Multivariate Graphs in Software Engineering

S. Diehl[1] and A. Telea[2]

[1] Department of Computer Science, University of Trier, Germany
[2] Institute Johann Bernoulli, University of Groningen, the Netherlands

Abstract. Multivariate networks, or graphs, are an essential element of various activities in the software engineering domain, such as program comprehension for software maintenance and evolution. In this chapter, we present the specific context in which multivariate graphs occur in software engineering, highlight their importance in domain-specific tasks, and survey several visualization solutions designed for such graphs in the software engineering field.

1 Introduction

Multivariate networks, or graphs, occur in many application domains. In this chapter, we focus on software engineering. We present the specific nature of the data, challenges, and visual exploration solutions for multivariate graphs stemming from software engineering applications. Our goal is twofold. First, we draw attention to specific software engineering aspects, and the ensuing multivariate graphs, which make their (visual) understanding hard. This should help researchers to better understand the software engineering challenges related to multivariate graphs, and thus contribute to solutions. Secondly, we present existing approaches for the visual exploration of multivariate software graphs. This should help disseminating such solutions to areas beyond software engineering.

The structure of this chapter is as follows. In section 2, we outline the importance and scope of software visualization. Section 3 details the characteristics of the data involved in such visualizations and the scope of multivariate graphs herein. Section 4 presents a selection of relevant tasks addressed by software visualization which involves multivariate graphs, and also presents visualizations that address such tasks. Section 5 discusses the current state-of-the-art in multivariate visualization of software networks, and outlines the main challenges that this application domain currently faces.

2 Aims and Scope

To understand the specific challenges (and existing solutions) to the visualization of multivariate software graphs, we first need to understand the main aims and scope of software visualization (SoftVis). In this section, we provide an overview answer to this question. Given the huge scope of software engineering and, implicitly, SoftVis techniques and tools, we cannot aim at a complete review. Rather,

the aim is to outline the key value drivers that make SoftVis relevant to software engineering, and also to highlight how software visualization (with a focus on multivariate graphs) differs from other multivariate graph visualizations. For a comprehensive survey of software visualization, we refer to [12].

2.1 History and definitions

Early examples of software visualizations include the visual depiction of program control flow charts [16, 35], sorting algorithms [2], and software source code [13]. In the 1990s, software visualization was being recognized as a separate research field. One of its first definitions is as follows: "Software visualization is a representation of computer *programs*, associated *documentation* and *data*, that enhances, simplifies and clarifies the *mental representation* the software engineer has of the operation of a computer system" [38]. We see that SoftVis covers the full range of data artifacts produced by the software lifecycle. Equally importantly, we see that the key aim of SoftVis is to help software engineers to *understand* the operation of software systems. These aspects have stayed relevant throughout the history of software visualization, as further discussed.

A decade later (2007), a comprehensive survey of software visualization [12] proposed the following definition: "Software visualization targets the visual depiction of the structure, behavior, and evolution of software". The definitions of these three key data ingredients of software are as follows:

1. **Structure:** Describes all entities involved in the studied software, including their properties and relations between them;
2. **Behavior:** Describes how entities dynamically interact with each other, and also process data, during program execution;
3. **Evolution:** Describes how software is changed during the software lifecycle.

The reach of software visualization to software evolution parallels the growing interest in developing models, techniques, and tools for the data mining and analysis of software evolution processes [25]. In parallel, the audience of SoftVis is also enlarged, to include almost all stakeholders of the software lifecycle: product and process managers, architects, designers, developers, and testers [22].

2.2 Importance

To better advocate the necessity and added value of SoftVis, we consider the question: Is software visualization really needed? Answering this question has two parts. First, its application domain, the software industry, is large and growing: $457 billion for 2013, 50% larger than in 2008 [20]. For comparison, the total US healthcare spending in 2009 was $2.5 trillion [46]. Studies over two decades show that 80% of software development costs are spent on maintenance [37, 10, 25]. Secondly, over the same period of time, several studies have shown that over 50% of the effort spent by software engineers is dedicated to *understanding* the software [22, 12, 25]. As modern software systems become even larger, this understanding effort becomes a key component of the software lifecycle [5].

3 Data characteristics

Data involved in program comprehension is large, complex, and changes in time. As such, software visualization has a good potential to be an effective part of comprehension solutions. In a survey of over 100 practitioners involved in software maintenance and re-engineering, 42% of the participants stated that SoftVis is an important, but not critical, aid to comprehension; 42% other participants found SoftVis absolutely necessary for their work [22]. A survey on SoftVis tools highlighted as added value points the increase of productivity and quality of produced software, better management of complexity in large software systems, all leading to saving time and money in development and maintenance [3].

Software can be modeled by three orthogonal data aspects: *entities*, *relations*, and *attributes*. These are detailed in the following sections.

3.1 Entities

Software entities correspond to the *nouns* in the software description, *i.e.* describe the items which interact to form the structure, behavior, and evolution of a software system. Structural entities typically describe the static organization of a software corpus. Examples are folders, files, packages, components, classes, methods, and individual lines of source code. Behavioral entities describe the execution of a software system. Examples are program traces, profiling logs, method invocations, test results, and bug reports. Evolutionary entities are, largely speaking, related to the process of software maintenance. Examples are change requests, product documentation and requirement documents, development tasks, and actors with different roles in the development process (contributors to software repositories, testers, quality engineers, and release managers).

3.2 Relations

Relations correspond to the *verbs* in the software decription, *i.e.* describe how various software entities are connected and interact to form the structure, behavior, and evolution of a software system. Structural relations can be further organized into *hierarchical* and *association* relations.

Hierarchical relations describe the static structure of a software system. They form a part-whole hierarchy that captures the aggregation of smaller-scale software entities into larger units. Examples of containment relations are

1. **physical** relations (files in folders in higher-level folders);
2. **logical** relations (methods in classes in libraries in systems).

Several such hierarchies may be needed to describe a given system. For instance, C++ programs admit both a physical file-folder hierarchy and a logical namespace-class-method hierarchy, and the two hierarchies are not identical.

Association relations cover all relations which do not describe (hierarchical) part-whole relationships. Examples are

- **calls:** Function A calls function B;
- **inheritance:** Class A inherits from class B;
- **co-change:** File f_1 changed at the same time as file f_2;
- **duplication:** Files f_1 and f_2 share similar (cloned) source code;
- **change impact:** When changing file f_1, we next need to change file f_2;
- **ownership:** Developer D performed task T; class X owns an object Y;
- **data flow:** Component C reads data from component Y.

Clearly, several types of relations are needed to describe the structure, behavior, and evolution of a software system. Association relations can form both acyclic graphs (*e.g.* inheritance) but also cyclic graphs (*e.g.* a call stack containing recursive or re-entrant functions). Associations can be either undirected (*e.g.* clone or co-change relations) or directed (*e.g.* inheritance or call). Software associations are typically one-to-many relations (*e.g.* a function calls several other functions; a data object owns a collection of subordinate objects).

3.3 Attributes

Attributes model structural, behavioral, and evolutionary *properties* of both entities and relations. Examples are

- **syntax:** name, signature, and location in the source code for classes, functions, or individual symbols;
- **execution:** call duration, call stack depth, processor allocation, and resource usage of a function call in a program trace;
- **person:** name, role, and e-mail of a person involved in a software maintenance process;
- **testing:** time stamp, number of failed and passed tests, and amount of lines of code covered by tests for a code unit.

Typically, attributes are modeled as key-value pairs for an entity or relation, *e.g.* a class C has an attribute *name* with value C. An entity or relation can have several such attributes. Also, entities (or relations) of the same kind, or type, do not necessarily need to have the same number of attributes. This is, among others, due to incomplete data delivered by the various data mining tools used.

3.4 Software as multivariate time-dependent graphs

Entities, relations, and attributes can change during the lifetime of a software product. Two causes drive this process:

1. **behavior:** Running the same software system several times can yield different execution paths and data values. Thus, both relations (calls) and attributes (call durations) for the same entity (caller function) will differ;
2. **evolution:** Software continuously changes as it is maintained. Hence, an entity (*e.g.* file) can have different contents, relations (to other files), and developers owning it over time.

Putting it all together, we can describe software as a *multivariate time-dependent graph* $G = (V, E = V \times V)$. Nodes V model software entities. Edges E model structure and association relations. Each node $n \in V$ and edge $e \in E$ has a set of attributes $\{a_i^n\}$ and $\{a_i^e\}$ respectively. Each attribute $a = (key, val)$ is a key-value pair. Keys key are typically textual or categorical identifiers. Attribute values val belong to various domains, depending on the attribute type, *e.g.* $\mathbb{N}$ for code size, $\mathbb{R}^+$ for execution duration, $\mathbb{B}$ for test outcomes (passed, failed), or Σ^* for developer names (where Σ is the used alphabet). All elements of G, *i.e.* V, E, a_i^n, and a_i^e are time-dependent, *i.e.* functions of $t \in T$. Since both software execution and software evolution are discrete processes, and also since data is mined from software systems typically at discrete points in time, T is usually a finite set of ordered points in time $T = \{t_i \in \mathbb{R}^+ | t_i < t_j, \forall i < j\}$.

3.5 Reference implementation

Hierarchy-and-association graphs G are also often called *compound* graphs in the literature [40]. Creating, storing, manipulating, and ultimately understanding the information captured by such graphs is clearly very challenging, even for moderately-sized systems. Important questions in this respect are

1. **schema:** How to best model (capture) a given aspect of a software system in terms of entities, relations, and attributes?
2. **selection:** How to select data relevant for a given task from an entire G?
3. **implementation:** How to store G in a way that is efficient for quickly reading and writing large amounts of data?

Several so-called *data schemes* or data models for G have been proposed, *e.g.* [43, 27, 14, 30, 24]. This wealth of models can be puzzling for the practitioner interested in using existing SoftVis tools and techniques on given software datasets. More importantly, not all challenges of modeling multivariate software data become evident from studying such data schemes.

To outline such challenges, we present next a data model for G. This model is based on a SQL relational database, so it is simple to understand, scalable, computationally efficient, and can generically capture most degrees of freedom of G. First, we define the concept of a *selection*. Selections $S \subset G$ are subsets of nodes and/or relations that specify the part of G that we want to study. They are a necessary abstraction when using a single graph G to store all available data, or facts, mined from a software project. For example, if we are interested in studying the call graph of a system, we need a selection containing only software-structure nodes and call relations between them. If we want to study the contribution of a given developer, we need a selection containing only entities and relations that the respective developer has worked on.

The proposed SQL schema contains the following elements (see also Fig. 1 top):

1. **Keys:** each node, hierarchical and association edge, and selection, has a unique ID (further used as primary key);

2. **Hierarchy** table: one hierarchical edge, listed as $(parent, child)$ node IDs, per row;
3. **Association** table: an association edge, listed as $(from, to)$ node IDs, per row;
4. **Node attribute** table: each row stores all attributes (metrics) $a_1^n, \ldots, a_k^n$ of a given node n as k columns;
5. **Association attribute** table: each row stores all attributes (metrics) $a_1^e, \ldots, a_k^e$ of a given association edge e as k columns; different edge types, *e.g.* calls, uses, includes, are modeled by adding an edge-type attribute;
6. Two **selection** tables per selection $s \in S$, for the node IDs and edge IDs of the items in s, respectively;

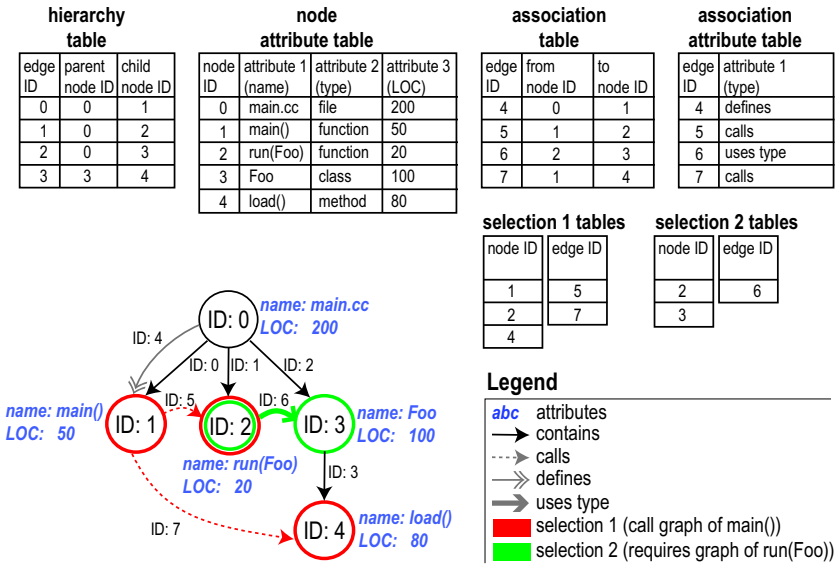

Fig. 1. Database schema (top) for a compound attributed graph (bottom) and two selections: The call graph of $main()$ and the 'uses' graph of $run(Foo)$.

Figure 1 (bottom) illustrates this schema this for a simple program. Hierarchy consists of a file *main.cc* containing two functions $main()$ and $run(Foo)$, and a class Foo with a method $load()$. Associations are call, define, and 'uses type' relations, modeled as edge 'type' attributes. Nodes have two attributes: name and lines-of-code size (LOC). Two selections exist: the call graph of $main()$ (red), and the 'uses' graph of $run(Foo)$ (green).

This schema can store any compound (hierarchy-and-association) attributed graph, *e.g.* annotated syntax graphs, call graphs, developer networks, or code duplication relations. New association types can be added to a database without changing its schema, since types are stored as attributes. This allows incrementally refining an existing *fact database e.g.* by adding new results obtained from

additional data mining processes. Adding node or relation attributes amounts to adding new columns for the node and edge attribute tables respectively. Attribute types can be any of the supported data types of the underlying SQL database (*e.g.* numeric, text, date-time, image, or binary blob). Multiple association types can be stored in a single pair of association and association-attribute tables. Hierarchy data is stored separately in a hierarchy table. This follows the observation that, in software databases, hierarchy data is much smaller in size (and typically changes less frequently) than association data. As such, this schema is more efficient for fast querying and updating. If desired, several hierarchy tables can be used to model multiple software hierarchies (Sec. 3.2).

Multiple selections can be stored in separate selection tables. This fully decouples data *storage* (node, association, and hierarchy tables) from data *usage* (*e.g.* visualization). Users can create and iteratively refine selections by executing SQL queries on already existing selections. This supports the visual information-seeking mantra "overview, filter, then details on demand" [36]. The above schema scales well to databases of millions of entities and relations [33].

However, the above schema for G does not capture time-dependency. Schemas that model time-dependent graphs have been proposed, *e.g.* [50, 30]. However, to be scalable to large software projects, such schemas are geared towards capturing specific types of relations, rather than the generic model of the graph G outlined above. A fully general solution to efficiently and effectively modeling G is not yet known, and this is a topic for future research.

3.6 Software data *vs* other Infovis domains

Large multivariate time-dependent graphs having the model outlined in Sec. 3.5 are not unique to software engineering. They arise also in other application areas, most notably biology, chemistry, and bioinformatics. As such, relevant questions are: What is specific to the software understanding domain, which underlies the evolution of software visualization as a discipline separated from biological visualization (BioVis) or, more generally, information visualization (InfoVis)?

BioVis: Comparing our software graphs to networks in the BioVis domain, we notice several similarities: In both domains, large graphs (hundreds of thousands of entities, relations, and attributes or more) are common in real-world use-cases. As such, scalability and efficiency are common concerns. Moreover, both domains feature problems involving multivariate and time-dependent graphs. However, several differences exist. First, SoftVis artifacts, and thus graphs extracted from them, are *man made*. In contrast, BioVis graphs capture (the measurement of) natural processes. In other words, SoftVis graphs are *constructive*, whereas BioVis graphs are *observational*. This deserves additional explanations. We could, on the one hand, say that SoftVis graphs are also observational, if we consider a software process as a "black box" which is monitored from the outside *e.g.* to reverse-engineer its behavior. On the other hand, software *is* constructed by humans. As such, the underlying software understanding process takes the form

of recovering (possibly lost) semantics. In contrast, understanding biological processes often aims at discovering yet-unknown natural laws and designs.

A second difference relates to uncertainty. Software processes are defined by an underlying *exact* computational model given by the processor and semantics of used programming languages. For example, there is no uncertainty as to which is the type-usage or inheritance graph of a given code base. In contrast, BioVis data typically contains more uncertainty, due to measurements and the natural variability of experiments. If, however, we add human aspects to SoftVis data, *e.g.* we want to reason about developer properties, then data uncertainty becomes more important, and this distinction gets blurred.

Last but not least, a major distinction is induced by the *user group*. In BioVis, users are typically not computer scientists themselves. As such, they are likely less familiar with various algorithmic, implementation-level, and data modeling aspects involved in the construction and usage of visualization tools. In contrast, developers and users of SoftVis tools largely overlap – they are all computer science professionals. On the one hand, this makes the task of SoftVis developers easier, as they understand both the end goals and mechanisms their tools should support and respectively provide. In contrast, developing effective BioVis tools is a much harder proposition, as their developers have to become, at some point, experts in *both* information visualization and biology.

InfoVis: Software visualization can be seen as a specialized sub-branch of information visualization (InfoVis). However, if we compare the focus of many InfoVis research projects with their SoftVis counterparts, several differences emerge. First and foremost, SoftVis 'solutions' (techniques, tools, and applications) show a strong coupling of data mining and visualization components, covering the entire pipeline from getting the raw data, filtering and analyzing this data to extract relevant information, and next exploring this information visually to (in)validate a hypothesis related to a software process or product. As such, SoftVis is closer to what is currently called visual analytics. In contrast, a significant part of InfoVis research focuses on more generic problems, such as the visualization of large *generic* graphs, tables, or hierarchies. Typical program understanding challenges involve correlating a multitude of different aspects, such as source code, execution traces, documentation, and developer activities. As such, SoftVis datasets are by nature high-variate graphs which contain attributes of a multitude of different *types*. The challenge of visually understanding multivariate data is thus fundamental to SoftVis. In contrast, multivariate data is not, by definition, a key aspect to *all* InfoVis applications and solutions.

4 Applications

In the previous section, we have shown that SoftVis datasets consist naturally of large multivariate time-dependent attributed graphs. In this section, we overview a number of techniques and tools that have been developed in the SoftVis domain for visualizing such data. We organize the presentation along the structure,

behavior, and evolution aspects introduced earlier. For each aspect and solution, we also outline the *tasks* that the respective solution aims to support (Fig. 2), and also emphasize the multivariate nature of the visualized data.

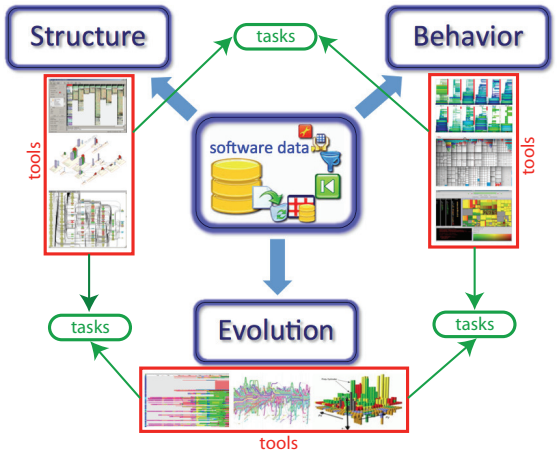

Fig. 2. Data sources, tools, and tasks in software visualization.

4.1 Structure visualization

Software has a hierarchical structure (Sec. 3.2). At the lowest level, we can visualize individual lines of code. Fig. 3 shows two examples. Both examples share the same core idea, introduced by the SeeSoft tool [13]: show each code line as a horizontal pixel line, scaled by the line's length (in characters), and colored by a data attribute computed for that line. Similar to the table lens [31], line-level visualizations scale well up to tens of thousands of code lines on a single screen. Image (a) shows the Tarantula tool [21]. Here, lines are colored by a data value indicating testing outcomes. Red lines show many failures, green lines show passed tests, and gray lines show code not covered by tests. Image (b) shows a similar design in the CSV tool [23]. Here, colors are added to syntax blocks in source code, rather than individual lines. Users can pick specific language constructs, such as functions, class declarations, iterative statements, conditional statements, variables, or comments, using a classical tree browser for the language's syntax, and assign them specific colors. Matching code blocks are displayed using these colors by the shaded cushion technique introduced by Van Wijk and Van de Wetering for treemaps [48]. The spatial cushion nesting conveys the code's nesting depth. The color distribution conveys the overall code structure. For instance, in Fig. 3b, green shows comments. We can thus see that the visualized code (around 10K lines) is densely and uniformly commented.

Line-based visualizations have a natural multiscale aspect. Zooming out, we can continuously transition from the simplified images in Fig. 3 to classical text

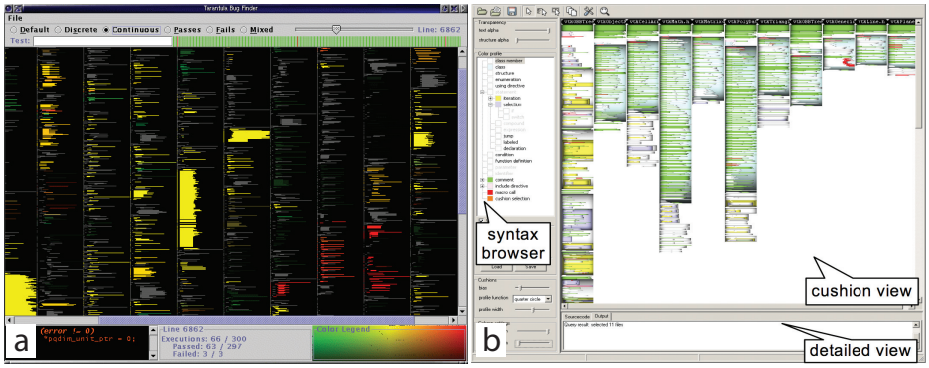

Fig. 3. Line-level (a) and syntax-level (b) visualization of program structure.

views where individual code lines are readable [23]. Several software aspects can be viewed simultaneously: structure-and-evolution (Fig. 3a), and structure-and-behavior (Fig. 3b). However, we also see several limitations. First, in the continuous transition described above, continuous *attribute interpolation* is hard to do for non-numerical attribute types. Secondly, given the limited display space, it is hard to show several attributes per item (line of code).

Structure at higher levels than code lines involves folders, files, classes, and methods (Sec. 3.2), and also their relations. Figure 4 shows several such visualizations. Image (a) shows a classical UML diagram (nodes=classes, edges=interitance, 'has' and 'uses' relations). Nodes are laid out using graph drawing algorithms such as Sugiyama-style methods or spring embedders [15]. Multiple attributes, *e.g.* code quality metrics, are shown atop of each class using glyphs (bar and pie charts) sized and colored by the metric values. Glyphs are laid out in the same (grid) order in each class. This helps correlating the same attribute across different classes. As typical UML diagrams contain only tens of nodes (classes), nodes offer enough space to show several per-class metrics. Image (b) shows how the third dimension brings an additional degree of freedom – here, glyph heights attract attention to extreme attribute values. This 3D technique is generalized in CodeCity [51], where the UML 'base' layout is replaced by a treemap to increase information density (Fig. 4e).

Additional structure can be added by considering so-called *areas of interest* (AOIs). AOIs are sets of nodes which share a common property, *e.g.* all thread-safe or all platform-dependent classes in a system [8]. Such sets can be nested, overlap, or be disjoint. AOIs can be shown with Venn-Euler diagrams, by surrounding all elements in a set by a smooth shape (Fig. 4c). Adding shaded cushions [48] helps seeing how AOIs overlap or nest. Nodes can also have *per-AOI* metrics. These are multiple attribute values that a node has, one for each AOI it belongs to – for example, the amount of thread-safe, respectively platform-dependent code lines that a class has. To visualize these attributes, space is used outside the node icons (which are reserved to show the AOI-independent node

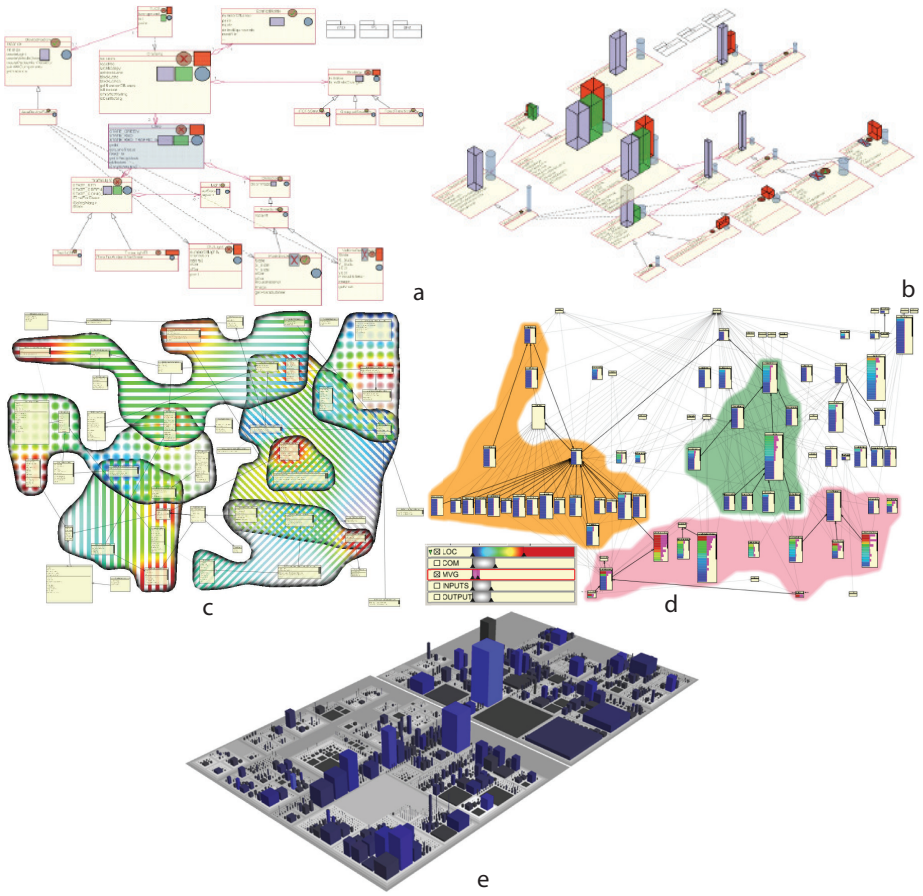

Fig. 4. Code structure and multiple code metrics shown on UML and treemap diagrams

attributes) – specifically, per-AOI attributes are drawn on the AOI cushions using texture and color interpolation. Texturing creates a weaving pattern which helps mapping the identity of an attribute to its corresponding AOI.

Detail information can be added by a table lens [31] atop each class icon (Fig. 4d). Rows are class methods, and columns show 1..3 metrics for each method. All class tables can be sorted synchronously, which allows easily comparing the metrics' distributions across an entire diagram.

Despite considerable work in the graph drawing community, classical node-link diagrams are effective only for graphs up to a few hundred nodes. Beyond this, clutter created by node-node, edge-edge, and node-edge overlaps makes reading such images hard. Also, for large graphs, the white space left between nodes by such algorithms makes their use less scalable. A different approach is taken by hierarchical *edge bundling* (HEB) methods. Pioneered by Holten [17], HEB assumes its input is a compound (hierarchy-and-association) graph. Start-

ing with a *given* node layout, HEB groups, or bundles, straight-line association edges between these nodes using the hierarchy relations. If the given node layout is compact (space-filling), then HEB can scale easily to thousands of nodes and edges on a single screen. Additional automatic level-of-detail and interaction options make HEB scale to hundreds of thousands of nodes and edges [19].

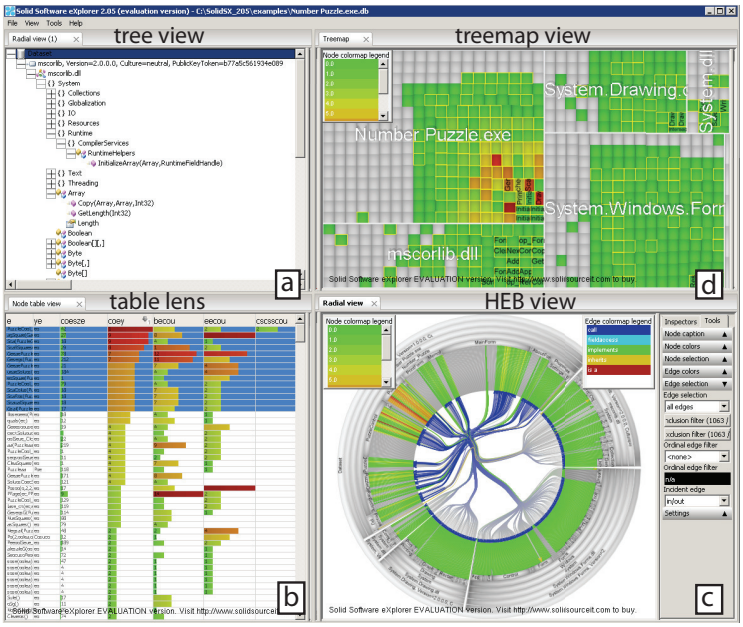

Fig. 5. Syntactic structure and attributes visualized with multiple space-filling views.

Figure 5 shows the SolidSX Software eXplorer tool [33]. The input compound graph captures the syntactic structure of a C# program (50K lines of code). Image (a) shows the program hierarchy (assemblies, classes, methods, and files) using a classical tree browser. In image (b), a table lens [31] shows several method-level attributes (name, size, complexity, number of callers, and number of callees). Sorting this table allows finding *e.g.* the most complex and/or largest methods. In image (c), these methods are highlighted atop of the software structure-and-association graph shown with HEB. Relations (calls, inheritance, and type usage) between elements are shown by bundles, colored by relation type. Finally, a fourth view, image (d), uses a treemap to show two different node attributes encoded in the treemap-cell colors and sorting order.

Several aspects are relevant here. Visual scalability is achieved by using different types of space-filling techniques: table lenses, treemaps, and HEB plots. Understanding aspects which are encoded in the correlation of *multiple* attributes is done by using multiple views linked by selection and brushing. This implicitly makes the entire solution scalable to multivariate data – the four-view display

in Fig. 5 can show, in practice, ten such attributes per data element. However, this puts an extra burden on users in terms of performing the interactive view linking. Separately, bundling creates (by construction) many edge overlaps. Although this reduces visual clutter as compared to classical node-link displays, it also makes it hard to use color-mapping to show individual attributes at edge level. Also, even for limited amounts of edge overlaps, showing *multiple* attributes per edge in the same time is not possible.

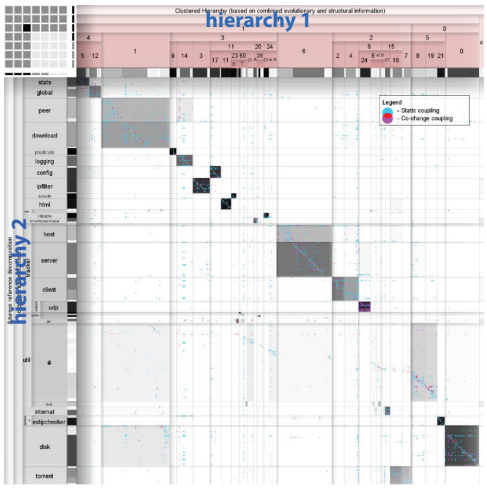

Fig. 6. Adjacency matrix visualization for comparing two software hierarchies.

Figure 6 shows a different use-case – the comparison of two hierarchies of a software system [4]. The choice of structures used supports different use-cases, *e.g.*, the comparison of the logical and physical views (Sec. 3.2) of a system, or the comparison of two related systems such as two versions of a software code base. The horizontal and vertical icicle plots show the two considered structures. The shaded cells in the central adjacency matrix indicate how entities match between the two structures. In the shown figure, these cells are quite close to the diagonal, indicating a strong similarity of the two structures. Adjacency matrix plots are a good space-filling alternative to HEB views for showing compound graphs, and have been used to visualize very large call graphs [47, 1].

4.2 Behavior visualization

Apart from program structure, behavior is an essential part of program comprehension. Captured by execution traces, behavior can be visualized using activity charts. Figure 7a shows a typical chart, produced by the Shark profiling tool on Mac OS X. Table rows correspond to function calls, sorted on calling order, call duration, or other user-specified criteria. The right table part shows per-CPU-core occupancy, color-coded by CPU code ID. This gives insight in how well

a parallel program is designed to take advantage of a multi-threaded architecture. Similar techniques are used to visualize software behavior on superscalar-processors (Rivet tool, [39]) and Java program executions (Jinsight tool, [29]). Apart from the per-function-call view, the call stack metaphor is also used to visualize execution traces (Fig. 7b). Here, an icicle plot shows function call nesting and call duration. In this view, the call stack depth, as well as time spent in a function call itself *vs* time spent in deeper-called functions, are easily visible.

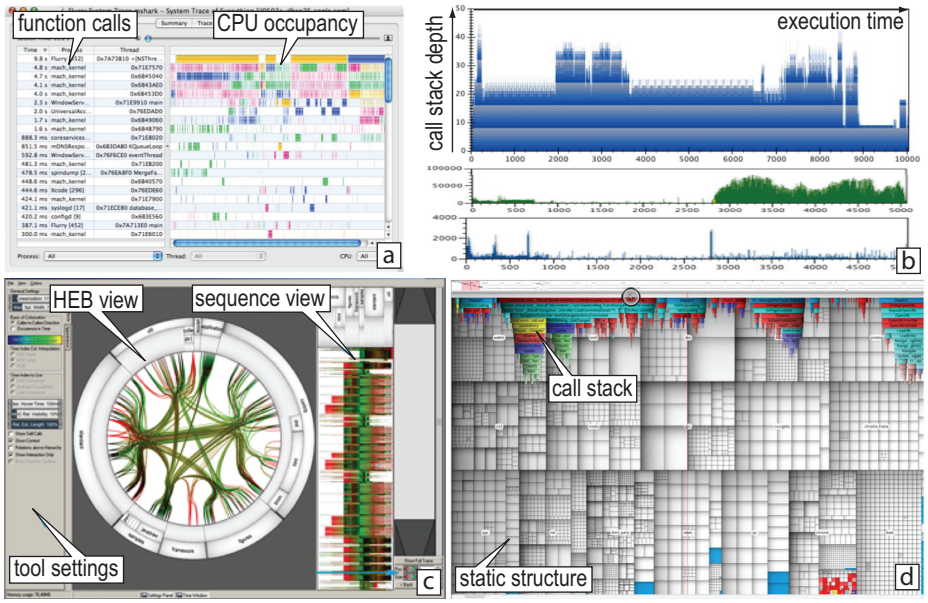

Fig. 7. Visualizations of execution traces (a-b) combined with program structure (c-d).

Program structure can be added to execution trace data. The Extravis tool [11] does this by showing execution traces with a sequence view, where each call is drawn as a horizontal line (Fig. 7c, right). Line endpoints are aligned to match the layout of an icicle plot that shows the program static structure (Fig. 7c, top-right). This shows when, and how often, a given function declaration (in the static structure) was called during the execution. Separately, a HEB view shows the static structure and calls within a user-selected time range.

An alternative structure-and-behavior combination is proposed by the ViewFusion tool [45]. An icicle plot (Fig. 7d, top) shows the call stack, similar to Fig. 7b. This plot is overlaid atop of a treemap showing the static system structure, and can be interactively panned and zoomed to select interesting execution time ranges. Function calls (from the call stack) are correlated with function declarations (from the treemap) using interaction and color mapping. Just as in

Extravis, interaction and multiple views help cope with the multivariate data implied by program structure and execution information.

TraceDiff [44] extends the ViewFusion idea to compare two execution traces T_1 and T_2. (Fig. 8). The traces are shown at the top and bottom of the view, using the same icicle plot design as in Fig. 7d. HEB-like bundles are computed between function call sequences in T_1 and T_2 that match a user-supplied similarity criterion. To simplify the view, matching calls that are close to each other in both time and caller space are aggregated and rendered as thick shaded tubes, following a visual simplification technique originally proposed for HEB views [42]. Tubes are colored to encode the call similarity. Zooming the view in and out allows users to find matching call sequences at different levels of detail.

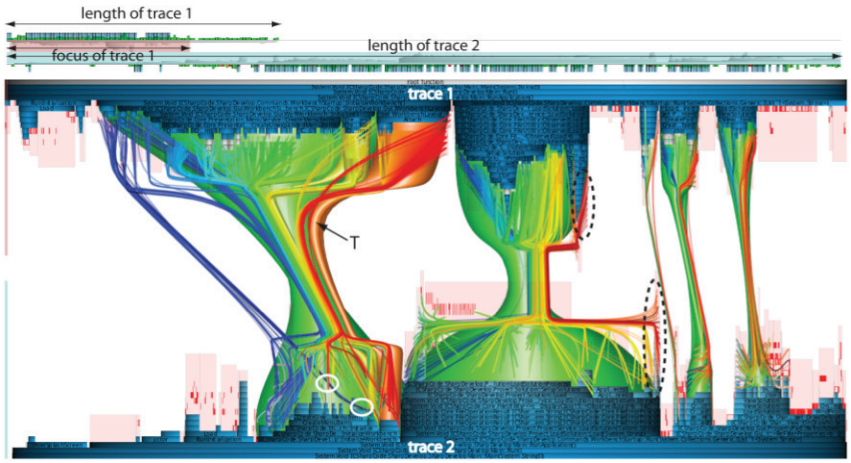

Fig. 8. Multiscale visual comparison of two execution traces.

Multiple behaviors can also be visualized against software structure. The Gammatella tool [28] collects deployment data of multiple instances of a given software system, and shows a *distribution* of this data at detailed code-line level (Fig. 9, code view), SeeSoft-like level (file view), and package level (treemap structure view). On each shown element, the computed metric distribution is visualized using a color gradient ranging from red (failed) to green (successful).

4.3 Evolution visualization

Software evolution generates time-dependent data in terms of different versions, or revisions, of a software system. These can be mined from source control management (SCM) systems, or *repositories*, such as CVS, SVN, Git, or TFS. Each revision yields a multivariate compound attributed graph that can be visualized using the techniques described in Secs. 4.1 and 4.2. However, the sheer amount of data a typical repository stores is huge: thousands of files having tens

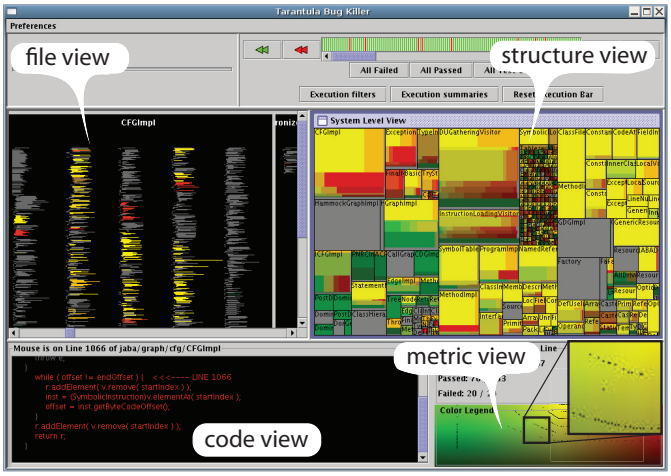

Fig. 9. Multiple deployment results *vs* system structure at different levels of detail.

of thousands of revisions spread over years. Repositories store additional data besides software structure and behavior, such as commit logs, change requests, time stamps, and the identity of developers who changed the software. This only increases the number of attributes available per data item. One approach is to reduce the amount of information by using rule or pattern mining techniques first. It turns out that the number of mined rules or patterns is still very large and that standard visualization techniques can be applied to interactively explore these rules [7]. Analyzing the data without the information loss induced by rule or pattern mining asks for different, more scalable, visualization techniques.

Fig. 10a shows a first solution for evolution visualization [50], applied to a SVN repository. The x axis maps time. Each file is drawn as a horizontal line starting when the file was first committed in the repository. Lines are cut into chunks, one per interval between consecutive revisions. Chunk colors show an attribute, *e.g.* revision author, testing results, or code quality metrics. Files can be sorted along the y axis to support several analyses. In Fig. 10a, files are sorted by decreasing activity (revision density per unit time). This allows finding the most active files (placed at the top), and correlating these with other attributes, such as age, developer identity, or code metrics. In Fig. 10a, revisions are colored by developer ID. We see a large purple spot over the first evolution half for the top files, and a large green spot over the second evolution half. This shows that, halfway the project, the main development switched between two different persons ('purple' and 'green' developers).

Figure 10b shows a detailed view from a TFS repository. Only C# source code files were selected for analysis. The top widgets show the distribution of two code metrics (complexity and size) color-coded from blue to red. These views allow easily spotting the dominant values of these metrics across a desired time or file range, which helps assessing the average code quality. The metric navigator view allows smoothly changing the color encoding used in the main file view

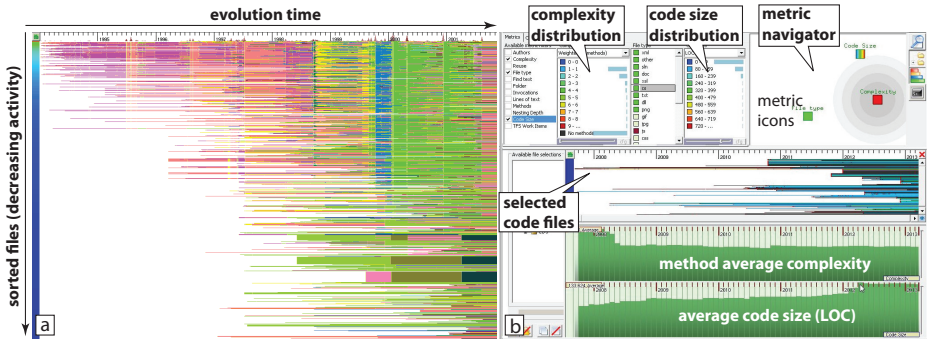

Fig. 10. Visualization of software activity and code quality trends in a repository.

between several metrics of interest, by dragging the red 'observer' icon between the respective metric icons, following the preset controller technique [49]. Below the file view, two graphs show the evolution in time of the selected metrics. We see, for example, that the average code size (number of lines of code per file) slightly increases, but the method average complexity first sharply decreases, then stays constant. The sharp complexity decrease is a good indicator of the presence of a refactoring event. The stability of the code quality metrics is, in turn, a good indicator that the software is well maintained.

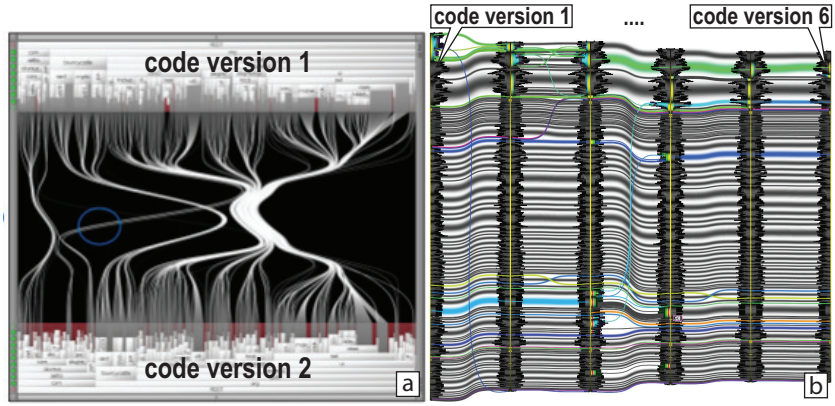

Fig. 11. Structural visualization of source code evolution.

The aggregated views in Fig. 10 scale well to show the evolution of industry-size software at *coarse* file or folder levels. However, they cannot show *relations*. Figure 11 shows two techniques that address this challenge. In image (a), the hierarchical structures of two different versions of a software system are shown using the icicle plot technique explained earlier for trace comparison (Fig. 8). In contrast to traces, where x position encodes call time, the hierarchies are

now permuted to place similar subtrees close to each other along the x axis [18]. Next, these subtrees are visually connected by HEB bundles. Asymmetries in the bundle structure indicate differences between the two hierarchies. The hierarchy sorting removes unnecessary bundle twists and thus increases readability. This idea is further extended by the CodeFlows tool (Fig. 11b). Each vertical icicle plot shows the syntactic structure of a version of a code file with the file start at top and the file end at the bottom. Similar code elements in consecutive versions, found using a syntax-aware clone detector, are connected by shaded tubes, akin to the ones used for trace comparison (Fig. 8). The 'flows' along the tubes indicate code refactoring events – parallel tubes show stable code, diverging ones show code insertions or deletions, and crossings show code permutations. Tube colors indicate attributes of interest, such as code element types (function, class, statement, symbol) for changed code, while gray indicates unchanged code.

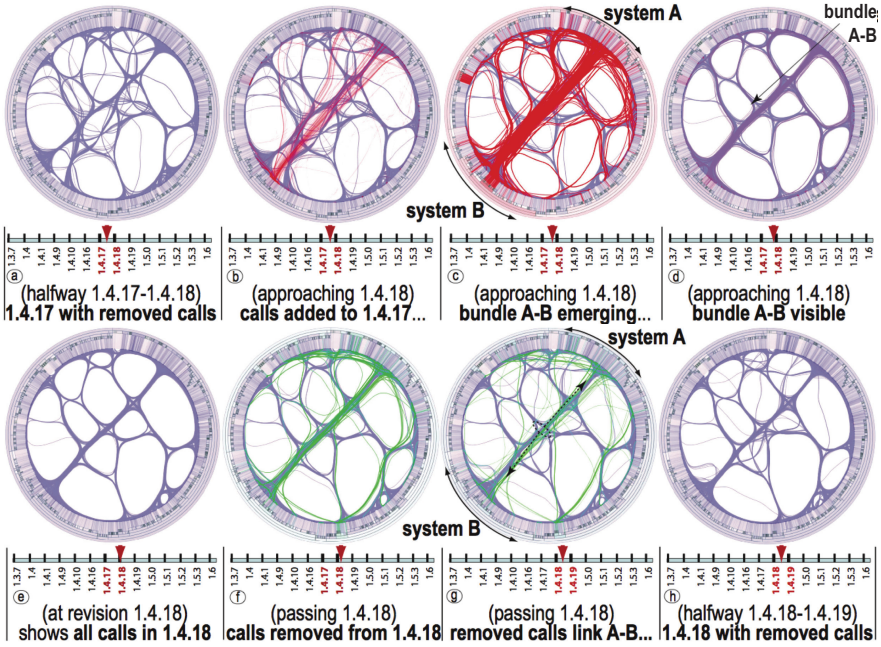

Fig. 12. Visualization of call graph change across multiple software versions.

Although effective to show structure change, the above techniques cannot show *association* changes. This is next achieved by extending edge-bundling to cope with dynamic graphs (Fig. 12). The input consists of n compound graphs mined using static analysis from n revisions in a software repository. Here, associations are function calls. For each revision, a HEB layout is built using the associations in that revision and, as structure, the union of all entities from the n revisions. This guarantees that the radial icicle plot stays fixed for all views.

Next, a continuous animation is created by smoothly interpolating corresponding edges in consecutive revisions. In parallel, appearing edges are interpolated towards their bundle, and faded in using blending, while disappearing edges are interpolated away (unbundled) and faded out. Color coding reinforces this effect: stable edges are blue; appearing edges are red, and disappearing ones are green. The images in Fig. 12 show eight frames from this animation between two consecutive revisions. We first see a red bundle appearing between components A and B. During the last four frames, a green bundle connecting A and B fades out. This indicates an important refactoring event between the considered revisions, when many calls connecting A and B were changed.

Visualizing changing associations in compound graphs in a single, static, image without animation is proposed by the TimeRadarTrees tool [6] (see Fig. 13). The top-left image shows three frames $G_1..G_3$, or snapshots, from a time-dependent compound directed graph. Hierarchy edges are orange, and associations are black. The top-right image shows the proposed visual encoding: Each node $A..E$ is represented as both a small thumbnail icon, and the correspondingly aligned sector in the large central disk. Disks are sliced in concentric rings, each one encoding a snapshot. Ring sectors in thumbnails encode the presence of an *outgoing* edge from the respective node 'revision' towards the node given by the sector's orientation. Similarly, ring sectors in the central disk encode *incoming* edges for all nodes, all revisions. Figure 13 (bottom) shows an application. The data is the folder-file hierarchy in the JEdit code base. Associations between two files $f_1..f_2$ indicate that f_1 was changed together (in the same revision) with f_2. Ring sectors are colored to indicate association weights, measured as number of lines co-changed between two files. Blue shows large co-changes, while gray shows no co-change. The dark-blue 'wedges' visible in the lower-right part of both the central disk and thumbnails for files $TODO.txt$ and $CHANGE.txt$ indicate that these two files co-changed over nearly the entire evolution period.

In contrast to the animation shown in Fig. 12, TimeRadarTrees unfolds time across the space (radial) dimension. As such, observing detailed evolution events is arguably easier, as all data is captured in one image. On the other hand, this method is less scalable in terms of number of nodes and relations.

5 Challenges and Future Directions

Summarizing our overview on multivariate graphs in software visualization, the following observations can be made.

Importance: Software structure, behavior, and evolution maps naturally to multivariate compound attributed time-dependent graphs. Such graphs have *many* attributes per node and edge, and attributes can have many different types. The *type* data is crucial to program understanding – for instance, we need to know whether a node is a function, class, or folder, and whether an edge is a call, clone, or inheritance relation, to be able to address our analysis goals.

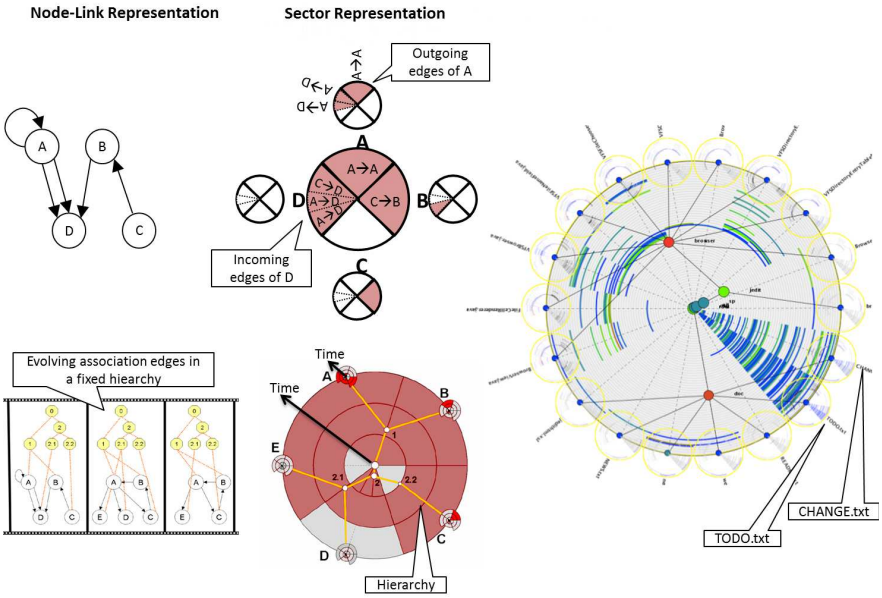

Fig. 13. Visualization of file co-change across multiple software versions.

Scalability: Being able to display *large* graphs with *many* attributes per node and/or edge is crucial to software understanding. Scalability in terms of item counts can be achieved by space-filling and dense-pixel techniques – treemaps, table lenses, timelines, SeeSoft-like views, and edge bundles. However, scalability in terms of number of attributes shown per item is quite low – most existing methods cannot show more than 2 or 3 such attributes. Linked views partially address this, but require additional user effort. Potential directions for scalability improvements are *subsampling* (drawing less items) and *dimensionality reduction* (drawing less attributes per item). However, both are challenging: For subsampling, we still do not know how to generically aggregate non-numerical attributes [11, 26]. Dimensionality reduction is a quite complex process, and can produce images which are too abstract for typical users. These issues are not unique to graphs emerging from the software engineering application domain, but important at large for any multivariate temporal graph, as discussed in more detail in Chapter **??**.

Patterns: A grand open challenge in SoftVis is how to show structural, behavioral, and evolutionary *patterns*. Patterns are essential to capture (and reason about) non-trivial events in the software, such as design decisions, execution bottlenecks, and refactoring and re-architecting. However, current visualization techniques show such patterns only *implicitly*, putting the burden of detection on the user's vision. Explicitly showing such patterns would significantly guide the

user towards a faster, and more profound, understanding of the studied software.

Standardization: Software visualization does not exist in a void. Many researchers have stressed that the current lack of toolchain integration (design tools, compilers, profilers, debuggers, SCM systems, and visualization tools) is a key adoption blocker of SoftVis tools in the IT industry [22, 32, 9, 5, 34, 33]. Tool communication via shared data formats [43] is helpful but not sufficient. A certain progress is visible in the last years in terms of SoftVis tools available as plug-ins to mainstream development environments such as Eclipse and Visual Studio. However, the largest majority of SoftVis tools does not follow this pattern. Separately, standardization of visual encodings used in SoftVis solutions, *e.g.* types of (2D *vs* 3D) layouts, diagrams, glyphs, and color maps, is an important, but not yet covered, requirement.

In this chapter, we have presented the role of multivariate graphs in the representation and visualization of the structure, behavior, and evolution of software systems. The presented application and tool examples show that such graphs play a key role in many program understanding scenarios. Recent research in software visualization has pushed the scalability limits in terms of number of items and attributes that can be visualized. However, in the same time, we see that more challenging analysis scenarios require even more powerful tools able to display larger and more complex software patterns.

As such, developing efficient and effective techniques and tools for visualizing large, complex, multivariate, and time-dependent graphs extracted from software systems remains one of the key open challenges to software visualization. As the size and importance of the software industry grows, the creation of such tools and techniques becomes ever more necessary. In the same time, the development of such solutions for software visualization has a great potential to benefit other information visualization domains where such graphs also become pervasive.

References

1. J. Abello and F. van Ham. Matrix zoom: A visual interface to semi-external graphs. In *Proc. InfoVis*, pages 183–190. IEEE, 2004.
2. R. Baecker. Sorting out sorting, 1981. 30 minute color film (developed with assistance of Dave Sherman, distributed by Morgan Kaufmann, University of Toronto).
3. S. Bassil and R. Keller. Software visualization tools: Survey and analysis. In *Proc. IWPC*, page 717, 2001.
4. F. Beck and S. Diehl. Visual comparison of software architectures. In *Proc. ACM SOFTVIS*, pages 136–143, 2010.
5. A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. H. Gros, A. Camsky, S. McPeak, and D. Engler. A few billion of lones of code later: Using static analysis to find bugs in the real world. *Comm. of the ACM*, 53(2):66–75, 2010.
6. M. Burch and S. Diehl. TimeRadarTrees: Visualizing dynamic compound digraphs. *Comp. Graph. Forum*, 27(3):823–830, 2008.
7. M. Burch, S. Diehl, and P. Weissgerber. Visual data mining in software archives. In *Proc. ACM SOFTVIS*, pages 37–46, 2005.
8. H. Byelas and A. Telea. Visualization of areas of interest in software architecture diagrams. In *Proc. ACM SOFTVIS*, pages 105–114, 2006.

9. S. Charters, N. Thomas, and M. Munro. The end of the line for software visualisation? In *Proc. IEEE VISSOFT*, pages 27–35, 2003.
10. T. Corbi. Program understanding: Challenge for the 1990s. *IBM Systems Journal*, 28(2):294–306, 1999.
11. B. Cornelissen, A. Zaidman, D. Holten, L. Moonen, A. van Deursen, and J. J. van Wijk. Execution trace analysis through massive sequence and circular bundle views. *J. Sys. & Software*, 81(12):2252–2268, 2008.
12. S. Diehl. *Software Visualization: Visualizing the Structure, Behaviour, and Evolution of Software*. Springer, Berlin, 2010.
13. S. G. Eick, J. L. Steffen, and E. E. Sumner. Seesoft—a tool for visualizing line oriented software statistics. *IEEE TSE*, 18(11):957–968, 1992.
14. R. Ferenc, A. Beszédes, M. Tarkiainen, and T. Gyimóthy. Columbus reverse engineering tool and schema for C++. In *Proc. ICSM*, page 172181, 2002.
15. E. R. Gansner and S. North. An open graph visualization system and its applications to software engineering. *Software - Practice & Experience*, 30:1203–1233, 2000.
16. H. H. Goldstine and J. von Neumann. Planning and coding of problems for an electronic computing instrument, 1947. Part II, volume I of a report prepared for the U.S. Army Ord. Dept., reprinted in [41].
17. D. Holten. Hierarchical edge bundles: Visualization of adjacency relations in hierarchical data. *IEEE TVCG*, 12(5):741–748, 2006.
18. D. Holten and J. J. van Wijk. Visual comparison of hierarchically organized data. *Comp. Graph. Forum*, 27(3):759–766, 2008.
19. H. Hoogendorp, O. Ersoy, D. Reniers, and A. Telea. Extraction and visualization of call dependencies for large C/C++ code bases: A comparative study. In *Proc. ACM VISSOFT*, pages 137–145, 2009.
20. InfoEdge. Global software industry forecast, 2013. http://www.infoedge.com.
21. J. A. Jones, M. J. Harrold, and J. Stasko. Visualization of test information to assist fault localization. In *Proc. ICSE*, pages 467–477, 2002.
22. R. Koschke. Software visualization in software maintenance, reverse engineering, and re-engineering: a research survey. *J. Soft. Maint. and Evol.*, 15(2):87–109, 2003.
23. 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 4–12, 2005.
24. J. Maletic, M. Collard, and A. Marcus. Source code files as structured documents. In *Proc. IWPC*, pages 87–91, 2002.
25. T. Mens and S. Demeyer. *Software Evolution*. Springer, 2008.
26. S. Moreta and A. Telea. Multiscale visualization of dynamic software logs. In *Proc. Eurovis*, pages 11–18, 2007.
27. O. Nierstrasz, S. Ducasse, and T. Gîrba. The story of Moose: an agile reengineering environment. In *Proc. ACM ESEC/FSE*, pages 1–10, 2005.
28. A. Orso, J. Jones, and M. J. Harrold. Visualization of program-execution data for deployed software. In *Proc. ACM SOFTVIS*, pages 67–75, 2003.
29. W. D. Pauw, E. Jensen, N. Mitchell, G. Sevitsky, J. Vlissides, and J. Yang. Visualizing the execution of Java programs. In *Proc. Inl. Sem. Revised Lectures on Software Visualization*, pages 151–162. Springer LNCS, 2001.
30. D. Quinlan. ROSE: Compiler support for object-oriented frameworks. In *Proc. CPC*, pages 81–90, 2000. see also http://www.rosecompiler.org.

31. R. Rao and S. K. Card. The table lens: Merging graphical and symbolic representations in an interactive focus+context visualization for tabular information. In *Proc. ACM Conference on Human Factors in Computing Systems (CHI)*, pages 318–322, New York, 1994. ACM Press.
32. S. P. Reiss. The paradox of software visualizaton. In *Proc. IEEE VISSOFT*, pages 59–63, 2005.
33. D. Reniers, L. Voinea, O. Ersoy, and A. Telea. The Solid* toolset for software visual analytics of program structure and metrics comprehension: From research prototype to product. *Science of Computer Programming*, 79(1):224–240, 2014.
34. T. Schafer and M. Menzini. Towards more flexibility in software visualization tools. In *Proc. VISSOFT*, pages 20–26, 2005.
35. A. E. Scott. Automatic preparation of flow chart listings. *Journal of the ACM*, 5(1):57–66, 1958.
36. B. Shneiderman. The eyes have it: A task by data type taxonomy for information visualizations. In *Proc. IEEE Symposium on Visual Languages*, pages 336–343, 1996.
37. T. A. Standish. An essay on software reuse. *IEEE TSE*, 10(5):494–497, 1984.
38. J. Stasko, M. Brown, and B. Price. *Software Visualization*. MIT Press, 1997.
39. C. Stolte, D. Tang, J. Gerth, M. Rosenblum, and P. Hanrahan. Rivet: a flexible environment for computer systems visualization. *ACM TOG*, 34(1):68–73, 2000.
40. K. Sugiyama and K. Misue. Visualization of structural information: Automatic drawing of compound digraphs. *IEEE Transactions on Systems, Man and Cybernetics*, 21(4):876–892, 1991.
41. A. H. Taub. *John von Neumann: Collected Works*. Pergamon Press, 1965.
42. A. Telea and O. Ersoy. Image-based edge bundles: Simplified visualization of large graphs. *Computer Graphics Forum*, 29(3):543–551, 2010.
43. S. Tichelaar, S. Ducasse, and S. Demeyer. FAMIX and XMI. In *Proc. WCRE*, pages 296–300, 2000.
44. J. Trümper, J. Döllner, and A. Telea. Multiscale visual comparison of execution traces. In *Proc. ICPC*, pages 262–270, 2013.
45. J. Trümper, A. Telea, and J. Döllner. ViewFusion: correlating structure and activity views for execution traces. In *Proc. TPCG*, pages 45–52. Eurographics, 2012.
46. USA Today. US healthcare spending, 2009. www.usatoday.com/news/health.
47. F. vam Ham. Using multilevel call matrices in large software projects. In *Proc. InfoVis*, pages 227–232, 2003.
48. J. J. van Wijk and H. van de Wetering. Cushion treemaps: Visualization of hierarchical information. In *Proc. IEEE InfoVis*, pages 73–78, Los Alamitos, CA, 1999. IEEE Press.
49. J. J. van Wijk and C. W. A. M. van Overveld. Preset based interaction with high dimensional parameter spaces. In F. Post, G. Nielsen, and G. Bonneau, editors, *Data visualization - State of the art*, pages 391–406. Kluwer, 2003.
50. L. Voinea and A. Telea. Visual querying and analysis of large software repositories. *Empirical Software Engineering*, 14(3):316–340, 2009.
51. R. Wettel and M. Lanza. Visualizing software systems as cities. In *Proc. IEEE VISSOFT*, pages 92–99, 2007.