Visually Exploring Multi-Dimensional Code Couplings

Fabian Beck, Radoslav Petkov, Stephan Diehl
University of Trier
Germany
Email: {beckf,diehl}@uni-trier.de

***Abstract*—Dependencies and coupling relationships between code entities can be manifold. They form a graph structure with several different types of edges. Visualizing these graphs presents two challenges: the often large size of the graphs and the readable representation of the different edge types. In this paper we present a new node-link graph visualization technique addressing these challenges. Different types of edges are represented in multiple, small diagrams. These diagrams are placed side-by-side like the pictures of a comic strip. Among other things, edge bundling is applied to improve the scalability of the visualization. Elaborate interaction techniques with respect to focusing and merging data aim at providing help for comparing different edge types. A case study demonstrates how the visualization can be used to analyze a mid-size software project in practice.**

I. Introduction

The building blocks of software systems—variables, methods, classes, components, etc.—depend on each other, communicate with each other, or share certain characteristics. In particular, these code entities are directly coupled by static structures like method calls, inheritance, aggregation, but also indirectly, for instance, by shared code clones, similar semantic, shared code owners, or frequent co-changes. These coupling concepts are leveraged across various applications in software engineering, however, only usually considering one or a combination of two of these concepts. A first step towards understanding the yet mostly unknown relationships between the multi-dimensional code coupling concepts is to visually explore these data sets. Moreover, visualization can be a vehicle to exploit the coupling information in the development and maintenance process of software systems. In this paper, we present a new graph visualization technique tailored for the comparison of different coupling concepts in software projects. Figure 1 provides a first preview on this visualization showing three types of structural code couplings that connect the 78 classes and interfaces of the JFtp project.

A. Visualization Problem

The problem of visualizing multi-dimensional code couplings can be considered as a complex graph visualization problem: Code couplings describe a graph structure on the code entities of a software system. Two entities could be related with respect to different types of coupling. Furthermore, there may exist couplings of different strengths. Code entities usually are not an unordered set but are structured hierarchically. For instance, in a typical object oriented system, methods are grouped into classes, which are part of packages, which are themselves hierarchically organized. The hierarchy may help to structure the entities in the visualization.

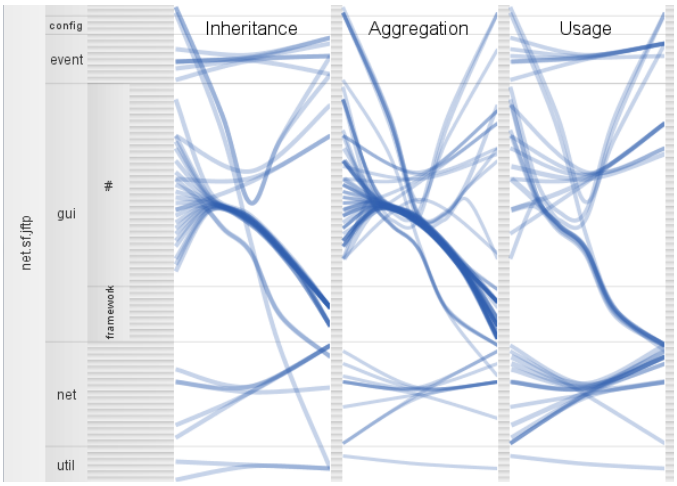

Fig. 1. A visualization of three types of structural code coupling in the JFtp project.

Formally, the problem is to visualize a *weighted compound graph with multiple types of edges* $G = (V, T, \mathcal{E})$ where V is the set of vertices of the graph. Furthermore, $T = (\hat{V}, \hat{E})$ is the hierarchy of the compound graph where $\hat{V}$ is the set of hierarchy vertices connected by inclusion edges $\hat{E} \subset \hat{V} \times \hat{V}$. The graph vertices V form the leafs of the hierarchy T (hence, $V \subset \hat{V}$). Finally, $\mathcal{E}$ is a family of sets over $V \times V \times \mathbb{R}^+$—different sets of weighted edges that describe the different coupling concepts. Non-weighted edge types can be modeled as having an edge weight of 1.

B. Requirements

Besides the obvious requirement of visualizing the data of the introduced graph structure, analyzing software projects poses additional challenges. One problem is the amount of data we usually have to deal with when analyzing non-trivial software systems. Our goal is to visualize at least mid-size software projects on the level of classes and interfaces, which we believe to be an appropriate granularity to explore the abstract structure of software projects.

Presenting the coupling data in a readable way is not equivalent to being able to compare different coupling concepts.

Hence, to foster this comparison task is an additional requirement for the visualization. A comparison can be supported by the layout of the visualization, but also interactions could play an important role.

In short, the visualization should be able *(data structure:)* to visualize a compound graph including different types of edges, *(scalability:)* to present the complete coupling structure of mid-size software projects on class level for a set of coupling types (up to 10 types), and *(task:)* to facilitate the comparison of the coupling concepts. Hence, the goal is to develop an expert visualization that meets these requirements and can be used by researchers to analyze the relationship between different code coupling concepts as well as by software engineers to understand, debug, and refactor their software systems.

The visualization we came up with is based on an a linearized node-link layout of graphs. The key contribution of this visualization technique is the scalable comparison of different edge types. This is implemented by space-efficient but still scalable representations of the single edge types, which are juxtaposed in a static image.

The remainder of this paper is structured as follows: We first discuss how far existing approaches already solve the introduced visualization problem (Section II). Then, we propose a new visualization technique specifically designed to fulfill the imposed requirements (Section III). Advanced interactions enable the user to visually compare the different edge types (Section IV). A case study shows how the visualization approach can be leveraged in practice (Section V). Finally, we conclude the paper (Section VI).

II. Related Work

UML class diagrams [1] are the industry standard to visualize code entities and their dependencies. In such diagrams, mainly three types of edges are distinguished: inheritance, aggregation/composition, and association. The edges are represented by visual links between the classes and interfaces of the modeled system; edge types are encoded in different types of arrow heads. Although this representation is intuitive, it is not applicable to visualize larger parts of a system because it does not scale well: The nodes of the diagram consume much space and only a selection of code dependencies can be visualized as links.

Software visualization research has proposed and discussed many other approaches to visualize code coupling. A selection is presented in a survey on software architecture visualization by Ghanam and Carpendale [2]. But since coupling among code entities can be considered as a graph structure, general graph visualization techniques can be applied as well, especially those that are optimized to display large graphs. A recent survey by von Landesberger et al. [3] provides an overview of the state-of-the-art in this domain including some node-link based approaches on comparing graphs. Although much has been achieved in visualizing code coupling and large graphs, we are not aware of a scalable approach that directly supports the comparison of different types of edges. Nevertheless, there exist some approaches that partly meet our requirements, which we introduce in the following.

Edge Bundling has been applied to improve the scalability of graph-based software visualizations [4], [5]. These approaches use a hierarchy on the nodes to control the bundling of edges and thereby profit from the inherently hierarchical structure of software projects. A problem, however, is that the technique of bundling conflicts with the representation of multiple edge types—bundling together edges of different types would seriously obfuscate the type; creating different bundles for different types in the same diagram would largely outweigh the positive effect of bundling.

Another way of integrating different types of edges or showing the evolution of edges is to stack several two-dimensional graph layouts on top of each other, which results in a three-dimensional layout. Equivalency between nodes in the different layers can be expressed by connecting them through additional links [6] or by aligning them vertically [7]. Here, occlusion affects readability in larger graphs.

Pretorious and Wijk [8] present an elaborate approach focusing on graphs with different types of edges. They represent each type as an additional node in the middle of the diagram. Then, edges of a particular type are routed through the corresponding edge type node. They show that the visualization can be efficiently used to understand a graph structure and retrieve information by interactive queries, even for large graphs and a considerable set of edge types. But it seems that most of the overview a graph visualization may provide is lost in the visual clutter of the static image.

Visualizing the graph as an adjacency matrix provides good scalability [9]. Recently, two approaches were proposed to represent different edge types in such matrices: Beck and Diehl [10] introduced an approach to compare two different types of coupling relations using different colors for the types. Showing more than two different edge types is possible, but would not be very readable in this approach. Moreover, Zeckzer [11] splits each cell of the matrix into n pieces, each representing a different edge type. This approach, however, decreases the scalability of the matrix by factor n.

A dynamic graph represents a graph that is changing over time and is usually modeled as a sequence of static graphs. The comparison of types of edges is related to visualizing dynamic graphs because a static stand-alone graph can be created for each edge type and concatenated into a sequence of graphs. But using animation, which is the standard approach in dynamic graph drawing, is not applicable in our application because a precondition for a readable animation is that only few things change between to subsequent states. Nevertheless, there exist some approaches that depict the dynamic graph in a single image without using animation [12], [13]. Better than animation-bases techniques, these approaches suit our application scenario of comparing different types of coupling, but they do not scale well [14].

Visually similar to our visualization approach are parallel coordinates plots, in particular when applying edge-bundling [15]. But they target a totally different application

as these diagrams represent multi-dimensional data instead of relational data. Another visually related, but not directly competing visualization technique is Code Flows [16]. Here, bundled flows between linearly arranged icicle plots depict the flow of code in source code documents over subsequent versions.

Concluding this review of related work, we see no approach that completely fulfills the imposed requirements. So far, node-link diagrams do not scale well or do not appropriately support the comparison of different types of edges. Matrix diagrams are more scalable but cannot distinguish more than two edge types without losing parts of this scalability. Finally, dynamic graph drawing approaches are not able to handle large difference in the structure of the edges or are not scalable enough.

III. Visualization Technique

In this section, we introduce a visualization technique based on node-link diagrams that is designed to meet our requirements (Figure 1). This visualization approach combines known techniques and some novel ideas to enable the interactive comparison of different types of edges in a scalable way. The basic layout of the approach is based on the TimeArcTrees visualization technique [13], a dynamic graph visualization that represents the dynamic as a sequence of diagrams shown in a single static image. A prototype of our new approach was implemented in Java using Processing.

A. Graph Visualization

Node-link diagrams are the straightforward way to visualize a graph structure. They consist of nodes—often circles or rectangles—representing objects and links—straight or curved lines—representing the relations between the objects. In our case, we want to compare different edge types (coupling concepts) based on the same set of nodes. The simple idea of overlaying different types on the same drawing area, however, is strongly limited with respect to the number of types due to overlap and interfering colors. Hence, we decided to juxtapose different diagrams, each representing a different edge type.

Juxtaposition, also known as *small multiples*, is an alternative to overlaying when comparing visual objects. As the term *small multiples* already indicates, a downside of this approach is the limited space assigned to each single diagram. Hence, the challenge is to display a potentially large graph in a small area of screen space. We tackle this problem by applying two tricks, one concerning the node layout and one concerning the edge routing.

1) Node Layout: Arranging a set of diagrams side by side on the screen leaves narrow stripes of screen space for each diagram. In a traditional node-link diagram as depicted in Figure 2 (a), matching the representation of the same node in different diagrams becomes quickly difficult for larger graphs because the user has to memorize the horizontal and the vertical position of the node. It would be much easier to just follow a horizontal line to get from one representation to the other. This idea suggests to linearly arrange the nodes onto an imaginary vertical line (Figure 2 (b)). A similar technique was already used in the TimeArcTrees approach [13]. But drawing links between the nodes results in massively overlapping arcs, crossing in small angles, which is particularly hard to read [17].

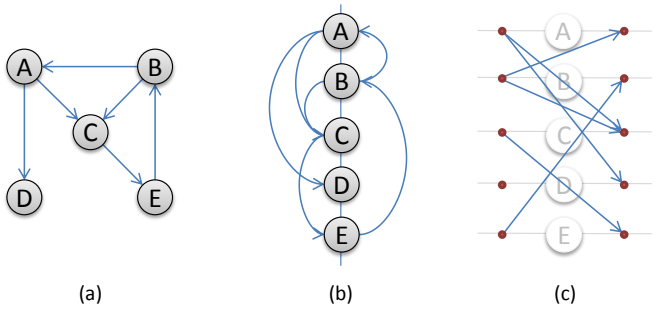

Fig. 2. A small sample graph in three different layouts: (a) a usual node-link diagram, (b) linearized nodes, (c) linearized nodes with split ports.

The key trick concerning the node layout is to introduce two ports for each node, one assembling the incoming edges, and one summarizing the outgoing edges. The two ports of a node normally would be positioned onto the borderline of the circle or rectangle representing the node. In contrast, our approach moves them apart from each other on an imaginary horizontal line as shown in Figure 2 (c). The result is that all edges head from left to right, in other words, from the ports of the outgoing edges on the left to the ports of the incoming edges on the right. It is still possible to easily follow these edges. The direction of an edge even becomes much clearer than in a usual node-link diagram. Moreover, the approach as presented so far is already quite scalable because each node only require a few pixels of the height of the drawing area. A drawback of this layout is that following paths in the graph becomes more difficult—this task, however, plays only a limited role in our application scenario.

2) Edge Routing: To further improve the scalability of the graph visualization, we apply hierarchical edge bundling [4], a technique to visually group edges into bundles according to a hierarchical organization of the nodes. The hierarchy is given, in our application example, by the hierarchical structure of the software system, in case of Java systems, the package structure. The resulting edge routing as shown in Figure 1 is similar to a work by Holten and van Wijk that compares two hierarchies using bundled edges [18]. The main difference to this approach is that, in our case, the two hierarchies are identical but the graph connects arbitrary nodes. The bundling approach simplifies the diagram and reduces visual clutter at the cost of obfuscating the trajectory of single edges that are summarized into bundles. This leads to a better overview on the graph, still preserving the outliers. The obfuscated details of an edge belonging to a larger bundle can be retrieved interactively as we explain in Section IV.

To retrieve the density of edges in a bundle, alpha blending makes the edges slightly transparent. The weight of an edge is encoded in the thickness of the line that represents the

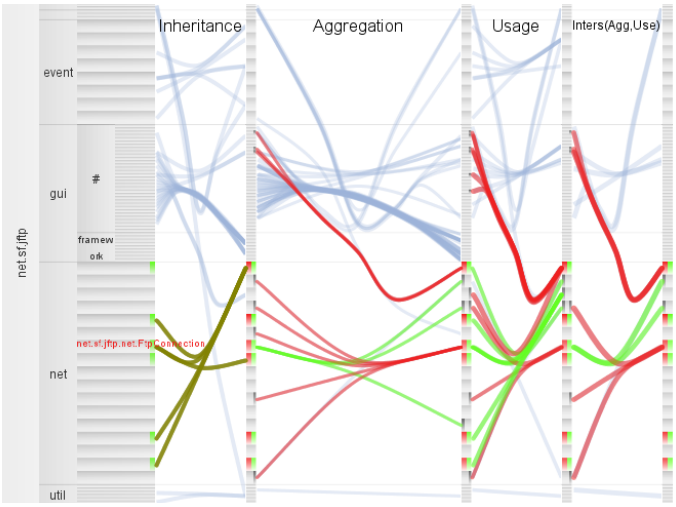

Fig. 3. Applying different interactions techniques to the example shown in Figure 1.

edge, which grows logarithmically with the weight. When comparing different code couplings, the visualization has to deal with totally different scales of weights. To overcome this problem, we normalize each edge type to a total weight of 1, or in other words, we divide each original edge weight by the total weight summed over all edges of the current type.

B. Hierarchy Visualization

The hierarchical structure of a software project not only provides a criterion to bundle edges, but also helps to linearly arrange the nodes onto the vertical lines. The hierarchy is shown at the left side of the diagram (Figure 1) as a layered icicle plot [19], a simple but scalable kind of hierarchy visualization. Nodes have labels if enough screen space is available. We continue the lines separating packages over the full width of the visualization so that it is easier to relate a particular node in one of the diagrams to the hierarchy. Furthermore, the vertical, separating lines between the diagrams are used to repeat the leaf level of the icicle plot.

IV. Interactions

The static diagram already allows comparing the different edge types in a relatively scalable way, but only adding interactions unleashes the full potential of the approach. Basic interactions that enable the users to query details, to focus and zoom the diagram are supplemented by advanced techniques that help comparing the different edge types. Figure 3 provides an impression on how the diverse interactive features may influence the visualization.

A. Basic Interactions

We cannot display labels for the leaves of the hierarchy because the height of their visual representation usually is only a few pixels. Hence, a very simple but essential interaction is to blend in labels for these entities when hovering the mouse over their representation. In Figure 3, the class `FtpConnection` is labeled on demand.

The users are able to focus one or more entities by clicking; when clicking on an inner node of the hierarchy all contained entities are focused. In the example provided in Figure 3, some elements of the `net` package are focused. Those focused nodes are highlighted in green and red in the icicle plot as well as in the separator of the diagrams. Moreover, edges that start or end at the focused entities also become part of the focus. Here, coloring provides the information whether the edge starts at a focused entity (green), ends at a focused entity (red), or both (brownish green)—red-green color blind users may choose a different color scheme. When entities are focused, all non-focused edges are still visible but are drawn in a lighter blue. To facilitate retrieving the non-highlighted sources or targets of highlighted edges, small gray markers identify these entities.

An interaction technique that largely improves the scalability of the visualization technique is zooming. The zooming feature as implemented in our tool is similar to those introduced by Table Lens [20]. It does not globally enlarge the diagram but always preserves the context of the zoomed area. Since multiple entities can be zoomed independently, horizontally as well as vertically, the technique allows setting multiple foci as demonstrated in Figure 3 where the `event` and the `net` package as well as the aggregation diagram are enlarged.

An interactive feature of particular interest when analyzing source code is to connect the visualization to the code. In our visualization, double-clicking on a class or interface opens an editor with the respective source code. Activating a package this way, the corresponding directory is opened in the file manager. This enables, for instance, to check whether particular edges exist or to browse through the content of a package in detail.

B. Interactively Comparing Types of Edges

Beyond the introduced basic interaction techniques, which improve the usability and scalability of the visualization tool, more advanced interactions directly aim at the task of visually comparing types of edges. In general, visual comparison techniques can be classified in three categories [21]: First, *juxtaposition* is based on placing the objects just next to each other. Second, *overlaying* arranges the objects being compared in different layers on top of each other. And third, *fusion* creates a new object out of the compared ones. These three techniques do not exclude each other, but can be combined into a staged comparison mechanism as we demonstrate in the following.

1) Juxtaposition: As discussed in Section III, the diagrams representing the different edge types are placed side-by-side. The node layout is globally consistent over the diagrams so that these diagrams can be directly compared to each other. Thus, comparisons based on juxtaposition are inherently enabled by the visualization. To further improve on this technique, the edge diagrams can be moved so that the diagrams of interest become neighbors.

2) Overlay by Focusing: We extended the functionality to focus and highlight sets of entities to implement a weak form of overlay comparison. The idea is to focus a set of entities by certain characteristics of one type of edges. Since the focus is globally applied, this is a kind of overlay of the local focusing criterion onto all other types.

We propose different algorithms to add the focus to or remove the focus from the set of currently focused entities with respect to the edges of a selected type: A basic characteristic is whether an entity has any incoming or outgoing edges of a particular type. For instance, all classes that are inherited by some other class could be focused this way. Moreover, the reachable entities with respect to the outgoing or incoming edges of the type starting from the currently focused entities can be added to the focus. Using this strategy, the user may focus a connected component in the graph. This is demonstrated in Figure 3 with respect to inheritance starting from the `FtpConnection` class. Complementing the advanced focusing features, reverting and resetting the set of focused entities is possible.

3) Merging: Comparing different types of edges by merging means to generate a new type of edges that aggregates the information. Since each edge type is represented as a set, common set operations can be applied.

- *Union:* All information contained in two or more types of edges is summarized into one type. The user may choose to sum up or take the maximum weight as the edge weight for the united type.
- *Intersection:* Intersecting two or more types leaves over those edges that concurrently belong to all merged types. Here, the edge weight can be set to the sum, the maximum, or the original weight of one of the merged types.
- *Difference:* The difference of two types deletes the edges belonging two both types from one of the types. Hence, this merging operation is asymmetric and can be applied in two directions. The aggregated edge weights are automatically set to the weight of first type.

Applying the merging operation, a new diagram is generated by default. Though aggregated, it can be still retrieved which edges are of which type by looking at the original diagrams. If one of the former types should be replaced, the user is able to delete the respective graph after merging. In Figure 3, an intersection of the aggregation and usage edges was applied, which added the rightmost diagram.

The three comparison approaches—juxtaposition, overlay, and fusion—can be considered as different stages of escalation as they are ordered according to their invasiveness: While a comparison based on juxtaposition is even possible without changing the diagram, overlaying is implemented by highlighting some entities temporarily, and fusion means creating persistent, new data structures. For instance, a common visual pattern among two juxtaposed diagrams may raise an assumption, which could be checked using the advanced focusing mechanism and may finally be recorded applying a merge operation.

V. Case Study

The case study aims at showing how to apply the introduced visualization technique in practice. Beyond providing examples, we also try to identify typical activities a developer or researcher may perform when analyzing the code couplings of a software project.

A. Data Set

The software project that we take as an example is Checkstyle, a popular software to check coding conventions of Java source code. It is itself written in Java and released under an open source license. Version 5.1, which is the version we analyze, consists of 261 classes and interfaces grouped into 21 packages. Although our visualization technique is able to visualize larger data sets, we chose this project of moderate size for the paper because reading the visualization is more difficult in a printed, static version than in the interactive version on the screen.

We extracted five different types of code couplings.

- *Inheritance:* This type models the inheritance or implementation relation between classes and interfaces. Each edge of this type has weight 1.
- *Aggregation:* Aggregating a class or an interface means that another class uses this entity in the declaration of a field. The weight counts the number of fields using the particular type.
- *Usage:* We agglomerate all other structural code couplings except of inheritance and aggregation in this type including method calls and usage as local variables or method parameters. The weight counts the number of methods using the particular class.
- *Co-Change:* If two classes or interfaces were changed together in a transaction of the version archive they were co-changed. The number of co-changes denotes the weight of the edge.
- *Code Clones:* Code clones are identical or similar code fragments that were probably created using copy-and-paste. Two classes are coupled by clones if they share a code clone. The weight is provided by the amount of clone overlap, which is a value between 0 and 1.

The first three types are extracted from the bytecode using the tool *DependencyFinder*, the co-change information is mined by analyzing the transactions retrieved from the SVN code repository, and the clone information is collected by searching exact clones (type I clones) using the Java API *JCCD*.

Figure 4 visualizes the described data set. Starting from the default view, only two interactions were applied: The `api` package was highlighted because of its central role in the following analyses. Furthermore, inheritance and code clone couplings were intersected for a detailed analysis.

B. Detecting Coupling Features

We provide examples of what features of the coupling structure might be of interest for software developers and researchers when analyzing code couplings.

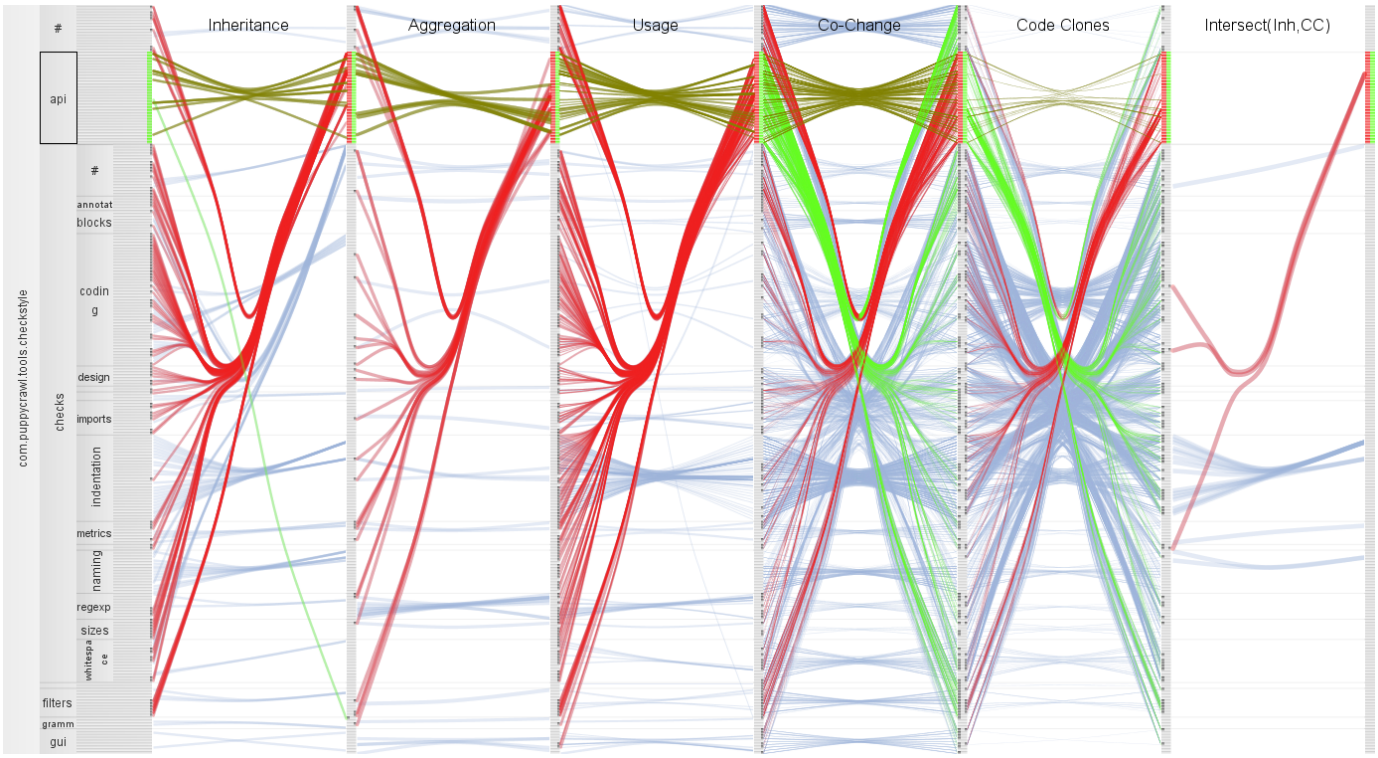

Fig. 4. Five types of couplings plus one merged type of couplings for the Checkstyle project; the `api` package is focused.

1) Key Classes: Classes that are coupled to many other classes form the key classes of a software system—depending on the coupling concept we refer to, they may agglomerate important features or changing them may also require changing many other files. Key classes can be identified by a high out-degree or in-degree of edges. In the visualization, such a high degree manifests in large fan-in or fan-out patterns of edges.

In the example of the Checkstyle project (Figure 4), we observe for inheritance coupling significant fan-in patterns, for instance, for some classes of the `api`, `indentation`, and `naming` packages. In the context of inheritance, such fan-in patterns identify classes that are extended by many other classes or interfaces implemented by many classes. In contrast, fan-out patterns cannot be found for this type of edge because classes are only allowed to extend one other class and usually implement only a few interfaces if any.

Comparison: The visualization allows comparing visual patterns such as a fan patterns across different edge types. For instance, the diagrams for aggregation and usage show fan-in patterns very similar to the ones found for inheritance. But when comparing inheritance to code clones, we cannot retrieve such a direct correlation directly: In the case of code clones, too many fan patterns overlap to clearly discern the bundles. But looking at the ports where the edges start or end, the user is still able to judge by the thickness of the bundle whether there might be a relevant fan pattern. Then, focusing the node of interest clearly would reveal the fan. For instance, Figure 4 shows code clone fan-in patterns in red and fan-out patterns in green for the focused `api` package. Our first impression was that the red pattern might be congruent to the one in the inheritance graph. We hence intersected the two types, inheritance and code clones, which results in the diagram attached as the last diagram of Figure 4, and found that both types are not very congruent with respect to the `api` package. Nevertheless, the last diagram also shows that there exists one significant fan-in pattern (in blue) for both types in the `indentation` package—a developer might want to check whether all these clones are necessary within this inheritance structure.

2) Coupling Outliers: Although simplifying the diagram, edge bundling does not obfuscate outliers. Such an outlier is an edge or a small set of edges that follows a certain path through the diagram without being bundled with any other edges. Outliers are in practice often unwanted, error-inducing or at least notable couplings. In Figure 4 the single inheritance edge from the `api` package to the `grammars` package (highlighted in green) is an outstanding example: It is even the only outgoing edge connecting the `api` package to another package by inheritance.

Comparison: Now, it would be interesting to know whether the `api` package is linked to other packages by aggregation or usage, the two other forms of structural coupling. At first glance it seems that there is a equivalent outlier connecting the `api` package to the `grammars` package among the usage couplings because there an edge is following a very similar path. But focusing the `api` like done in Figure 4 exposes that this coupling does not start at the `api` package because

otherwise it would be highlighted. Moreover, we see that there exist no further couplings in the first three diagrams that connect the `api` package to other packages—the detected outlier of inheritance might indeed be a potential flaw in the architecture of the system. An alternative way to retrieve this information would have been to aggregate the first three types of coupling and to analyze the package with respect to this summarized information.

3) Loose Ends & Independent Components: Nodes that have no incoming edges or outgoing edges represent the loose ends of a coupling graph. These could be classes that are independent of other classes (no outgoing structural code couplings), entry points for the program (no incoming structural code couplings), classes that were never changed (co-change coupling), or classes not covered by any code clones. Those loose ends that either have no incoming or outgoing edges form the borders of independent components in the graph structure, which might indicate, for example, independent subsystems or independent parts of development.

Comparison: The highlighted `api` package might be a loose end with respect to the structural couplings (inheritance, aggregation, and usage) as it contains many nodes that have no outgoing edges, at least not to other packages. But the single outlier, which we already described, may indirectly connect the `api` package to larger parts of the system. To check this, we aggregated inheritance, aggregation, and usage couplings into a single edge type (not documented in Figure 4). By applying one of the assisted focusing features, we then added all reachable entities following the outgoing edges to the focus. The result was that only a single interface in the `grammars` package was reachable. We thus localized the potential problem and propose to consider moving this interface from the `grammars` package to the `api` package.

4) Cohesiveness and Coupling of Packages: A good modular structure of a software system is said to follow the principle of low coupling and high cohesion [22]. Applying this principle to a package structure, there would exist only few edges between different packages (low coupling), but more edges within a package (high cohesion). In our visualization, edges within a package manifest through a simple horizontal bundle of edges. All other edges, which usually are summarized to a bundle on a higher level, connect different packages vertically and thereby account to the coupling.

Taking the usage edges as an example in Figure 4, we find such horizontal bundles expressing high cohesion values mainly for the `api` and `indentation` package. Quite outstanding, nearly all edges that cross package borders head towards the `api` package, which seems to provide an interface to an external library. Hence, with respect to usage, the systems seems to be well organized.

Comparison: Also considering other types of edges, we find that inheritance and aggregation edges show very similar patterns with respect to cohesiveness and coupling. In contrast, the co-change and code clone edges provide totally different structures because, in general, more edges are included and the graphs are symmetric. But despite the high edge density we see differences: For instance, the `indentation` package is quite cohesive for co-change. At the same time the package is lowly coupled because it does not have too much co-change connections to other packages. In contrast, for code clones, this `indentation` package appears to be coupled to other packages much more than it is cohesive. Furthermore, if a package is not cohesive with respect to any of the edge types, this can be an indicator for a badly designed package, which needs to be restructured—the `sizes` package is such a candidate for restructuring as no type of coupling reveals any notable cohesion.

5) Layered Architectures: The architecture of software systems often follows a layered design. The idea behind these layers is that the direction of coupling only goes from the top layers to the low-level layers. Our visualization helps analyzing a layered architecture because it clearly shows the direction of the edges and groups the classes according to the packages they are contained in.

Comparison: For checking the layers in our example presented in Figure 4, the directed, structural couplings by inheritance, aggregation, and usage are of particular interest. We already observed certain outliers among those types of coupling, which could be potentially violating the architectural layers. Moreover, large cross-cutting fan-in patterns indicate certain layers: Only the `api` package is accessed from all over the project and hence seems to be assigned to the lowest layer. Ignoring the few outliers, all other packages seem to belong to the same layer—a minor exception is that there exist couplings between some sub-packages of the `checks` package and the classes that are directly included in the `checks` package (represented by the `checks.#` package). Allowing the user to reorder the packages or applying an automatic sorting algorithm may further enhance the visualization with respect to this scenario but is not implemented yet.

C. Applied Comparison Strategies

The examples we provided showed useful application scenarios for a visualization of code couplings. While some facts could have been also found by using a visualization technique based on a single type of couplings, the retrieval of many findings relies on or at least is supported by the comparison abilities of the visualization. We finally want to summarize the applied comparison strategies, which can be considered as a set of recipes to use the visualization.

1) General Characteristics: The first and very simple strategy is to look at the whole picture and retrieve some general characteristics of the edges of the different types at a glance. For example, we easily see how dense the edge structures are, whether a type of edge is directed or not, whether there exist many edges that connect the classes of the same package, etc. This comparative overview on the whole data set calls the attention of the user to similarities and differences, which can be analyzed in detail in further steps.

2) Equivalent Structures: Key classes and outliers are features of a single graph, which can be retrieved searching for certain visual patterns in one of the diagrams of a single edge

type. A major strength of the presented visualization is to enable the users to check whether these patterns also exist for other types of edges without the need to manipulate the diagram. If the first check was positive, the users might want to refine the analysis—the visualization provides different features to facilitate this: The users could move the two types next to each other and enlarge both. Focusing certain nodes enables to prove that a similar looking visual pattern indeed covers the same classes. Moreover, the merging features provide a tool to globally compare different edge types.

3) Focusing the Analysis: Due to a previous observation or certain knowledge of the project, users might have ideas what could be parts of the system or combinations of types of particular interest. The visualization allows focusing the analysis so that users are able to check these details. For instance, we highlighted the `api` package in Figure 4 because the large fan-in pattern looked suspicious, and in the following, the highlighting revealed interesting details with respect to outliers and layers of the architecture. Examples for focusing on the comparison of two types are the findings that shed light on the relationship between code clones and inheritance.

VI. Conclusion

We introduced a visualization approach to compare different types of coupling data that connects code entities with each other modeled as a directed compound graph containing different edge types. Since software projects create large data sets, the design of the visualization technique thoroughly takes scalability issues into consideration by splitting ports for incoming and outgoing edges as well as by applying edge bundling. The basic layout of the visualization enables the user to compare the different edge types based on juxtaposed diagrams. Moreover, interactions integrate more advanced comparison techniques based on overlay and fusion.

The case study on a mid-size project provides recipes of how the visualization can be used for analyzing the multi-dimensional couplings of a software project. The interactive comparison features help retrieving information that would have been difficult to get using other visualizations. The diverse application scenarios that we demonstrated in the case study shows that the visualization can be applied by software developers to improve or understand their software system as well as by researchers for understanding the relationships of different kinds of code couplings. Beyond that, we see other scenarios where the introduced visualization technique might be profitably applied. For instance, different types of relationships between the individuals of a large social network might be compared.

Acknowledgment

The authors would like to thank Michael Burch for the fruitful discussions in early phases of this work.

References

[1] G. Booch, J. Rumbaugh, and I. Jacobson, *Unified Modeling Language User Guide, The (2nd Edition) (Addison-Wesley Object Technology Series)*. Addison-Wesley Professional, 2005.

[2] Y. Ghanam and S. Carpendale, "A survey paper on software architecture visualization," University of Calgary, Tech. Rep., 2008.

[3] T. von Landesberger, A. Kuijper, T. Schreck, J. Kohlhammer, J. J. van Wijk, J. D. Fekete, and D. W. Fellner, "Visual analysis of large graphs," in *12th Joint Eurographics/IEEE-VGTC Symposium on Visualization*, 2010.

[4] D. Holten, "Hierarchical Edge Bundles: Visualization of Adjacency Relations in Hierarchical Data," *IEEE Transactions on Visualization and Computer Graphics*, vol. 12, no. 5, pp. 741–748, 2006.

[5] A. Telea, H. Hoogendorp, O. Ersoy, and D. Reniers, "Extraction and Visualization of Call Dependencies for Large C/C++ Code Bases: A Comparative Study," in *VISSOFT '09: Proceedings of the 5th IEEE International Workshop on Visualizing Software for Understanding and Analysis*. IEEE, 2009, pp. 81–88.

[6] M. Baur, U. Brandes, M. Gaertler, and D. Wagner, "Drawing the AS graph in 2.5 dimensions," in *Graph Drawing*, ser. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2005, vol. 3383, pp. 43–48.

[7] O. Greevy, M. Lanza, and C. Wysseier, "Visualizing live software systems in 3D," in *SOFTVIS '06: Proceedings of the 2006 ACM symposium on Software visualization*. New York, NY, USA: ACM, 2006, pp. 47–56.

[8] A. J. Pretorius and J. J. van Wijk, "Visual inspection of multivariate graphs," *Computer Graphics Forum*, vol. 27, no. 3, pp. 967–974, 2008.

[9] 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*, 2004, pp. 17–24.

[10] F. Beck and S. Diehl, "Visual comparison of software architectures," in *SOFTVIS '10: Proceedings of the ACM 2010 Symposium on Software Visualization*, Salt Lake City, Utah, USA, 2010, pp. 183–192.

[11] D. Zeckzer, "Visualizing software entities using a matrix layout," in *SOFTVIS '10: Proceedings of the 5th international symposium on Software visualization*. New York, NY, USA: ACM, 2010, pp. 207–208.

[12] M. Burch and S. Diehl, "TimeRadarTrees: Visualizing dynamic compound digraphs," *Computer Graphics Forum*, vol. 27, no. 3, pp. 823–830, 2008.

[13] M. Greilich, M. Burch, and S. Diehl, "Visualizing the Evolution of Compound Digraphs with TimeArcTrees," *Computer Graphics Forum*, vol. 28, no. 3, pp. 975–982, 2009.

[14] F. Beck, M. Burch, and S. Diehl, "Towards an Aesthetic Dimensions Framework for Dynamic Graph Visualisations," in *IV '09: 13th International Conference on Information Visualisation*. Los Alamitos, CA, USA: IEEE Computer Society, 2009, pp. 592–597.

[15] H. Zhou, X. Yuan, H. Qu, W. Cui, and B. Chen, "Visual clustering in parallel coordinates," *Computer Graphics Forum*, vol. 27, no. 3, pp. 1047–1054, 2008.

[16] A. Telea and D. Auber, "Code flows: Visualizing structural evolution of source code," *Computer Graphics Forum*, vol. 27, no. 3, pp. 831–838, 2008.

[17] W. Huang, S.-H. Hong, and P. Eades, "Effects of crossing angles," in *Proceedings of the IEEE VGTC Pacific Visualization Symposium 2008*, 2008, pp. 41–46.

[18] D. Holten and J. J. van Wijk, "Visual Comparison of Hierarchically Organized Data," *Computer Graphics Forum*, vol. 27, no. 3, pp. 759–766, 2008.

[19] J. B. Kruskal and J. M. Landwehr, "Icicle Plots: Better Displays for Hierarchical Clustering," *The American Statistician*, vol. 37, no. 2, pp. 162–168, 1983.

[20] R. Rao and S. K. Card, "The table lens: merging graphical and symbolic representations in an interactive focus + context visualization for tabular information," in *CHI '94: Proceedings of the SIGCHI conference on Human factors in computing systems*. New York, NY, USA: ACM, 1994, pp. 318–322.

[21] F. Beck, S. Diehl, T. Dwyer, M. Gleicher, C. Hansen, I. Jusufi, K.-L. Ma, A. Perer, J. C. Roberts, J. Yang, and D. Zeckzer, "Dagstuhl Seminar on Information Visualization (10241), Results of the Working Group on *Comparison in Infovis*," 2010. [Online]. Available: http://www.dagstuhl.de/Materials/index.en.phtml?10241

[22] W. P. Stevens, G. J. Myers, and L. L. Constantine, "Structured design," *IBM Systems Journal*, vol. 13, no. 2, pp. 115–139, 1974.