Visual Comparison of Software Architectures

Fabian Beck Stephan Diehl
Computer Science
University of Trier, Germany
{beckf,diehl}@uni-trier.de

ABSTRACT

Reverse engineering methods produce different descriptions of software architectures. In this work we analyze and define the task of exploring and comparing these descriptions. We present a novel visualization technique to compare architectures based on the decomposition of the software system and on the dependencies among the code entities. A case study related to software clustering shows how we can apply this technique in practice.

Categories and Subject Descriptors

H.5.m [**Information Systems**]: Information Interfaces and Presentation

General Terms

Design

Keywords

software architecture, hierarchy comparison, graph visualization

Note. This paper makes heavy use of colors. Please read a colored version of this paper to better understand the ideas presented.

1. INTRODUCTION

Understanding the architecture of a software system is important for maintaining and evolving the system. The architecture is often described in manually created documents and diagrams. But there is no guarantee that these files match the architecture that is actually implemented. The only reliable data source of this factual architecture is the source code itself. But it contains the architecture just implicitly—in form of the code structure and its dependencies.

There are different methods to extract the implicit architecture from the code. We can just take the directory or package structure of a project. We might ask an expert to manually decompose the system [5]. Or we apply a software clustering algorithm [21] to generate a hierarchical structure of the source code. These methods provide different decompositions of the system as a partial description of its architecture.

Dependencies between code entities reflect another important part of the architecture. For instance, inheritance represents a dependency between two classes, while method calls form dependencies between methods. But there could also be hidden dependencies: Two code entities might be related by their common evolution—they might have changed together frequently [34]. Documenting these dependencies on a high level of abstraction also makes implicitly contained architecture information explicit.

There are many possible descriptions of an architecture in form of different software decompositions and dependency types—there does not exist *the* architecture description of a system. It is important to compare these different descriptions because this

- could approach the factual architecture of the system,
- might hint at the design and development process of the system, and
- may help to create more reliable architecture descriptions.

For instance, comparing the initial architecture of a system to the architecture at a later point of development could reveal architectural drifts. Checking the implemented architecture against the documented one may identify architecture violations. Or contrasting the architectures automatically extracted by different algorithm might enable us to combine the advantages of the algorithms.

There are many tools that visualize software architectures [10]. But these tools only show one description of an architecture. In this work, we present a novel software architecture visualization that focuses on comparing different architecture descriptions. The visualization helps to understand the similarities and differences of these descriptions.

At first, we need to identify concrete tasks that enable such a comparison of software architectures (Section 2). We introduce our visualization technique to explore and compare architectures based on software decompositions and code dependencies (Section 3). In a case study we apply this visualization technique to analyze package structures and clustering results (Section 4). Finally, we compare our approach to related techniques (Section 5) and draw some conclusions (Section 6).

2. COMPARING ARCHITECTURES

In a previous study [4] we compared the capabilities of different data sources to recover the architectures of software systems. In particular, we used different dependency types and applied a clustering algorithm that produced a hierarchical decomposition of the system. To assess the quality of the automatically generated decompositions, we had to compare them to a reference decomposition. There exist different metrics that implement such a comparison of software decompositions [21]—we used *MoJoFM* [32], a metric that counts the minimal number of move and join operations necessary to transform one decomposition into the other. This metric-based approach solved the problem of assessing the quality of the decompositions but left some questions unanswered:

- What are the matching and non-matching parts of the decompositions?
- Why does the clustering algorithm produce a particular result?
- How can we explain the different results when applying the algorithm to different software projects?

Starting from these questions, we felt that there is a need for understanding the differences of software decompositions and code dependencies. The experience gained from our study now helps us to identify important tasks that enable such a comparison of software architectures. Later, we will return and apply our visualization technique to the data set of this previous study (Section 4).

2.1 Decompositions & Dependencies

Our work is based on extracting the architecture of the software system from source code. The extracted architecture consists of two parts: the decomposition of the system and the dependencies among the parts of the system.

We call the elementary code units of the system *code entities*. Depending on the particular application, these entities could be methods, classes, packages, or components. Let V be the set of all code entities. The *dependency structure* of the system is a directed graph on the set of code entities $G = (V, E_G)$ where the set of edges $E_G \subset V \times V$ represents the dependencies between the entities V. A *software decomposition* divides these entities into groups or clusters of entities, which are usually hierarchically organized (e.g., like in a package structure or as a result of a hierarchical clustering algorithm). Thus, a *software decomposition* is a hierarchy (i.e., a tree) $H = (\hat{V}, E_H)$ where $\hat{V} = V \cup C$ consists of all code entities V and all clusters C, and the tree edges $E_H \subset \hat{V} \times \hat{V}$ express the containment relation such that V contains all leaf nodes and C all intermediate nodes of the hierarchy H. In terms of graph theory, such a combination of a graph and a hierarchy is called a *compound graph*.

Since our approach aims at the comparison of these data structures, we want to contrast at least two such compound graphs on code entities.

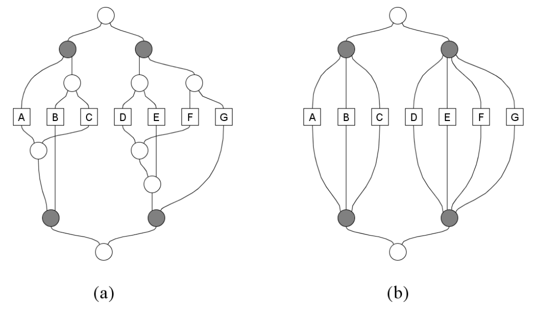

Figure 1: Two different decompositions on the same set of code entities: (a) totally expanded decompositions; (b) gray clusters collapsed.

2.2 Tasks

Before actually designing a tool that enables a user to compare software architectures based on software decompositions and code dependencies, we first need to analyze the comparison process in greater detail. To this end, we will identify key tasks. Since the user should be able to solve these tasks, they form the requirements for a comparison tool.

When we only look at the dependency structure of a system, some interesting questions already arise. For instance, dependency information might only sparsely cover the entities, or there might be clusters of entities, outliers, or hubs. Such characteristics emerge, in particular, when comparing different dependency types.

Task 1. *Analyze and compare different types of code dependencies.*

Software decompositions are supposed to follow the concept of high cohesion and low coupling [29]: The code entities of a cluster should be linked by many dependencies (high cohesion) whereas there should only be few dependencies that cross cluster borders (low coupling). Thus, a decomposition of the software might be closely related to the dependency structure. Connecting both, we might be able to answer questions like why entities belong to the same cluster, or how strong is the cohesion of a cluster and the coupling between clusters. Such information might explain the relation between a dependency type and the decomposition. If the decomposition is created automatically, it could show how the dependencies influence the clustering results.

Task 2. *Relate a software decomposition to the dependency structure.*

Finally, we look at the different software decompositions. Two decompositions are similar if the clusters of one decomposition fairly match the clusters of the other decomposition. But finding out which clusters actually match each other and which clusters do not have any match in the other decomposition is quite more interesting than an overall similarity value.

Another aspect is the level of detail of a decomposition. The hierarchical structure of the decompositions allows considering clusters on different levels. The two decompositions presented in Figure 1 (a)—one above the entities, one below—look significantly different at first glance. But when

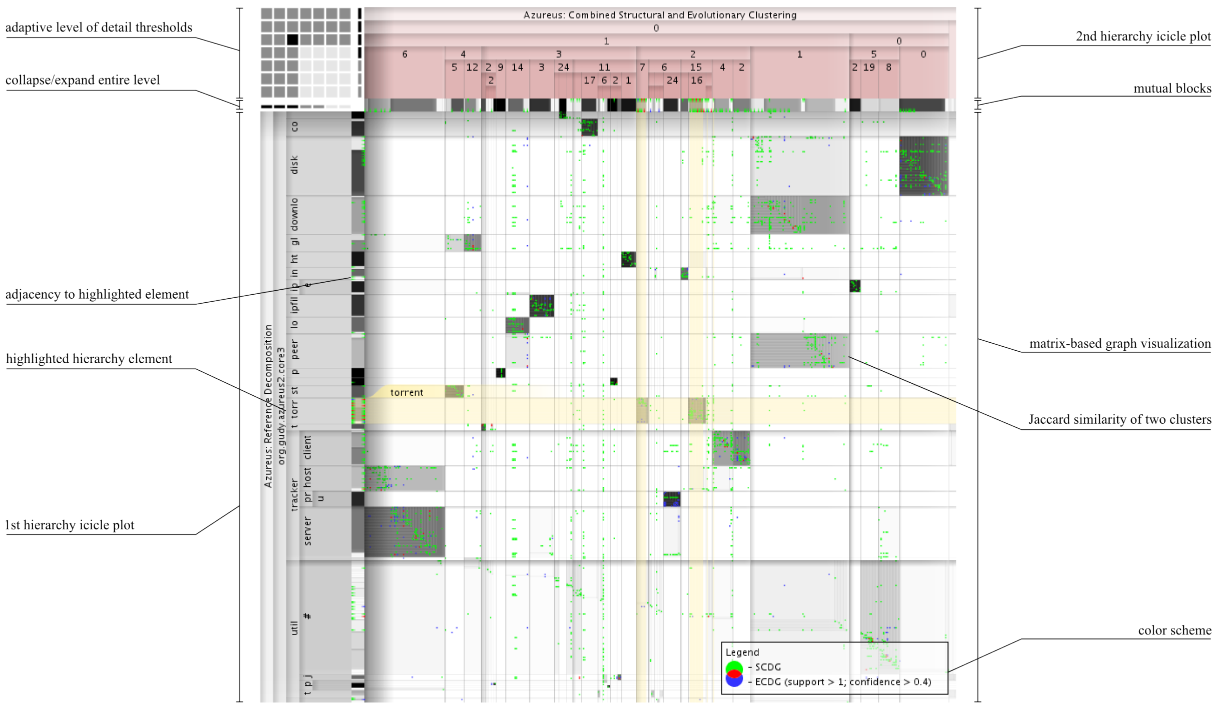
adaptive level of detail thresholds
collapse/expand entire level
adjacency to highlighted element
highlighted hierarchy element
1st hierarchy icicle plot
Azureus: Combined Structural and Evolutionary Clustering
Azureus: Reference Decomposition
org.gudy.azureus2.core3
torrent
2nd hierarchy icicle plot
mutual blocks
matrix-based graph visualization
Jaccard similarity of two clusters
color scheme
Legend
- SCDG
- ECDG (support > 1; confidence > 0.4)

Figure 2: Example of the novel matrix-based visualization technique to compare different dependency graphs and software decompositions.

collapsing particular clusters of the decompositions as depicted in Figure 1 (b), the partition of code entities is identical in both decompositions. Finding such matching levels of detail might, however, be difficult for larger decompositions. A tool that supports the user in this process would be necessary.

With respect to comparing software decompositions, we require such a tool to show similarities and differences of decompositions and to support to find matching levels of detail.

TASK 3. *Compare different software decompositions on a matching level of detail.*

Since these tasks focus on an explorative, qualitative—not quantitative—analysis, we believe a visualization is most suitable. A metric-based approach or a textual representations would not provide sufficient overview and flexibility.

3. VISUALIZATION TECHNIQUE

A visual technique that supports the user to solve the three tasks is required

- to concurrently display dependency graphs and software decompositions,
- to reveal similarities and differences in graphs and decompositions, and
- to support to find matching levels of detail in different software decompositions.

To simplify these requirement somewhat, we decided to only allow two dependency graphs and two software decomposition at maximum. Nonetheless, multiple comparisons could be realized by several pairwise comparisons.

Figure 2 provides a preview on how our novel visualization looks. It shows a representation of the Azureus[1] system, a BitTorrent client. The following sections will introduce the visualization step by step. We start with discussing the representation of the dependency graphs and the software decompositions. Further sections explain how the visualization helps to choose an appropriate level of detail in the software decompositions and how sorting increases the readability.

3.1 Dependency Representation

Our visualization technique is based on an adjacency matrix representation of graphs: It represents code entities as rows and columns of a matrix, and it depicts dependencies as cells of the matrix. A colored box at the intersection of row A and column B thus encodes a dependency from code entity A to code entity B. Hence, all code entities are represented twice, once as a row and once as a column. In a usual adjacency matrix, rows and columns are ordered equally so that self-dependencies form the diagonal of the matrix. However, our visualization deviates from this paradigm; we explain later on why.

We preferred a matrix representation over a node-link approach—diagrams where nodes represent code entities and visual links between these nodes represent dependencies—for several reasons:

Scalability Node-link diagrams suffer from occlusion problems when it comes to visualizing larger and denser graphs. Elaborate layout algorithms may ease the problem, but cannot eliminate it. In contrast, no visual elements overlap in matrix visualizations by definition. Ghoniem et al. [11] provide empirical evidence

[1]now called Vuze; `http://www.vuze.com`

for the superiority of matrix representations of larger graphs in many applications.

Edges Since we want to analyze differences in dependency graphs, we are interested in the existence of particular edges (i.e., dependencies). In contrast, tracking longer paths—an obvious shortcoming of matrix-based graph visualizations—is less important for our application. A matrix visualization focuses on edges; it explicitly shows existing and non-existing edges.

Clusters Depending on a good layout, both node-link and matrix diagrams are able to reveal clusters. But as Henry et al. [14] point out, for dense clusters, matrix representations still provide detailed information while node-link representations produce clutter.

Figure 2 shows that the matrix is the central part of our visualization. In this example, we used 477 classes as code entities. Thus, each cell indicating a dependency has only a few pixels on screen, but we can still see and discern these small points. Moreover, we observe that these colored cells are not evenly distributed over the image. The visual clusters formed by these cells hint at clusters in the dependency structure.

Our matrix-based approach is able to visualize two graphs on the same set of code entities in the same diagram. The dependencies just need to be drawn in different colors—one color for each graph and a third color to represent duplicate dependencies (i.e., dependencies that occur in both graphs). The legend depicts this color scheme. Concurrently visualizing more than two graphs with this approach is possible, but would probably confuse the user by ambiguous colors. A comparison of n graphs would need $2^n - 1$ different colors plus a background color.

3.2 Decomposition Representation

We consider software decompositions as hierarchies. A visual representation of a hierarchy can be easily attached to the sides of the matrix. We use a layered icicle plot [16] to depict a hierarchy. Such an icicle plot lays out the nodes similar to a usual tree diagram, but depicts each node as a box that fills the available space around the node. It is more space-efficient and easier to label than an equivalent node-link hierarchy diagram.

3.2.1 Hierarchies

The visualization in Figure 2 displays a software decomposition in form of the package structure on the left hand side of the diagram. Soft shadows separate the clusters, not only in the hierarchy but also continuously in the matrix. If enough screen space is available, labels identify the clusters.

Since the rows and columns of the matrix can be sorted independently, we are able to add a second software decomposition on top of the diagram. The example in Figure 2 depicts a decomposition automatically generated by a clustering algorithm.

The hierarchical structure of each decomposition implies some constraints on the order of the code entities: Only sibling entities or clusters are allowed to be switched without destroying the representation of the decomposition. Hence, code entities have to be sorted differently with respect to rows than with respect to columns.

3.2.2 Similarity Metric

The task of comparing the two decompositions consists of finding similarities and differences in the cluster structure. But without assistance, this would be a time-consuming and strenuous task: Considering a particular cluster, it is hard to identify its most similar correspondent because it has to be manually compared to every cluster in the other decomposition. A metric that is able to rank the possible correspondents with respect of their similarity to a selected node might solve the problem.

A cluster consists of a set of code entities. Thus, comparing two clusters is equivalent to comparing two sets A and B. To get a similarity measure, we are interested in how many entities concurrently belong to both clusters in relation to the size of both clusters. This can be expressed as the size of the intersection of the two sets divided by the size of the union of the sets—the Jaccard coefficient:

$$\mathrm{sim}(A, B) := \frac{|A \cap B|}{|A \cup B|}.$$

We integrate the similarity information based on the Jaccard coefficients in the background of the matrix representation. The clusters form a matrix-like meta-structure where the cluster—not the code entities—represent the rows and columns. Each comparison of two clusters can be represented as a cell of this matrix. We use the background brightness of the cells to encode the Jaccard similarity value of the according cluster: Dark backgrounds visualize high similarity values. Coloring each possible pair of clusters like this would, however, lead to overlapping cells and thus ambiguous shadings. Hence, this approach is able to compare two decompositions only on one level of clusters for each decomposition. In our visualization the user is able to choose this level manually or supported by an optimization algorithm (details will follow in Section 3.3).

Figure 2 shows that the visualization enables the user to identify similar clusters at a glance: We can immediately detect the most similar cluster combinations with the help of this background structure. Moreover, non-matched clusters result in rows or columns consisting only of a set of light-gray boxes without any darker one. Table 1 lists some examples of such matched and non-matched clusters.

Table 1: Examples of matched and non-matched clusters in Figure 2.

Package Decomposition	Clustered Decomposition
disk	0.0.0
tracker.pr.u	0.1.2.6.24
ipfil	0.1.3.3
util.#	–
–	0.1.1

3.3 Level of Detail

Nevertheless, when comparing two decompositions, it is necessary to chose an appropriate level of detail. Especially clustered decompositions tend to be deep and fine-grained hierarchies while, for instance, package structures are normally flat and more coarse-grained.The user should be able to collapse clusters to find an appropriate level. A level is appropriate if the following conditions are true:

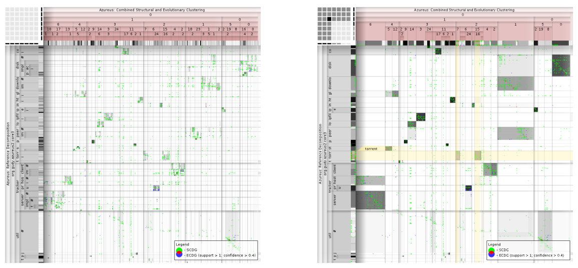

Figure 3: The example from Figure 2 without and with an appropriately chosen level of detail.

Conformance The decompositions match as far as possible with respect to a measure of similarity.

Significance The structure of both decompositions is preserved (i.e., not too many clusters should be collapsed).

It is always possible to reach maximum conformance by totally collapsing both decompositions. But this obviously violates the condition of significance. Hence, these two conditions usually must be traded off against each other.

Figure 3 gives an example of how important the level of detail is. While in the default visualization on the left hand side the background patterns are much too fine-grained to easily find differences and similarities in both decompositions, the right hand side image is much more readable because it has an appropriately chosen level of detail.

3.4 Interaction

The visualization allows the user to collapse or expand clusters by clicking on their visual representations. Furthermore, slim markers on the side of the hierarchy enable the user to collapse or expand whole hierarchy levels (Figure 2, top left). These markers also indicate which levels are totally collapsed (light gray), partially collapsed (gray), or fully expanded (dark gray). A collapsed cluster still claims as much space as an expanded cluster. The main difference is that collapsing a cluster moves the cluster comparison to a higher level: Larger gray scale background boxes now display the cluster similarity metric values on this higher level.

The gray scale matrix in the background is the most important criterion to assess the similarity of the two decompositions at a particular level. Roughly speaking, few black boxes and many white boxes indicate a high conformance while many low-contrast gray boxes indicate a low conformance. The interactive expand and collapse mechanism allows the user to explore different levels, but especially for larger data sets, a good automatically proposed level of detail would be of great help.

3.5 Optimization Criterion

To implement an automatic algorithm, we had to find a formal optimization criterion that assesses the quality of a particular level of detail state. Such a state consists of the partially collapsed decompositions. Each collapsed decomposition implies a partition of the code entities like presented in the example of Figure 1. Hence, an optimization criterion is a real-valued objective function defined on two partitions.

We propose an objective function that counts the number of matching clusters of the two partitions P_1 and P_2. The degree of similarity could be again computed by the Jaccard similarity coefficient. Adding these similarity coefficients for all possible cluster combinations, we come up with the following objective function.

$$f(P_1, P_2) = \sum_{A \in P_1} \sum_{B \in P_2} \omega_{A,B} * \mathrm{sim}(A, B)$$

To consider the different sizes of the cluster, we added a weighting coefficient ω. For two clusters A and B, the coefficient just sums up the number of elements of both clusters: $\omega_{A,B} := |A| + |B|$. It is independent of the similarity of the clusters and gives larger cluster combinations a higher weight.

3.5.1 Significance Level Thresholds

This objective function, however, only evaluates the level of detail with respect to conformance and does not consider significance. But it is difficult to balance significance and conformance in a single objective function. An appropriate balance might also depend on the concrete application the user has in mind.

To allow high conformance on different levels of significance, we introduce a significance level threshold for each of the two decompositions. This threshold prevents collapsing clusters beyond this level while optimizing the conformance.

The grid pattern in the upper left corner of the visualization displays the two significance level thresholds. The user is able to set both levels with a single click. For instance, in Figure 2 the user has clicked on the cell at the intersection of the third column and the third row, indicated by a black box. This means that both decompositions have to stay expanded up to the third level while optimizing the conformance of the decompositions.

3.5.2 Optimization Algorithm

The two decompositions, the objective function, and the two significance level thresholds form a constrained maximization problem. As an optimization strategy for this problem, an exhaustive search, however, is not applicable for nontrivial data sets. The number of possible partitions induced by a single hierarchy might already grow exponentially with the number of leaf nodes n: In the worst case—a binary hierarchy—at least the $\frac{n}{2}$ intermediate nodes of the lowest level can be independently collapsed and expanded. This leads to at least $2^{\frac{n}{2}}$ different partitions.

Instead, we use a hill climbing algorithm to find a local maximum of the optimization problem. As an initialization, the algorithm expands the two decompositions to the minimal level, which is defined by the two significance level thresholds. Then it tries to maximize the objective function as follows (expand operations that improve the objective function persist while all other expand operations are directly undone):

1. Expand each collapsed node of the first hierarchy one by one.

2. Repeat step (a) for all collapsed nodes of the second hierarchy.

3. If nothing has improved in step 1 and 2, try to expand two nodes concurrently, one in the first and one in the second hierarchy (systematically over all possible combinations).

These three steps are repeated until they cannot provide any further improvement. The third step turned out to be helpful to skip local maxima.

Thus, our optimization strategy provides an interactively selectable level of detail with an adaptive refinement to underline matching parts of the two decompositions. Clicking on a cell of the threshold visualization, the algorithm automatically produces a layout of few black boxes surrounded by many white ones. Figure 3 illustrates this process: While the image on the left hand side shows two totally collapsed hierarchies, the image on the right hand side is actually created applying the optimization algorithm (this image is also depicted in Figure 2 in larger size). In our experiences using the visualization, the level of detail optimization turned out to be very valuable because it drastically reduces the time to find an appropriate level of detail.

3.6 Sorting

The linear ordering of the rows and columns is elementary for the readability of a matrix graph visualization [22]. With a random ordering no structure would be visible, whereas a good ordering would reveal important graph structures like clusters, hub vertices, or outliers. Different approaches and algorithms exist to create a reasonable layout—Mueller et al. [22] as well as Henry and Fekete [13] survey these techniques in detail.

The hierarchical representation of the two software decompositions, however, constrains this ordering in our visualization. If the decomposition follows a certain semantic, this mandatory sorting may already help to reveal the structure of the dependency graphs. Nevertheless, the ordering still leaves some degree of freedom: The positions of sibling clusters and code entities can be switched.

3.6.1 Local Diagonals

Interpreting matrix diagrams, the diagonal is an important reference line [23]: In a typical matrix representation of a graph, the cells on the diagonal represent self edges. Our visualization depicts two decompositions at the same time. Thus, in contrast to most matrix graph visualizations, it uses different vertical and horizontal entity orders. A side effect is that the former diagonal entries, which we will call *self-referencing cells* in the following, are scattered all over the diagram. To regain a local diagonal structure, we use the remaining degrees of freedom and sort sibling code entities.

Figure 4 illustrates the algorithm: In the first decomposition, every pair of code entities in the same cluster is switched if the two self-referencing cells define an ascending line (read from left to right). The same procedure is applied to the second decomposition. Finally, all lines between sibling elements are descending and form local diagonals.

3.6.2 Mutual Blocks

Besides the locally regained diagonal structure, the result of this local sorting algorithm are blocks of neighboring entities in the one decomposition that all belong to the same cluster in the other decomposition. These *mutual blocks* reveal an additional important information for comparing the two decompositions: They show how a cluster in one decomposition is spread over the other decomposition.

The mutual blocks are encoded as boxes on the leaf level of the two decompositions. They look like a bar code and form the border lines between the icicle plots and the adjacency matrix (Figure 2). Each box represents a mutual block, which relates two clusters from the two decompositions. The brightness of the box corresponds to the Jaccard similarity of the two associated clusters: A black box stands for a good match while a gray or white box represents mutual blocks that only partially cover the two clusters.

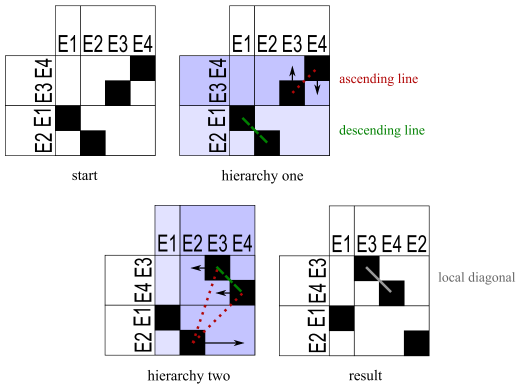

Figure 4: Sorting algorithm on leaf level. Black boxes mark self-referring cells; ascending lines are dotted; descending lines are dashed; arrows indicate the transformations.

Although the similarity of clusters is already encoded in the matrix background, these mutual blocks help to detect further interesting phenomena. On the one hand, they show more clearly whether a particular cluster is matched by a cluster of the other decomposition—for instance, we see at a glance that the *torrent* package in Figure 2 is only half matched. On the other hand, also small differences in mostly matching clusters become visible, as it is the case for the *disk* package in Figure 2. This ability is very important for analyzing evolving decomposition structures.

3.7 Details on Demand

In practical application, it is important to get further details on demand. Moving the mouse over a code entity or a cluster, the tool shows its name as a tooltip and highlights adjacent vertices (the *torrent* package is highlighted in Figure 2). When moving the mouse over a colored matrix cell representing a dependency, the labels of both related code entities appear.

4. CASE STUDY

The following visual analysis applies the introduced visualization technique to a previous study on software clustering [4]. The study incorporated the software clustering tool Bunch [19], an approach based on the principle of high cohesion and low coupling of modules, to compare different data sources for software clustering. We assessed the clustered software decompositions retrieved from six sample projects by comparing them to a reference decomposition: the actual package structure of the project. As discussed in the Section 2, this quantitative assessment left some questions unanswered.

In the following we will analyze the software decompositions again, but now in a more qualitative and explorative approach. The tasks defined in Section 2.2 provide different viewpoints on the data sources and clustering results.

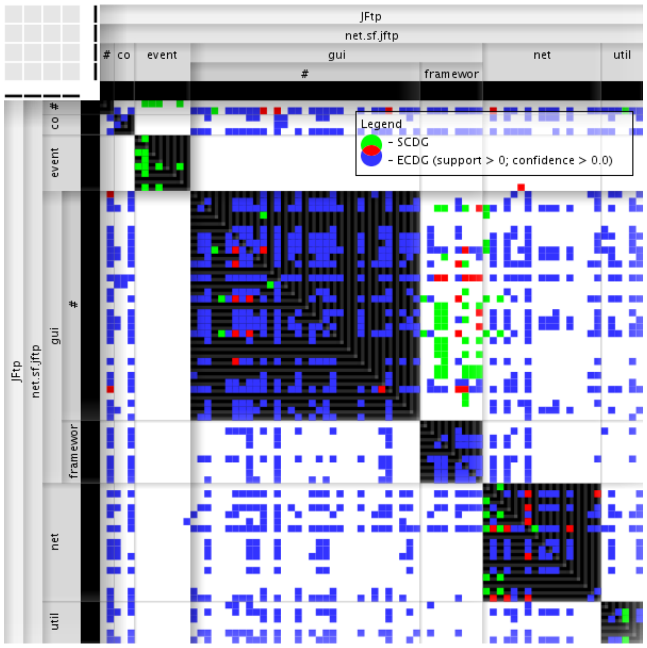

Figure 5: Graph comparison between the SCDG and the unfiltered ECDG for the JFtp project; the package structure provides a default decomposition.

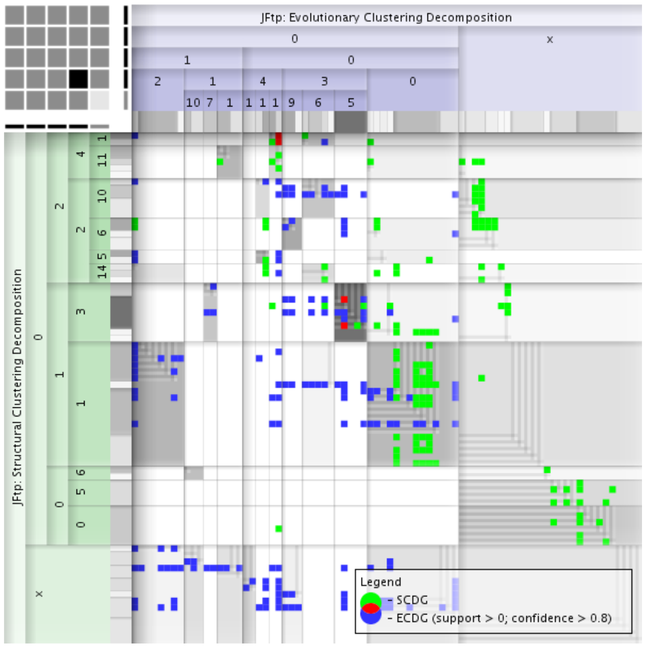

Figure 6: Two different clustered decompositions, one based on structural dependencies (vertical), the other based on evolutionary dependencies (horizontal).

In general, our analyses consider all six sample projects of the study, namely, Azureus, JEdit, JFreeChart, JFtp, JUnit, and Tomcat. For practical reasons, we only depict the resulting visualizations for JFtp, the smallest of the sample projects.

The case study will only show that the approach basically works. A task-oriented user study based on refined versions of the tasks discussed in Section 2.2 may provide more reliable results on the usability, readability, and efficiency of the proposed visualization technique. Nevertheless, it could be difficult to define an appropriate control group because there does not exist a directly competing approach.

4.1 Compare Dependencies (Task 1)

The dependency graphs are the basis for the clustering process: They are the input for the clustering algorithm. On the one hand, we used static code dependencies—like inheritance, aggregation, and usage—to represent a traditional software clustering approach. These dependencies form the Structural Class Dependency Graph (SCDG). On the other hand, we used evolutionary couplings, which form the Evolutionary Class Dependency Graph (ECDG), to represent hidden dependencies. These evolutionary couplings relate two classes if these classes have been frequently changed together in the history of the software project (co-changed). The dependency strength consists of a support value—the absolute number of co-changes—and a confidence value—a relative number of co-changes. To reduce the noise in the data set, a filter eliminates weak dependencies.

In this first phase of our case study, we use the visualization as a graph comparison tool (Task 1). Since clustered decompositions are not yet relevant, the package structure is employed as a default decomposition. The background structure thus only shows black boxes on the diagonal, but does not carry any further information. The graph visualization, however, reveals significant differences in the graph structures, as illustrated for JFtp in Figure 5.

SCDG (green & red dependencies) Sparse graphs, but with dependencies that cover most of the nodes at least once. Some outstanding nodes with many incoming or outgoing dependencies, which form a kind of hub nodes.

ECDG (blue & red dependencies) Dense graphs (without filtering) up to very sparse ones (with a strong filtering). Local concentrations of edges form dense clusters. But many nodes are not covered by any dependency.

These results show two main drawbacks of the ECDG: the local concentration of dependency information and the overall low density of the dependency graph, especially for stronger filtering setups. A solution might be a local filtering instead of the global filtering of these evolutionary dependencies.

Furthermore, the intersection of the dependencies (red dependencies) of both graphs is small and mostly relates classes of the same package. This observation explains why it is beneficial to give those dependencies more weight in the clustering process.

4.2 Decompositions & Dependencies (Task 2)

In this second stage of our analysis, we also consider software decompositions produced by the employed clustering approach. We use the vertical axis to depict the clustered decomposition based on the structural code dependencies (SCDG) and the horizontal axis for the one based on the evolutionary dependencies (ECDG). Figure 6 shows such a visualization for the JFtp project.

In the evolutionary software decomposition (Figure 6, top), cluster *x* looks interesting: It roughly covers a third of the hierarchy, but is not subdivided further. There also exists a cluster *x* in the structural software decomposition (Figure 6, left), but it is much smaller. This cluster *x* represents all elements that could not be clustered because there was not any dependency information available for them. Thus, there are no evolutionary dependencies available for about a third of the classes of the software system, and the clustering algorithm could only cluster the other two thirds of the system. This situation is even worse in the other sample projects. This sparse coverage seems to be the main problem of clustering a software system exclusively with evolutionary information (ECDG).

Admittedly, we already uncovered this fact in the previous study. But there we needed a metric to measure the coverage, in contrast to the visualization, where we are able to grasp the same fact without even intentionally looking for it.

Analyzing the relation of the hierarchies and the graphs in more detail, we observe in the visualization that deeper hierarchies come along with clearly identifiable clusters indicated by visual clusters in the dependency graphs. The clustering algorithm seems to produce flatter hierarchies when the classification of classes is more ambiguous. But we are not able to find any significant difference between structural and evolutionary decompositions with respect to this effect.

All in all, the conformance between the structural and evolutionary decompositions is low. This might indicate that both data sources actually cover different dimensions of code dependencies—a combination of both data sources promises to combine these two dimensions. Actually, this lead to slightly better clustering results as we found out in the previous study.

4.3 Compare Decompositions (Task 3)

With our visualization the user is able to detect matching clusters at first glance—perhaps the most striking feature of the technique. For instance, in Figure 7 the *event* package is almost perfectly matched by cluster *0.0.2*, as we learn from the background shading of the matrix. The precondition is an appropriate level of detail, which might be easily gained by the level of detail optimization algorithm.

We use this ability of detecting well clustered packages to identify those packages that are either fairly matched or non-matched. In most cases, nearly perfectly matched clusters posses a high structural cohesion—many references among the classes of the cluster—often supported by a good evolutionary cohesion—the classes of the cluster were frequently changed together. In contrast, matching clusters predominantly based on high evolutionary cohesion are rare. This explains why combining structural and evolutionary data improved the clustering quality in the quantitative study and using exclusively evolutionary data was only partly successful.

In contrast, utility packages—packages that provide some global functionality—could hardly be retrieved by our clustering approach. The visualization supports to identify those packages, even when they are not named *utility* or *util*, by their characteristic structure: Utility packages do not have outgoing dependencies to other non-utility packages,

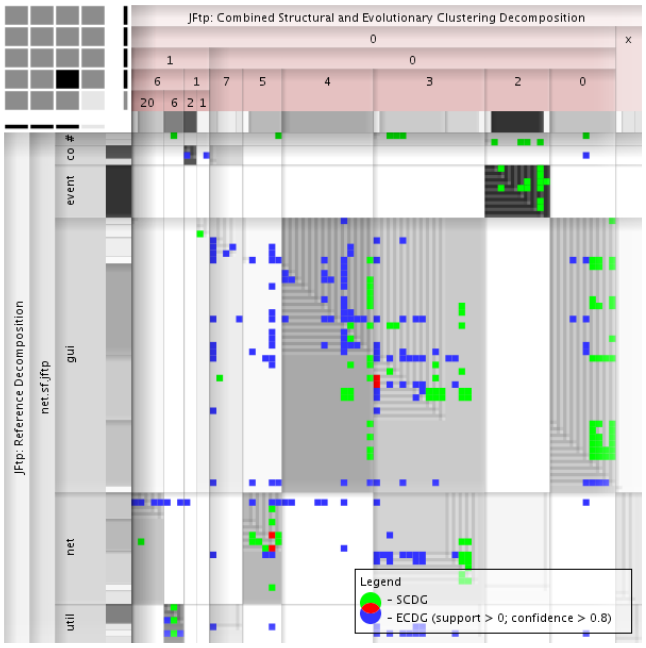

Figure 7: Clustered software decomposition based on the combined structural and evolutionary graphs compared to the reference decomposition (package structure).

but many incoming ones from diverse packages. We are able to gain this information either by looking at the adjacency matrix or by using the interactive details-on-demand that highlight all adjacent classes (including the direction of adjacency) for a package. Once we identified these utility packages, the visually encoded cluster similarity revealed no significant correspondence to the clustered packages. This problem is a known problem of dependency-based clustering approaches [2]. A preprocessing that detects such packages before the actual clustering, like proposed by Mancoridis et al. [19], might improve the clustering results.

Our visualization also showed that in some setups—in particular those involving the larger projects—the clustering algorithm was not able to create a decomposition with at least a roughly matching granularity: All possible levels were much too fine-grained in contrast to the reference decomposition. Repairing this weakness of the algorithm might also result in much better clustering results for these setups.

5. RELATED WORK

Software architecture visualization is an established discipline in software visualization research [8, 10]. Many tools from this area visualize software decompositions and code dependencies—*SHriMP* [30], *Software Landscapes* [3], or *Class Blueprints* [9], just to name a few. Most of these visualizations employ the node-link metaphor to represent a dependency graph structure. But matrix-based visualizations of graphs seem to gain importance due to their advantages when it comes to visualize larger graphs. They have already been employed to analyze dependencies of software projects, for example, method calls [31] or evolutionary couplings [7]. Originating from the analysis of manufacturing processes, so-called *Dependency Structure Matrices* are also able to visualize software architectures in a matrix structure [26]. Due to a specialized sorting, these matrices help to detect cyclic dependencies and architecture violations.

Visualization has also played a role in software clustering and has helped to present single clustering decompositions in a readable way. Hierarchical decompositions have been depicted in a form of tree diagrams [27], code dependencies have been represented as graph visualizations [20], and similarity of code entities in high-dimensional feature spaces have been visualized as similarity matrices [17]. Other clustering related research communities use similar forms of cluster visualizations.

These visualizations are able to present a single software architecture description or a single clustering result. But to the best of our knowledge, no approach, however, uses a matrix visualization to concurrently compare different graphs and hierarchies, neither in the domain of software architecture visualization nor in clustering-related visualizations.

Nevertheless, there exist specialized visualizations to compare different hierarchies (without a graph structure). A straightforward approach is to place two hierarchies face to face with each other and connect related leaves by visual links. Edge crossings reduce the readability of such an visualization. There exist some algorithms that alleviate this problem by minimizing the number crossings [6]. Holten and van Wijk [15] follow another strategy and enhance the approach by bundling links into meaningful groups. In contrast, brushing is a totally different paradigm to express similarities of hierarchy nodes. For instance, TreeJuxtaposer [24] displays similar sub-tree structures interactively by highlighting the best corresponding node (based on the Jaccard coefficient). Many other visualizations that compare hierarchical structures exist; Graham and Kennedy [12] provide a more exhaustive survey.

In the field of bioinformatics, *Cluster Heat Maps* are a popular visualization technique to analyze large clustered genome data sets [33]. These heat maps consist, first, of a color-coded matrix that usually relates genes (objects) to a set of conditions (attributes), and second, of an attached hierarchy retrieved by clustering. Not only the objects can be clustered, but also the attributes: A second hierarchy groups the attributes of the matrix. Concurrently finding an optimal clustering of objects and attributes is known as *biclustering* (e.g., [18] gives an overview). These cluster heat maps, indeed, look similar to our approach, especially with two hierarchies attached. Nonetheless, the fundamental difference is that the cluster maps do not compare two hierarchies on the same set of objects, but help to concurrently cluster two independent sets: objects and attributes.

Software clustering results are often evaluated by comparing them to a reference decomposition of approved quality. Like applied in our previous study a metric provides a similarity value. These metrics usually work on flattened decomposition. But there exist first approaches that additionally regard the hierarchical structure of the decompositions [27, 28]. These metric-based approaches may be sufficient to get a quality measure for an automatically created decomposition, but do not explain the difference.

Other tools allow the user to visually compare graph structures. For instance, Andrews et al. [1] present a node-link approach to compare business processes and surveys related node-link approaches. Beside these specialized tools, every dynamic graph visualization enables graph comparisons: The two contrasted graphs form a sequence of changing graphs. There even exist dynamic compound graph visualizations, which are able to concurrently display a changing hierarchy [25]. These visualizations, however, are more suitable for evolving graphs and hierarchies, but not to contrast two totally different data sets like those discussed in this paper.

6. CONCLUSION

In this paper we analyzed how to compare software architectures with respect to software decompositions and code dependencies. To this end, we developed a novel visualization technique based on an adjacency matrix representation of graphs. The visual analysis of the results of a previous quantitative study on software clustering shows that the visualization supports the analysis tasks introduced in Section 2.2. The main capabilities of the visualization are

- concurrently contrasting software decompositions and code dependencies,
- easily detecting matching and non-matching parts in software decompositions, and
- semi-automatically finding a matching level of detail comparing two software decompositions.

Our visualization technique is the first approach toward visually comparing software architecture descriptions. It may foster the understanding of differences in these descriptions. A major application is software clustering where software decompositions are generated automatically. These decompositions can be put into relation to certain dependency types and compared to each other. The visualization might help to improve software clustering, especially with respect to selecting an appropriate data source.

7. REFERENCES

[1] K. Andrews, M. Wohlfahrt, and G. Wurzinger. Visual graph comparison. In *IV '09: Proceedings of the 13th Conference on Information Visualisation*, pages 62–67. IEEE Computer Society, 2009.

[2] P. Andritsos and V. Tzerpos. Information-theoretic software clustering. *IEEE Transactions on Software Engineering*, 31(2):150–165, 2005.

[3] M. Balzer, A. Noack, O. Deussen, and C. Lewerentz. Software Landscapes: Visualizing the structure of large software systems. In *VisSym '04: Proceedings of the Joint Eurographics - IEEE TCVG Symposium on Visualization*, pages 261–266. Eurographics Association, 2004.

[4] F. Beck. Improving software clustering with evolutionary data. Diploma thesis, University of Trier, 2009.

[5] I. T. Bowman, R. C. Holt, and N. V. Brewster. Linux as a case study: its extracted software architecture. In *ICSE '99: Proceedings of the 21st international conference on Software engineering*, pages 555–563, New York, NY, USA, 1999. ACM.

[6] K. Buchin, M. Buchin, J. Byrka, M. Nöllenburg, Y. Okamoto, R. I. Silveira, and A. Wolff. Drawing (complete) binary tanglegrams: Hardness, approximation, fixed-parameter tractability. In *GD*

'08: 16th International Symposium on Graph Drawing, Lecture Notes in Computer Science, pages 324–335. Springer, 2008.

[7] M. Burch, S. Diehl, and P. Weißgerber. Visual data mining in software archives. In *SoftVis '05: Proceedings of the 2005 ACM Symposium on Software Visualization*, pages 37–46, New York, NY, USA, 2005. ACM Press.

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

[9] S. Ducasse and M. Lanza. The class blueprint: Visually supporting the understanding of classes. *IEEE Transactions on Software Engineering*, 31(1):75–90, 2005.

[10] Y. Ghanam and S. Carpendale. A survey paper on software architecture visualization. Technical report, 2008.

[11] M. Ghoniem, J. D. Fekete, and P. Castagliola. A comparison of the readability of graphs using node-link and matrix-based representations. In *INFOVIS '04: IEEE Symposium on Information Visualization*, pages 17–24, 2004.

[12] M. Graham and J. Kennedy. A survey of multiple tree visualisation. *Information Visualization*, 2009.

[13] N. Henry and J.-D. Fekete. Matrixexplorer: a dual-representation system to explore social networks. *IEEE Transactions on Visualization and Computer Graphics*, 12(5):677–684, September 2006.

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

[15] D. Holten and J. J. van Wijk. Visual comparison of hierarchically organized data. *Computer Graphics Forum*, 27(3):759–766, 2008.

[16] J. B. Kruskal and J. M. Landwehr. Icicle plots: Better displays for hierarchical clustering. *The American Statistician*, 37(2):162–168, 1983.

[17] A. Kuhn, S. Ducasse, and T. Gîrba. Enriching reverse engineering with semantic clustering. In *WCRE '05: Proceedings of the 12th Working Conference on Reverse Engineering*, pages 133–142, Washington, DC, USA, 2005. IEEE Computer Society.

[18] S. C. Madeira and A. L. Oliveira. Biclustering algorithms for biological data analysis: A survey. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 1(1):24–45, 2004.

[19] S. Mancoridis, B. S. Mitchell, Y. Chen, and E. R. Gansner. Bunch: A clustering tool for the recovery and maintenance of software system structures. In *ICSM '99: Proceedings of the IEEE International Conference on Software Maintenance*, pages 50–59, Washington, DC, USA, 1999. IEEE Computer Society.

[20] S. Mancoridis, B. S. Mitchell, C. Rorres, Y. Chen, and E. R. Gansner. Using automatic clustering to produce high-level system organizations of source code. In *IWPC '98: Proceedings of the 6th International Workshop on Program Comprehension*, pages 45–52, Washington, DC, USA, 1998. IEEE Computer Society.

[21] O. Maqbool and H. A. Babri. Hierarchical clustering for software architecture recovery. *IEEE Transactions on Software Engineering*, 33(11):759–780, 2007.

[22] C. Mueller, B. Martin, and A. Lumsdaine. A comparison of vertex ordering algorithms for large graph visualization. In *APVIS '07: Proceedings of the 6th International Asia-Pacific Symposium on Visualization*, pages 141–148, 2007.

[23] C. Mueller, B. Martin, and A. Lumsdaine. Interpreting large visual similarity matrices. In *APVIS '07: Proceedings of the 6th International Asia-Pacific Symposium on Visualization*, pages 149–152, 2007.

[24] T. Munzner, F. Guimbretière, S. Tasiran, L. Zhang, and Y. Zhou. Treejuxtaposer: scalable tree comparison using focus+context with guaranteed visibility. *ACM Transactions on Graphics*, 22(3):453–462, July 2003.

[25] M. Pohl and P. Birke. Interactive exploration of large dynamic networks. In *VISUAL '08: Proceedings of the 10th international conference on Visual Information Systems*, pages 56–67, Berlin, Heidelberg, 2008. Springer-Verlag.

[26] N. Sangal, E. Jordan, V. Sinha, and D. Jackson. Using dependency models to manage complex software architecture. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, volume 40, pages 167–176, New York, NY, USA, October 2005. ACM.

[27] M. Shtern and V. Tzerpos. A framework for the comparison of nested software decompositions. In *WCRE '04: Proceedings of the 11th Working Conference on Reverse Engineering*, pages 284–292, Washington, DC, USA, 2004. IEEE Computer Society.

[28] M. Shtern and V. Tzerpos. Lossless comparison of nested software decompositions. In *WCRE '07: Proceedings of the 14th Working Conference on Reverse Engineering*, pages 249–258, Washington, DC, USA, 2007. IEEE Computer Society.

[29] W. P. Stevens, G. J. Myers, and L. L. Constantine. Structured design. *IBM Systems Journal*, 13(2):115–139, 1974.

[30] M. A. Storey, C. Best, and J. Michaud. SHriMP views: An interactive environment for exploring java programs. *International Conference on Program Comprehension*, 0, 2001.

[31] F. van Ham. Using multilevel call matrices in large software projects. In *INFOVIS '03: Proceedings of the IEEE Symposium on Information Visualization*, pages 227–232, 2003.

[32] Z. Wen and V. Tzerpos. An effectiveness measure for software clustering algorithms. In *IWPC '04: Proceedings of the 12th International Workshop on Program Comprehension*, pages 194–203. IEEE Computer Society, 2004.

[33] L. Wilkinson and M. Friendly. The history of the cluster heat map. *The American Statistician*, 63(2):179–184, 2009.

[34] T. Zimmermann, S. Diehl, and A. Zeller. How history justifies system architecture (or not). In *IWPSE '03: Proceedings of the 6th International Workshop on Principles of Software Evolution*, Washington, DC, USA, 2003. IEEE Computer Society.