinder* tool.

On a different track, several methods have been proposed to visualize code evolution patterns, at code line level [18, 13], intermediate (group-of-lines) level [1], and file and component level [14, 16].

3. TRACKING SIMILAR CODE

We are interested in detecting code evolution patterns at several levels of detail, ranging from an identifier and expression up to a whole function, class, or namespace. In the following, we consider N versions of a given source code file $f_1, ..., f_N$. We proceed as follows.

For each version f_i, we extract its complete annotated syntax tree (AST) T_i using a `gcc`-based C++ parser, following the approach described in [13]. Next, we compare all successive trees T_i with T_{i+1} and identify similar subtrees. For this, we extend the method of Auber *et al.* to detect structurally similar subtrees [4, 7], by adding semantic (type and code text) information extracted by our parser. This allows us to flexibly specify what we consider 'similar' and/or 'different' during the code evolution.

Our method consists of two steps. In the first step, we group all extracted AST nodes into equivalence classes, based on structural and semantic information (Sec. 3.1, 3.2). Same-class nodes are likely to be similar, whereas nodes in different classes are not. In the second step, we construct explicit correspondences between similar subtrees rooted in the same class (Sec. 3.4). Finally, we visualize these correspondences (Sec. 4).

3.1 Structural node classification

In the first step, we group AST nodes from version-consecutive syntax trees T_i and T_{i+1} into *equivalence classes*. Same-class nodes are roots of potentially similar subtrees. In the following, we shall denote the subtree rooted at node u by $R(u)$. We compute equivalence classes using a distance metric $d(u, v)$ which combines structural and semantic information from $R(u)$ and $R(v)$, as follows.

Following [7], we define the structural distance component $d_{str}(u, v)$ as the difference between three topological invariant, graph-theoretic, metrics on $R(u)$ and $R(v)$: The number of direct children, or degree $\delta(u)$, of u; the size $\nu(u)$ of $R(u)$; and the Strahler number $\sigma(u)$ of $R(u)$. The first two metrics are well-known. The Strahler number is defined as follows

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

The Strahler number is used in graph theory to succinctly characterize the topological complexity (*e.g.* ramification) of a tree or a DAG. For further details, we refer the reader to the literature [15, 3].

With the above metrics δ, ν and σ, we define the structural distance $d_{str}(u, v)$ as

$$d(u,v) = (\tilde{\delta}(u) - \tilde{\delta}(v))^2 + (\tilde{\nu}(u) - \tilde{\nu}(v))^2 + (\tilde{\sigma}(u) - \tilde{\sigma}(v))^2 \quad (2)$$

where $\tilde{\delta}$ denotes the normalized version of the metric δ, *i.e.* $\tilde{\delta}(u) = \frac{\delta(u) - \delta_{min}}{\delta_{max} - \delta_{min}}$, where δ_{min} and δ_{max} are the minimal, respectively maximal values of δ over all nodes in T_i and T_{i+1}, and similarly for the metrics ν and σ.

3.2 Using type information

The distance $d_{str}(u, v)$ effectively 'hashes' structurally similar subtrees [7, 4]. However, when comparing software code, we want to consider the code semantics too. Each AST node n has a type $t(n)$, *e.g.* declaration, function, assignment, `for` statement, identifier, and so on. The C++ grammar our parser uses has approximately 150 such types. We use these types to enhance the structural similarity, as follows.

First, it is not meaningful to compare totally unrelated types, *e.g.* an arithmetic expression `a+b` with, say, a declaration `int x`, even though such constructs may have structurally similar ASTs[1]. We model this by a second distance $d_{typ}(u, v)$ which tells the difference between two node types $t(u)$ and $t(v)$. The simplest way is to test strict type equality, *i.e.* use $d_{typ}(u, v) = (t(u) == t(v))$. However, this distance is too strict. Often, one wishes to track 'fuzzier' patterns over the code evolution, *i.e.* patterns which are not exactly the same, but related, syntax types. To do this, we define several classes of type-equivalence $\mathcal{C}_i$ for C++ syntax node types, based on what one considers to be a change during evolution. A class $\mathcal{C}_i$ contains all types considered similar. Hence, the distance d_{typ} is:

$$d_{typ}(u, v) = \begin{cases} 0, & t(u) \text{ and } t(v) \text{ are in the same class } \mathcal{C}_i \\ 1, & \text{otherwise} \end{cases} \quad (3)$$

Examples of useful type-equivalence classes are: `for`, `while`, and `do` statements (if one is interested to track iterative constructs); and classes, structures, and unions (if one is interested to track aggregate types). Any classes can be defined, depending on what one considers to be 'the same' during the code evolution. In the above, a C++ `class` declaration that changes into a `struct`, but keeps the same internal structure, is considered to be the same code. We do not detail other classes here, as their exact definition depends on the task at hand, but also on the specific C++ grammar used.

[1]The ASTs of these constructs are both a tree with two leafs, *i.e.* the identifiers `a` and `b` for the expression `a+b`, and the type `int` and the declarator `x` for the declaration `int x`

The structural distance d_{str} and type distance d_{typ} yield our final distance $d(u, v)$:

$$d(u, v) = [1 + (w(u) + w(v))d_{typ}(u, v)]\, d_{str}(u, v) \qquad (4)$$

In Eqn 4, $w(u)$ is a real-valued *weight*, or importance, of the type $t(u)$ of a node u. If the types of the compared nodes u and v are in the same type-class, then $d(u, v)$ equals the structural distance $d_{str}(u, v)$. If not, *i.e.* $d_{str}(u, v) = 1$ then $d(u, v)$ is larger if the types $t(u)$ and $t(v)$ are considered to be more important. The importance weight is useful to emphasize certain types of changes during the code evolution. For example, if we are interested in interface evolution, then everything having to do with global declaration node types (*e.g.* functions, classes, etc) will be important, whereas implementation-related node types (*e.g.* expressions, iterative statements, assignments, etc), are irrelevant. We can achieve this by setting a high weight w for nodes of the former types and a low weight w for nodes of the latter types. For a neutral standpoint, all weights w for all AST node types are set to 1.

Note that the structural distance d_{str} and the type importance w could have been merged in a single function. However, this means defining a N by N matrix of real values, where N is the number of C++ node types. For $N = 150$, this is impractical. By decoupling the two concerns as discussed above, we use just two N-valued vectors: a set of disjoint type equivalence classes $\mathcal{C}_i$, and a real-valued vector of type weights w.

3.3 Labeling nodes

We use the distance $d(u, v)$ from Eqn. 4 to distribute the nodes in T_i and T_{i+1} in equivalence classes, as stated at the begin of Sec. 3.1. Each node n is labeled by its integer class-id $\lambda(n)$, following the algorithm in Fig. 1. Essentially, all nodes which are closer than a user threshold ε_d are put in the same class λ. The concrete values for ε_d depend, of course, on the range of the distance metric d (Eqn. 4), which further depends on the type weights w. If we normalize d between 0 and 1, we found that good values for ε_d are in the range of 0.005 to 0.01, which put only strongly similar nodes in the same class.

Clearly, the results may depend on the order the nodes are considered in, since not all distance pairs are computed. However, as discussed in [7], the structure-and-type distance function d (Eqn. 4) is strongly discriminative, so in practice there are just few cases when many nodes have similar distances to each other. A heuristic that further improves the result is to consider nodes sorted in the decreasing order of their Strahler metric σ (Eqn. 1) (line 3 in Fig. 1). This ensures that larger, more complex and thus more interesting, code fragments are classified first, thereby limiting the impact of the order nodes are treated in to the less interesting code fragments.

```
1  int L := 1;
2  S := T_i ⋃ T_{i+1};
3  sort S on decreasing σ;
4  while (S = {n_0, ..., n_k} not empty)
5  {
6    u := n_0; S := S \ {u};
7    λ(u) := L;
8    for(i:=1; i ≤ k; i++)
9      if (d(u, n_1) < ε_d)
10     { λ(s_i) := L; S := S \ {s_i}; }
11   L := L + 1;
12 }
13 //We have now L equivalence classes 1, ..., L
```

Listing 1: Node labeling algorithm

3.4 Correspondence construction

The first phase of our algorithm discussed so far distributes AST nodes from each two consecutive versions i and $i + 1$ into equivalence classes (Sec. 3.1). The first phase can be seen as a hashing which groups similar subtrees into the same equivalence class. Yet, after the first phase, we do not know for sure which of the subtrees in the same class are truly similar. The second phase performs an in-depth analysis which finds *and* marks those parts of all subtrees in an equivalence class which are indeed similar.

The correspondence construction works recursively top-down. For two nodes u and v in the same equivalence class, *i.e.* $\lambda(u) = \lambda(v)$, denote $F(u)$ and $F(v)$ the sets of direct children of u and v in T_i and T_{i+1} respectively. For a given equivalence class $l \in [1, L]$, denote $F_l(u)$ all children of u being in class l, *i.e.* having $\lambda(F_l(u)) = l$, and similarly $F_l(v)$ for v's children. We say that u and v are *D-similar* if

$$D(u, v) = \sum_{l=1}^{L} \text{abs}\left(|(F_l(u))| - |F_l(v))|\right) < \varepsilon_D \qquad (5)$$

Here, $|F|$ denotes the size of set F and 'abs' the absolute value. $D(u, v)$ can be seen as a similarity metric based on the direct children of two nodes in the same equivalence class. If we normalize D by the total number of children of a node, good values for ε_D are around 0.1.

The final step is to extend the similarity metric D, defined on the children of u and v, to a metric Δ on the full subtrees $R(u)$ and $R(v)$, as follows. If u and v are not D-similar, then we say $R(u)$ and $R(v)$ are also not Δ-similar. If u and v are D-similar, then we recursively compare the subtrees rooted by their children within the same equivalence class $F_l(u)$ and $F_l(v)$, for all classes $l \in [1, L]$. Denote the number of such subtrees which are found Δ-similar by $\kappa(u) = \kappa(v)$. We say u and v are Δ-similar if

$$\Delta(u, v) = \frac{\kappa(u) + \kappa(v)}{|R(u)| + |R(v)|} < \varepsilon_\Delta \qquad (6)$$

i.e. if we found within the trees of u and v a relative fraction of at least ε_Δ Δ-similar subtrees. The recursion ends when we compare two leafs. These are always Δ-similar, and always have $\kappa = 1$. In our experiments, we used $\varepsilon_\Delta = 0.8$. Higher values imply a stricter matching, the maximum being 1, when a strict identity is required. Lower values imply that less similar subtrees are considered to match.

The complete correspondence computation algorithm for two trees T_i and T_{i+1} is shown in listings 2 and 3. For all equivalence classes l, we check if the node pairs (s, s') in the class l are Δ-similar (Fig. 2). We only check trees rooted in the same class, since only these have a fair chance to be structurally similar (Sec. 3.1). The function `deltaSim` checks the Δ-similarity. If two Δ-similar subtrees are found, then all their nodes are marked as being in the same pattern. We store the values of $\kappa(u)$, which store for each node u the number of Δ-similar subtrees rooted at u's descendants, in an auxiliary field. $\kappa(u)$ is set to zero before each tree-pair comparison (line 10, listing 2) and updated in bottom-up order during the correspondence construction, following Eqn. 6 (line 27, listing 3).

A simple heuristic that improves the chance of finding good correspondences is as follows. When computing the Δ-similarity, we first sort the children F_l and F'_l of the current nodes n and n' under scrutiny before matching their subtrees (lines 13+14, listing 3). We sort nodes in order of their C++ node types, and in order of their code text for identical types. This ensures that nodes as similar as possible are matched first.

```
int α := 0;
for(n ∈ T_i ⋃ T_{i+1}) α(n) := 0;

for(l := 1; l < L; l++)
{
  S_l := {n ∈ T_i |λ(n) = l}
  S'_l := {n ∈ T_{i+1}|λ(n) = l}
  for((s, s') ∈ S_l × S'_l)
  {
    for(n ∈ R(s) ⋃ R(s')) κ(n) := 0;
    if(deltaSim(s, s', ε_D, ε_Δ))
    {
      α := α + 1;
      for(n ∈ R(s)) α(n) := α
      for(n ∈ R(s')) α(n) := α
      S_l := S_l \ {n ∈ S_l|α(n) = α}
      S'_l := S'_l \ {n ∈ S'_l|α(n) = α}
    }
}
```

Listing 2: Correspondence construction algorithm

The result of the entire algorithm is a labeling α of all nodes in the input trees T_i and T_{i+1}. A value $\alpha(n) = 0$ means that node n is not in a pattern which has a correspondence in the other tree. A value $\alpha(n) > 0$ means that node n is in the pattern with id $\alpha(n)$. Finally, by applying the algorithm on all tree pairs (T_i, T_{i+1}), we obtain the evolution patterns of all code fragments on the entire version set considered.

```
bool deltaSim(Node n, Node n', float ε_D, float ε_Δ)
{
  if(n and n' are leafs)
  { κ(n) := 1; κ(n') := 1; return true; }

  F := direct children of n
  F' := direct children of n'

  if(D(n, n') ≥ ε_D) return false;

  for(l := 1; l < L; l++)
  {
    F_l := {s ∈ F |λ(s) = l}; sort(F_l);
    F'_l := {s ∈ F'|λ(s) = l}; sort(F'_l);

    for((f, f') ∈ F_l × F'_l) in sorted order
      if (deltaSim(f, f', ε_D, ε_Δ))
      {
        F_l := F_l \ {f}
        F'_l := F'_l \ {f'}
      }
  }

  κ(n) := 1 + Σ_{u∈F} κ(u);
  κ(n') := 1 + Σ_{u∈F'} κ(u);

  return (κ(n)+κ(n')) / (|R(n)|+|R(n)'|) > ε_Δ;
}
```

Listing 3: Δ-similarity computation algorithm

4. VISUALIZATION

After computing the correspondences between code fragments, represented as AST subtrees, of a number of versions, we visualize these. The goal is to visually emphasize important evolution events, such as code refactoring, code drift, or interface and implementation changes.

4.1 Basic design

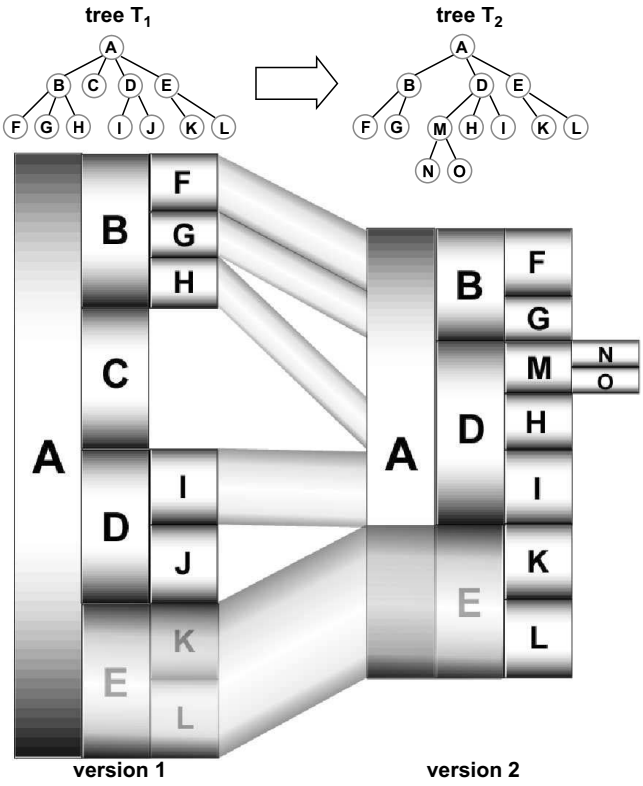

Figure 1: Visualizing two code versions and their syntax trees T_1 and T_2. Corresponding nodes are labeled with the same letters. In the evolution, C and J are deleted, the subtree M, N, O is inserted, and H moves from parent B to D. Shaded cushions emphasize both tree structure and correspondences.

Our visualization uses the design sketched in Fig. 1. Each syntax tree T_i for each version i in the evolution of a considered code file is drawn using a space-filling technique similar with so-called icicle plot [5]. A tree appears as a vertical strip, with the root to the left and the leafs to the right. Figure 2 shows the principle for a syntax tree for a small function. The syntax tree is shown using a classical tree layout (a) and using our space-filling visualization (b). ASTs of typical C++ source files of thousands of lines are very broad (tens of thousands of nodes) but not too deep (10..40 levels). Hence, trees appear as thin, tall vertical strips. The varying syntax tree depth (not to be mismatched for the code indentation depth) is shown by the strip's width (see Figs. 3 and 4 for actual snapshots). Wider strip portions indicate deeper, thus typically more complex, syntactic constructs, while thin strip portions show constructs which are shallow nested, *i.e.* close to the C++ global file scope. Additionally, we use shaded cushions as an effective means of emphasizing the structure, similar to other software visualization applications [18, 10]. The cushions, implemented as one-dimensional textures storing a parabolic luminance profile, are blended atop of the rectangular nodes, an operation which is very efficiently done in OpenGL. The entire visualization is implemented using the Tulip graph visualization system which offers sophisticated mechanisms for customized layout, rendering, interactive navigation, and level-

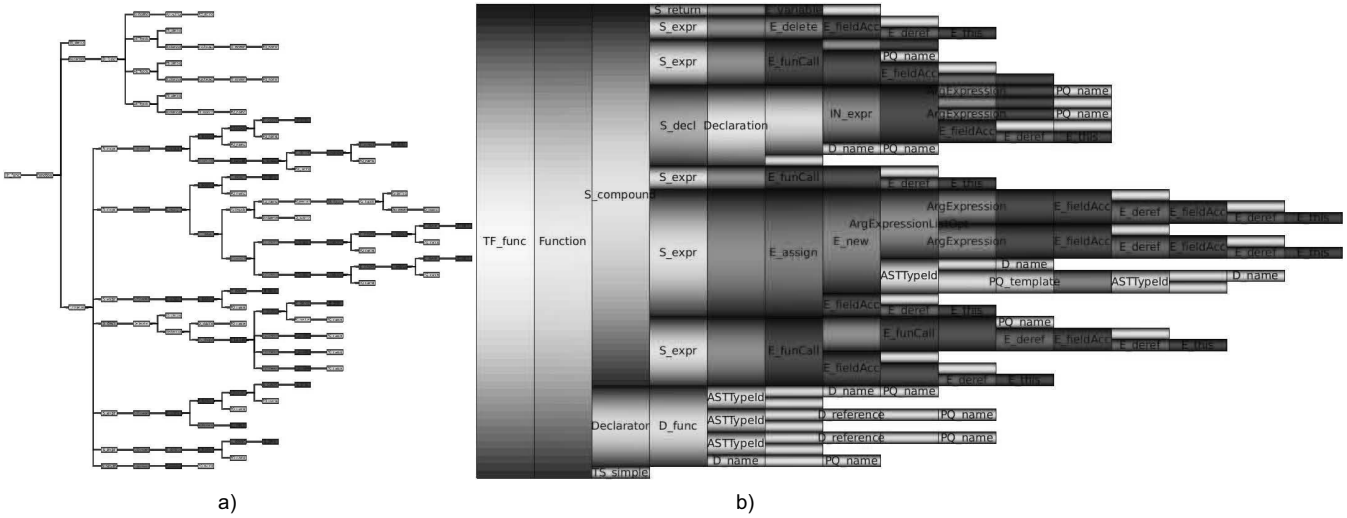

Figure 2: Syntax tree visualized in classical manner (a) and using shaded cushions (b). Labels indicate the syntax node types.

of-detail techniques for large trees and graphs up to hundreds of thousands of nodes [2].

After drawing all trees for all considered file versions, we draw the correspondences. As explained in Sec. 3, given N trees $T_1 ..., T_N$ for the file versions $f_1, ..., f_N$, we compute only correspondences between subtrees in consecutive versions T_k and T_{k+1}. For each correspondence, we connect the rectangles of the corresponding (matched) nodes from the neighboring tree strips using tube-like shapes, as shown in Fig. 1. The tubes are easy to interpret. Straight, horizontal tubes show code regions which stay constant during the evolution. This is so because the extracted syntax trees are ordered, so constant code corresponds to constant subtrees at the same place. Inclined tubes indicate code fragments which change places in the syntax tree. Crossing tubes indicate code block swapping. Since the top-to-bottom order of nodes in the tree view matches the order of code fragments in a program listing, long tubes indicate fragments which moved considerably during the evolution. Inserted and deleted fragments appear as regions in the tree which are not connected with tubes. Finally, the tubes' thickness indicate the amount of code in the corresponding pattern.

Tubes can be drawn in two different ways. In the first way, shown for nodes $F, ..., I$ in Fig. 1, tubes are drawn from the right edge of the node in version i to the left edge of the *parent* of its corresponding node in version $i+1$. In the second way, shown for node E, the tube is drawn to cover the two matched nodes and all their children (K, L), using a constant transparency value. We found the second method better as it clearly indicates node grouping in patterns. The second way for drawing the tubes has a more subtle advantage. If we look at the two nodes marked H in Fig. 1, the tube connecting them seems to end too low at its right end. Actually, this is a rather surprising optical illusion due to the shaded cushions. The second way for drawing the tubes, using superimposed cushions, corrects this problem.

Figure 3 shows the visualization design on two versions of a file of approximately 850 C++ lines. In the left image, no cushions are used. The nodes are colored with hues that indicate their C++ construct types, using different user-selected hues for classes, functions, methods, expressions, declarations, iterative statements, and conditional statements. The tubes are colored using a slightly darker hue of the correspondent nodes they connect[2]. The right image shows the same dataset. This time, both nodes and tubes are colored using a different hue for each identified pattern α (Sec. 3.4). Tubes are drawn using the transparency design explained earlier in this section. Deleted or inserted, thus unpaired, code fragments are colored in gray.

Figure 3 b clearly shows several evolution events: Fragment C is deleted, except a tiny code piece indicated by the red filament in the middle. Fragments A and B are swapped in an unchanged state. Fragments D and E stay constant.

We see here a limitation of our method. Although fragments D and E stay constant, they are identified (and drawn) as a multitude of blocks instead of one single block, as in the case of A or B. The reason is that A and B are single subtrees in the syntax tree, whereas D and E consist of sets of subtrees which do not share a constant parent. Actually, A and B are two global C++ scope methods, while D and E are sets of sibling nodes representing *several* same-scope declarations. Since our method only matches single subtree roots, some amount of fragmentation occurs. Note that, if desired, this can be easily removed by applying a postprocessing pass that groups sets of contiguous matched sibling nodes into the same pattern. We preferred to show the individual node correspondences, as these represent actual syntactic code fragments.

4.2 Visual enhancements

Several enhancements can be added atop of the above basic visualization. First, the user can choose the *level of detail* at which the syntax trees are shown. Given a level of detail, only syntax nodes up to that level, and associated correspondences, are drawn. This lets one simplify the visualization by removing fine code details which appear as those vertical strips at the right of the version visualizations (*e.g.* nodes $F, G, ..., L$ in version 2 in Fig. 1). A similar effect can be achieved by adapting the structural distance function d_{str} (Sec. 3.1). The difference is that changing the distance function potentially determines finding different evolution patterns, whereas culling finer detail from the visualization simply shows less correspondences from an already computed set.

[2]We strongly recommend viewing these figures in full color. See `http://www.win.tue.nl/~alext/CPPDIFF`

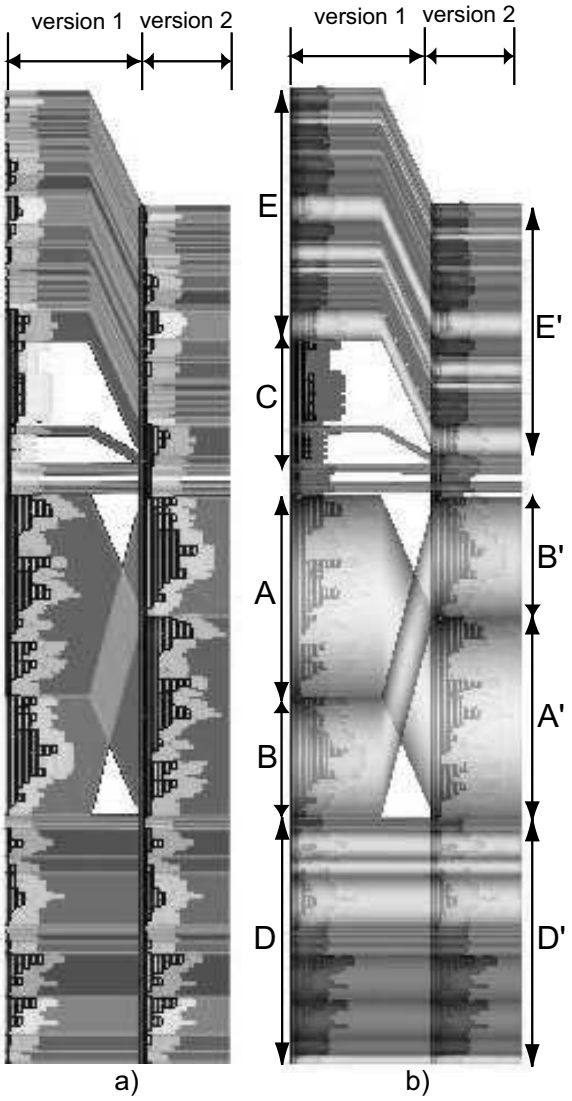

Figure 3: Two versions of a C++ file. We notice code deletion (C), block swapping (A,B), and constant code D.

5. EXAMPLE APPLICATION

We applied our structural code evolution visualization on several versioned C++ code bases. Due to space limitations, we present here just one example. The considered versioned code base is a complete C++ image processing program of approximately 5000 lines of code. From discussions with the developers, we know the versions correspond to successive small-scale system refactorings. After parsing each version, we keep in its AST all nodes originating from application code sources and headers, but eliminate nodes coming from system headers, as we are not interested to see changes at that level (if any). Figure 4 a shows a visualization of six versions of one file containing about 1500 lines from our C++ code base. Node colors indicate syntax construct types. Edges have the colors of the matched nodes they connect.

We quickly recognize a few zones in the code where important changes occurred (marked b-f in Fig. 4). For the rest of the code, the skewed parallel tubes running left-to-right in the overview image (Fig. 4 a) indicate constant or near-constant code regions. Overall, we see that the code gradually shrunk during evolution to about 80% in version 6 as compared to version 1.

Let us now concentrate on the detail views (Fig. 4 b-f). Figure 4 b shows two deletions that occurred when passing from version 1 to version 2. The top one is a `for` loop (approx. 30 code lines), the bottom one is a set of local variable declarations (10 code lines). The tube which separates the two deleted blocks corresponds to a code block which stays indeed unchanged, located between the deleted fragments. Figure 4 c shows a more complex modification pattern involving two functions `f1` and `f2`. The matching has detected a scope block `{...}` in `f1` and a large `if` statement in `f2` respectively which are constant. We also notice a small code block which moved from `f1` to `f2`. Moreover, we see several small code blocks at the lower end of `f2` and also just below the `if` block in `f2` which are matched. However, we see no tubes connecting the whole extents of `f1` and `f2` in the two versions: Although several blocks within are matched, the two functions are found too different to be set in correspondence.

Figure 4 d shows a function `f` which is split into two functions `g` and `h`. Most of `f`'s code from version 2 moves into `g` in version 3. However, the code indicated by the downward running diagonal tube and the thin horizontal tubes at the bottom of `f` moves into `h`. Figure 4 e shows a class `c` which undergoes several modifications from version 3 to 4. The thin downward running diagonal tubes indicate class members which stay the same. However, just as in the case of the functions `f1` and `f2` in Fig. 4 c, the entire class is not matched because there are too many additional changes, *e.g.* several newly inserted methods and data members. In the same image, we see also a template class `t` which undergoes massive code deletion when going from version 3 to 4. Finally, Figure 4 f shows the evolution of a part of a function implementation `F` which undergoes several low-level changes, such as insertion and deletion of local variables, changes of terms in arithmetic expressions, and replacement of `(a<b)? a:b` expressions by `if-else` statements. Although such changes are quite intricate, they are reliably found by the matching algorithm and shown by the visualization method.

6. DISCUSSION

To validate results, we applied our complete method on different code files extracted from different C++ projects, and checked the detected evolution patterns by manually inspecting the code changes, version by version, using a standard text editor. Conversely, we took a given C++ project and manually performed editing operations, ranging from simple ones, *e.g.* text reformatting, comment changes, identifier renaming, and declaration order changing) to complex ones, *e.g.* factoring out fragments of a large function into a separate function, code fragment swapping, random deletions and insertions of code fragments ranging between 1 and 50 lines, and changing `(a<b)? a:b` expressions into `if-else` statements, to mention just a few examples. It surprised us that the correspondence computation algorithm (Sec. 3) worked perfectly in all examined cases, *i.e.* found identical fragments in consecutive versions, even in presence of complex changes. By changing the ε_d,ε_D and ε_Δ parameters, we could filter out different types of low-level, less important, changes, such as minute modifications of expressions. This is useful when one is interested in larger-scale evolution patterns, such as changes of signatures of global objects or compositions of class declarations.

Our method has several conceptual similarities with existing code clone detection techniques. First of all, our method is insensitive to any lexical, code formatting, comments, and identifier name changes. Many clone detectors do not exhibit such properties, as they do not generate a parse tree but work at a lower, lexical level.

The closest method to ours seems to be that of Baxter *et al.*, who also extract and hash syntax nodes prior to tree comparison in their clone detection [6]. While they use only the node types in their hash function, we use a combination of structural parameters (node degree, tree size, and Strahler number) and node types (Sec. 3.1, 3.2). Our hash function seems to be weaker than the one in [6]. We actually cannot tell this for sure as [6] does not include full implementation details. This allows our method match subtrees more freely, a step which Baxter *et al.* solve by adding a separate "clone sequence" processing phase. Jiang *et al.* compute one fixed-length vector of type occurrences per subtree and use locally sensitive hashing (LSH) to group similar vectors, which is conceptually similar but differently implemented from our structure+type hashing. Their Euclidean distance used to compare vectors is, again, conceptually similar to our distance function d_{typ} and importance weights (Sec. 3.2), but has a quite different implementation.

Let us stress again that our aim is to identify small up to medium-scale evolution patterns in consecutive file versions, which is a very specific case of finding similar code fragments in files of moderate size (up to thousands of lines/file) exhibiting a moderate amount of change. While our method *could* be in principle used to detect and visualize code clones, this was not our purpose. Code cloning detectors stress scalability, performance, detection of certain pattern types, and do not typically focus on visualizing the detected correspondences. Whether our method can be further optimized and/or parameterized in the context of clone detection is a subject of future research.

Considering performance: The matching of six versions of a code base of approximately 5000 code lines per version took about 12 minutes on a 2 GHz PC running Linux. We are aware that our current implementation could be further optimized to yield the higher speed needed to analyze large code bases. However, since our target scenario is to interactively visualize *small-scale*, minute code modifications, typically at the level of a single file across several versions, we are less constrained by high performance requirements as *e.g.* code clone detection methods that run in automatic mode on full industry-size repositories.

7. CONCLUSIONS

We have presented a method to compute and visualize evolution patterns in C++ source code. At the center of our method is a structure-and-type matching technique running on the abstract syntax trees of consecutive versions. The matched subtrees are visualized using dense pixel layout and rendering methods. Our method is useful to detect and browse small to medium-scale changes during source code evolution, such as function and class-level refactoring code editing. Hence, our work fills the gap between line-level evolution analysis tools, like WinDiff and CVSscan, and file and architecture-level tools, like sv3D and CVSgrab.

We envisage several directions of our work. First of all, the presented visualization is just a first attempt to display the syntactic correspondences. Different variants of the basic idea should be tried out, such as leaving out the unchanged code fragments and showing only the added, modified, and deleted ones. Second, different distance functions can be designed to incorporate additional information beyond syntax tree structure and types, *e.g.* project or user-specific data like variable naming conventions, to detect different types of code evolution events of interest for specific refactoring tasks and filter out others. Third, the proposed evolution visualization can be enhanced to scale to show more versions of larger amounts of code, as well as more code attributes than just construct type. Finally, both the similar code detection and the visualization method may have the potential to be used in the context of finding and understanding software code clones.

8. REFERENCES

[1] E. Adar and M. Kim. SoftGUESS: Visualization and exploration of code clones in context. In *Proc. ICSE*, 2007.

[2] D. Auber. Tulip : A huge graph visualisation framework. In *Graph Drawing Software - Mathematics and Visualization*, pages 105–126. Springer, 2003.

[3] D. Auber, M. Delest, J. P. Domenger, P. Duchon, and J. M. Fédou. New strahler numbers for rooted plane trees. In *Coll. Math. & Comp. Sci. Algorithms*, pages 203–215. Birkhauser, 2004.

[4] D. Auber, M. Delest, J.-P. Domenger, and S. Dulucq. Efficient drawing of rna secondary structure. *J. Graph Algorithms and Applications*, 10(2):329–351, 2006.

[5] T. Barlow and P. Neville. A comparison of 2D visualizations of hierarchies. In *Proc. IEEE InfoVis*, pages 131–139, 2001.

[6] I. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. Clone detection using abstract syntax trees. In *Proc. ICSM*, pages 368–377, 1998.

[7] F. Chevalier, M. Delest, and J. P. Domenger. A heuristic for the retrieval of objects in low resolution video. In *Proc. Intl. Workshop on Content-Based Multimedia Indexing (CBMI)*. IEEE Press, 2007.

[8] E. Duala-Ekoko and M. Robillard. Tracking code clones in evolving software. In *Proc. ICSE*, 2007.

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

[10] D. H. R. Holten. Hierarchical edge bundles: Visualization of adjacency relations in hierarchical data. *IEEE Transactions on Visualization and Computer Graphics (Proc. InfoVis 2006)*, 12(5):741–748, 2006.

[11] L. Jiang, G. Miserghi, Z. Su, and S. Glondu. Deckard: Scalable and accurate tree-based detection of code clones. In *Proc. ICSE*, 2007.

[12] R. Koschke, R. Falke, and P. Frenzel. Clone detection using abstract syntax suffix trees. In *Proc. WCRE*, pages 253–262, 2006.

[13] G. Lommerse, F. Nossin, L. Voinea, and A. Telea. The visual code navigator: An interactive toolset for source code investigation. In *Proc. IEEE InfoVis*, pages 4–11, 2005.

[14] A. Marcus, D. Comorski, and A. Sergeyev. Supporting the evolution fo a software visualization tool through usability studies. In *Proc. IWPC*, pages 307–316, 2005.

[15] A. Strahler. Hypsomic analysis of erosional topography. *Bulletin of Geol. Soc. of America*, 63:1117–1142, 1952.

[16] L. Voinea and A. Telea. Cvsgrab: Mining the history of large software projects. In *Proc. EuroVis*, pages 187–194, 2006.

[17] L. Voinea and A. Telea. Visual data mining and analysis of software repositories. *Computers & Graphics, DOI 10.1016/j.cag.2007.01.031*, 2007.

[18] L. Voinea, A. Telea, and J. J. van Wijk. Cvsscan: Visualization of code evolution. In *Proc. ACM SoftVis*, pages 47–56, 2005.

[19] V. Wahler, D. Seipel, J. W. v. Gudenberg, and G. Fischer. Clone detection in source code by frequent itemset techniques. In *Proc. SCAM*, pages 128–135, 2004.

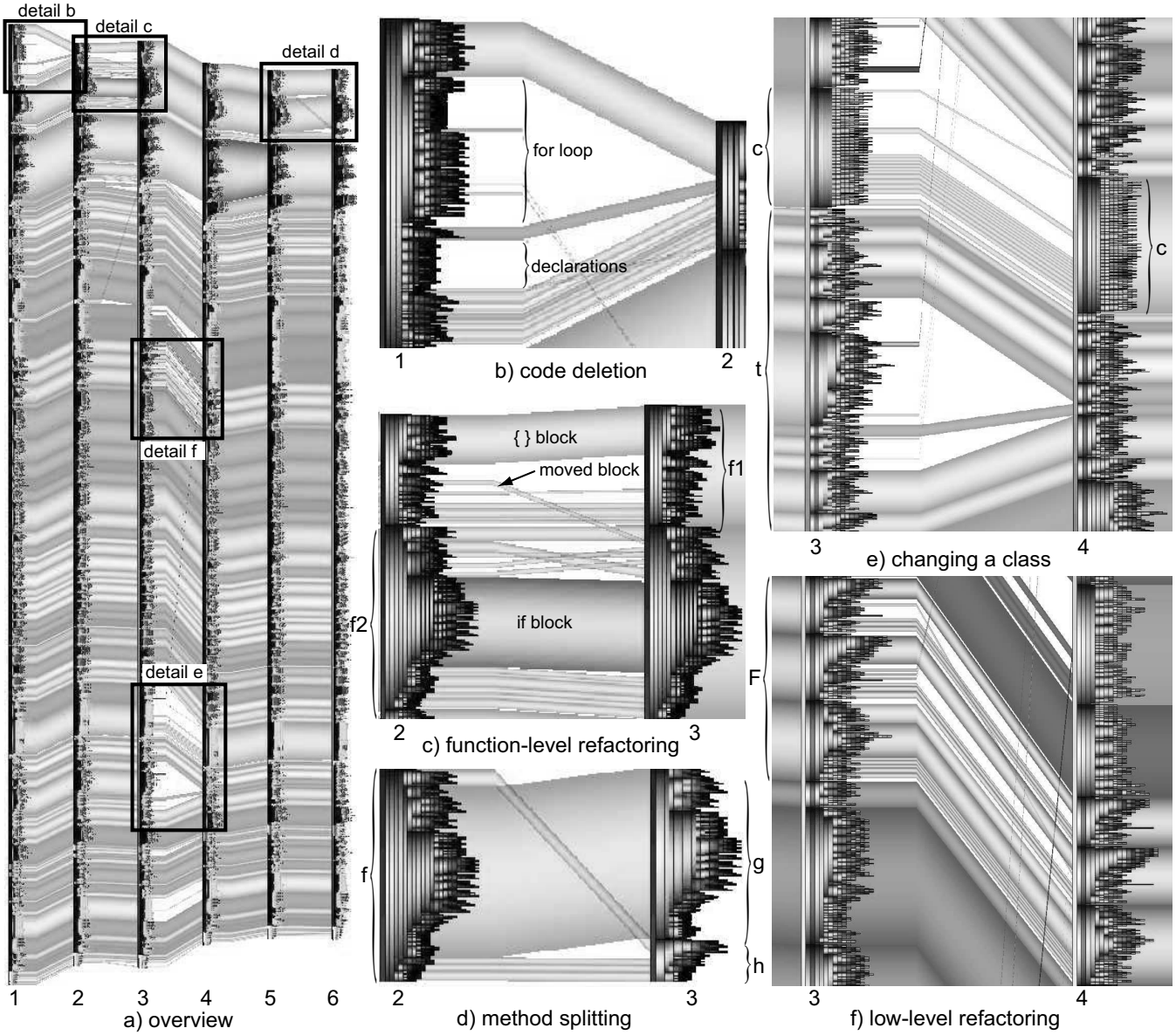

Figure 4: Visualization of six versions of a C++ software system. a) overview; b-f) details. See the individual captions and text for more details.