University of West Bohemia
Department of Computer Science and Engineering
Univerzitni 8
30614 Pilsen
Czech Republic

Large component diagrams visualization

The State of the Art and the Concept of Ph.D. Thesis

Lukáš Holý

Technical Report No. DCSE/TR-2012-08
July, 2012

Distribution: public

Technical Report No. DCSE/TR-2012-08
July 2012

Large component diagrams visualization

Lukáš Holý

Abstract

Software applications applications can easily consist of hundreds or thousands of components and it is thus difficult to understand their structure. Diagram visualisation does not help much because of visual clutter caused by big amount of elements and connections, especially in the case of flat component models. This work sums up current state of the art tools and approaches in component diagrams visualization. After that we propose a set of criteria for the evaluation of tools for component architecture visualization. Furtermore we present a novel approach which should ease the orientation and navigation in complex diagrams. It is among other benefits useful in the reverse engineering process. One of the key concepts of this approach is removing a large part of connections from the diagram while preserving the information about component interconnections. We also describe a viewport technique for use in the visualization of UML component diagrams. This technique should ease the work with complex diagrams by highlighting details of the important parts of the diagram and their related surroundings without losing the global perspective. The further aim is to integrate mentioned techniques and new techniques into one application.

The work was supported by the UWB grant SGS-2010-028 Advanced Computer and Information Systems.

Copies of this report are available on
http://www.kiv.zcu.cz/publications/
or by surface mail on request sent to the following address:

University of West Bohemia
Department of Computer Science and Engineering
Univerzitni 8
30614 Pilsen
Czech Republic

Contents

Chapter 1

Introduction

This work focuses on the effective visualization of large system component models. New methods of the visualization should bring clarity of represented data. These methods should support user interaction with the model for better customization according to user needs.

1.1 Problem Definition

1.1.1 Component Software Modeling

Software architects and developers have been using various forms of visualizing the structure of software applications since the advent of the discipline. In the last 20 years, the increased adoption of object-oriented programming lead first to several proposals for adequate modeling notations which were then gradually consolidated into the current standard – the Unified Modeling Language (UML) [54]. While UML is able to model both the static and dynamic aspects of many kinds of software, recent development in the field of component-based software engineering (CBSE) brings new challenges.

The visualization of component-based applications [65] is not a trivial task due to the rich structures of component interfaces and the differences between component models. Frameworks like EJB [64], CORBA [53], OSGi [55] and more can be found in commercial applications and even more component models – for example SOFA [13], Fractal [50] or CoSi [11] – are the subject of research.

The diversity of component models in terms of the features available on component interface is well described in e.g. [21]. On an abstract level, components have in common two basic properties: the black-box nature and the fact that the features they need and provide on their interface are well defined [65].

Their interface features can cover all known contract levels [8]:

- syntactic, e.g. functional interfaces in most models and events in EJB3 [64],
- semantic, e.g. triggers in SaveCCM [32],
- behavioural like protocol in SOFA [56],
- extra-functional property specifications, e.g. in Palladio [6],
- control interfaces like in Fractal [50].

This richness indicates that modeling and visualizing component applications is a challenging task.

1.1.2 Diagram Complexity

Software applications become more and more complex and although there are lots of tools which help the development process, they are still limited in helping human understanding of the application structure. Software components [65] are one of the ways to handle this complexity as they encapsulate parts of functionality to unified components. Even with the usage of the components, nowadays applications can easily consist of hundreds or thousands of them. It is therefore difficult to explore the structure of the application and create a mental model of the whole system.

One of the ways how to get an insight into a component application structure can be a diagram, eg. UML component diagram. When the diagram is large there are many problems with exploring it. One is the contradictory need of providing enough details and showing the complete diagram (application structure) at the same time. Diagrams displayed at the desired level of detail become too big to provide a sufficient overview and keep orientation; especially difficult is to trace dependencies between distant components.

Another question is how to reduce visual clutter [60] caused by the large number of elements and connections between them. The visual clutter makes tracing of dependencies difficult and hinders orientation in the diagram. Current tools do not offer features designed for work with such large diagrams [40].

It is possible to divide large diagram into smaller ones. But in this case user would lose the overview of the whole system and the information about interconnections among system parts. Although diagrams of hierarchical component models [13] usually does not have this problem because they keep the information about parts in their hierarchy, there is a lot of component models [64],[55] with flat structure where the described problem occurs.

1.2 Goal of the Work

The main goal of the thesis is to bring better ways of large component software visualization to increase the understanding of the application structure. We will mainly focus to node-link diagram representation, because it is intuitive and well known eg. in form of UML component diagram. Node-link diagrams are widely used in various domains and thus inventing new techniques in software visualization can be potentially generalized for graphs used in other domains.

We will use the existing ComAV tool which is a platform for visualization and reverse-engineering of component-based applications. Data aquired by this tool can be visualized by discovered techniques. We will also provide an implementation of discovered techniques to enable their further verification.

Chapter 2

Background and Related Work

2.1 Component Based Software Development

Szyperski defined components in [65] as following:

> A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.

Bachmann [5] states:

> Component-based software engineering is concerned with the *rapid assembly* of systems from components where: components and frameworks have certified properties; and these *certified properties* provide the basis for *predicting the properties of systems* built from components.
>
> Component is [5]:
>
> - subject to third-party composition,
> - an opaque implementation of functionality,
> - conformant with a component model.

Component based software development (CBSE) should speed up the development of new software by reusing the existing component. These can be developed by third parties. It should also increase the predictability of produced application. On the other hand there is some overhead in wraping functionality into components. User of third party components should check the changes of used components' versions. There is also diversity in component models and frameworks, which slows down the growth of large market.

2.1.1 Component Models and Frameworks

Bachmann [5] explains terms component model and framework as: The component model gives a uniformity to components and their composition. Its use is to define how a component should look like, how components communicate each other, which resources they use, etc. The component model ensures the components are compatible in terms of deployment, the communication, etc. It determines the rules components must hold to be able to cooperate and it minimalists misunderstood assumptions. A component framework is basically an implementation of a component model. It supports all mechanisms such as deployment, synchronization, life-cycle, communication of components which are defined in the component model.

Component models will impose standards and conventions of the following kind:

- component types,
- interaction schemes,
- resource binding.

A component framework is basically an implementation of a component model. It supports all mechanisms such as deployment, synchronization, life-cycle, communication of components which are defned in the component model.

2.1.2 Compositional Forms

The compositional forms influence the features needed in a tool used for component diagram visualization. There are compositional forms described in [5] as:

Component Deployment

Components must be deployed into frameworks before they can be composed or executed. The deployment contract(s) (as shown at point 1 in Figure 2.2) describes the interface that components must implement so that the framework can manage their resources.

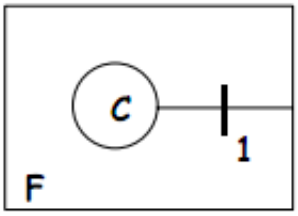

Figure 2.1: Component Deployment[5]

Framework Deployment

Frameworks may be deployed into other frameworks. Contract is analogous to the component deployment contract.

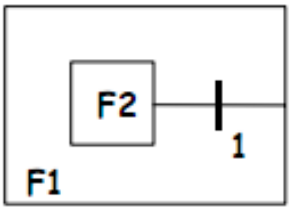

Figure 2.2: Framework Deployment[5]

Simple Composition

Components deployed in the same framework can be composed. The composition contract expresses component- and application-specific functionality; the interaction mechanisms to support this contract are provided by the framework.

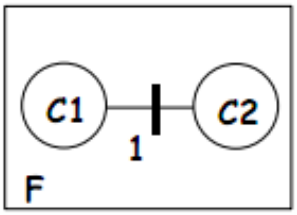

Figure 2.3: Simple Composition[5]

Heterogeneous Composition

Support for tiered frameworks implies composition of components across frameworks, whether across hierarchical (as illustrated in Figure 2.4) or peer frameworks. In either case bridging contracts are needed in addition to composition

contracts (as shown at point 2 in Figure 2.4) in order for interactions to span generic component models.

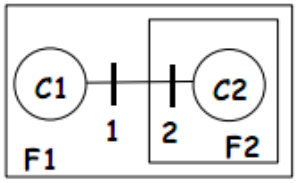

Figure 2.4: Heterogeneous Composition[5]

Framework Extension (Plug-In)

Frameworks may be treated as components, and may be composed with other components. This form of composition most commonly allows parameterization of framework behavior via plug-ins. Standard plug-in contracts for service providers are increasingly common in commercial framework.

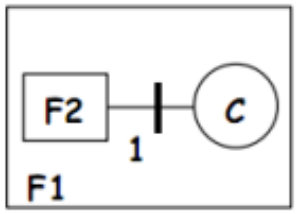

Figure 2.5: Framework Extension (Plug-In)[5]

Component (Sub)Assembly

A component-based system is an assembly of components. The ability to predict the properties of assemblies suggests a similar ability for subassemblies. Contract is used to compose C1 and subassembly C3, which contains one or more components. A question that arises is whether C2 is visible outside of C3 and whether it is separately deployed.

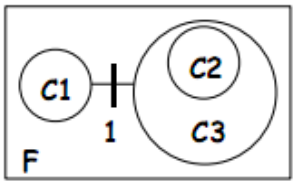

Figure 2.6: Component (Sub)Assembly[5]

Most common compositional forms are component deployment and simple composition, which we can find for example in OSGi[1]. Component (sub)assembly is the form represented in hierarchical component models such as SOFA 2[2]. The idea of a framework deployment form can be found for example in SpringDM[3] deployed in OSGi. In this situation we can find the idea of heterogenous composition between SpringDM and OSGi components. It is also possible to extend SpringDM by components running in the OSGi framework (eg. Equinox[4]).

2.1.3 Contracts

The interfaces are used for communication among components. There are several languages for interface description according to [22]:

- modelling languages (such as UML or different ADLs),
- particular specification languages (Interface Definition Languages),
- programming languages (such as interfaces in Java),
- some additions built directly in a programming language.

There can be also different types of iteraction [22]:

- port-based where ports are the channels for communication of different data types and events;
- functions in programming languages defining input and output parameters;
- interfaces or classes in Object Oriented programming languages.

The interfaces provide most of time a basic description of services and thus there are contracts for better describtion. Contracts among components should guarantee good interface connecting and determite "rights and duties"" of components involved. Contracts can be negotiated by involved sides and can be also changed in runtime, if all sides agree. They can also expire.

Contract definition according to [65]:

> A contract (an interface together with its specification) mediates between independently evolving clients and providers of the services the interface makes accessible.

[1] http://www.osgi.org/
[2] http://sofa.ow2.org/
[3] http://www.springsource.org/osgi/
[4] http://www.eclipse.org/equinox/

There can be following levels of contracts, according to [9]:

- **Syntactic** (or basic) The goal is to make the system work. It is generally specified with Interface Definition Languages (IDLs), as well as typed object-based or object-oriented languages. It ensures the components can be assembled.
- **Behavioral** The goal is to specify each operation. It is generally specified with a couple of assertions: a precondition and a postcondition. It ensures the operations offered and required are not only syntactically compatible but also semantically.
- **Synchronization** The goal is to specify the coordination of operations. It can be specified with an automaton labelled with operations. It ensures the operations are used in the proper order.
- **Quality of Service** The goal is to quantify a few features associated to operations. Performance, availability and quality of result can be specified and negotiated at that level.

Bachman [5] distiguishes between *component contracts* and *interatction contracts* and defines them as following:

- A **component contract** specifies a pattern of interaction rooted on that component. The contract specifies the services provided by a component and the obligations of clients and the environment needed by a component to provide these services.
- An **interaction contract** specifies a pattern of interaction among different roles, and the reciprocal obligations of components that fill these roles.

For purposes of this work we limit us to visualize the interfaces as defined in UML component diagram.

2.1.4 Component Diagramming - State-of-the-art

There are many software architecture modeling tools and visual syntaxes and their use is very common in practice. But a closer look reveals the lack of good model representations. On one hand, a generic representation like UML provides insufficient support for component-specific needs, on the other hand, tools focused on component development sometimes force the user to learn new visual syntax specific for the component model. Advanced features offered by the tools on top of basic architecture visualization are often limited.

Most commonly used visual language for displaying component applications structure is UML component diagram [52]. Its main features are:

- components,
- provided and required interfaces,
- stereotypes,
- tagged values,
- notes,
- hierarchy of inner component,
- including ports as parent's component interfaces.

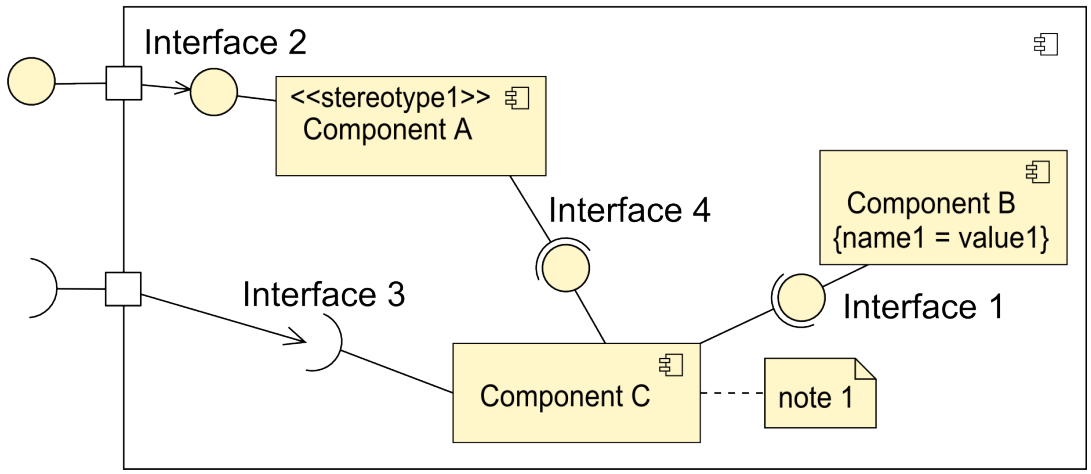

Figure 2.7: UML component diagram

2.2 Software Visualization

Visualization is very effective way in understanding software structure, behaviour or evolution. This section describes related software visualization approches and techniques, which help to increase understanding the software. When visualizing complex structures we usually face the problem of not having enough space on the screen to visualize the whole diagram in the desired level of details. Thus we are forced to use some technique to navigate through such a large diagram while showing only part of it on the screen.

2.2.1 Information Schemes

There are several main approaches while dealing with the complexity problem [20]:

- overviev and detail,
- pan and zoom,
- focus and context.

These principles can be combined together to offer a user good understanding of large diagram.

Overview and Detail

This approach is very comonly used in the software diagram tools as well as other visualizing fields like maps, CAD systems etc. Its main principle is to provide user two or more views with different level of details. Most common is using the detailed view for most of the screen area while the overview area is smaller for ensuring orientation. This approach is useful in large diagrams, but its scalability for very large diagrams is limited. It can be partly improved by using more overview levels, but it decreases the transparency of the whole approach.

Pan and Zoom

This approach is used for providing the ability to view a desired part of the diagram in desired level of detail. The panning feature usually moves the undelaying diagram according to mouse movements. The zooming feature provides the ability to see the diagram in different levels of size and detail. It is usually handled by mouse wheel, plus and minus keys or buttons dedicated for mouse control. This approach shows the focused and contextual information in views, which are in fact separated by time.

Focus and Context

This approach combines the both focus and context information into one view. Focused area shows detailed information, the context area shows the relevant contextual information and they are seamlessly integrated into one view. This integration can be achieved by several techniques such as fisheye distorsion, using the border for various type of marks or showing proxy elements for hidden objects. This aproach differs from the overview and detail in showing the detail view right

in the diagram where the less detailed information are shown. In the contextual part can be also shown information which lie away from the focused area. These information cannot be easily shown by overview and detail or pan and zoom approaches.

Animation

Animating the changes between showing different views helps user to better understand shown diagram. It can be used for various changes such as changing zoom [7] [68] level, moving between distant nodes in the diagrams or moving the elements during diagram modifications. Important factor while using animation is the time an animation takes. Longer time leads to better understaning of content, but it can slow down work with the tool. Appropriate values for the animation are suggested between 300 and 1000 miliseconds in [43]. Also the work of [69] about optical flow reduction can be helpful for this problem.

2.2.2 Graph Layouts

The node layout of the visualized diagram graph can significantly increase the understanding of the application. There are many methods for graph layouts such as:

- force-directed,
- orthogonal,
- circular,
- tree,
- layered.

Above mentioned layout methods are briefly described in following paragraphs. Our current knowledge about layouts is based on preliminary tests, discussions in community and several overviews and evaluations in publications [23], [1], [17],[34],[57], [31]. More rigorous choice of suitable algorithm for component diagrams is a subject for further research.

Force-directed layouts

For component diagrams visualization are suitable force-directed graph-drawing methods. In these methods the nodes' layout is computed according to under-

laying physical model. The iterative algorithm computes the nodes' placements until the energy in the whole system is minimal.

Classical force-directed algorithms like [28], [42] are suitable for drawing general graphs. They are also used in practice [12] for graphs containing hundreds of vertices. There are also available more efficient force-directed techniques for even larger graphs (tens of thousands of nodes)[33], [70].

Orthogonal layouts

Orthogonal methods are using only horizontal and vertical directions for drawing the edges. Therefore it can be tedious to trace dependencies in large diagram while having a detailed view.

Circular layouts

Circular layouts place the nodes on the circle and the edges connects them inside or outside a circle. The edges can be drawn straight (inside a circle) or bended. Also an edge bundling techniques (see Section 2.2.4) are suitable to be used for this layout. Nodes in this layout can be placed on the circle to minimize the edge crossings.

Tree layouts

Tree layouts are suitable for drawing tree graphs. Usually the root of a tree is drawn in the middle and its children are placed around it. The component diagrams are usually not in a tree structure, thus this category of layouts is unlikely to be used for implementation

Layered layouts

Layered layouts are suitable for acyclic or nearly acyclic graphs. They place the nodes into horizontal layers. Layers are connected among each other and the nodes in each layer are placed to minimize lines crossings among layers.

2.2.3 Nodes Visualization

Nodes represent individual components in UML component diagrams. From visualization point of view there are several main node factors, which can be adjusted to express desired metrics or attributes:

- dimensions - such as width, height (or depth in 3D),
- colors - including various color effects eg. trasitions,
- shape.

While using various node dimensions and colors can be in accord with visual syntax of used model, changing the shape of the node is usually violating it.

The work of Anslow [16] uses basic shapes combined with colors to represent individual nodes as shown in Figure 2.8.

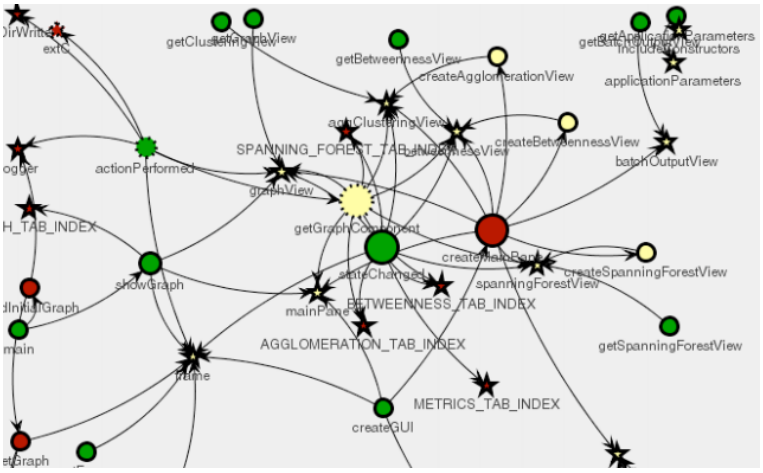

Figure 2.8: The ExtC Graph View using various node representations [16]

The work of Sazzadul [4] shows the application in 3D as a city, where are used various glyph for node representaion as shown in Figure 2.9.

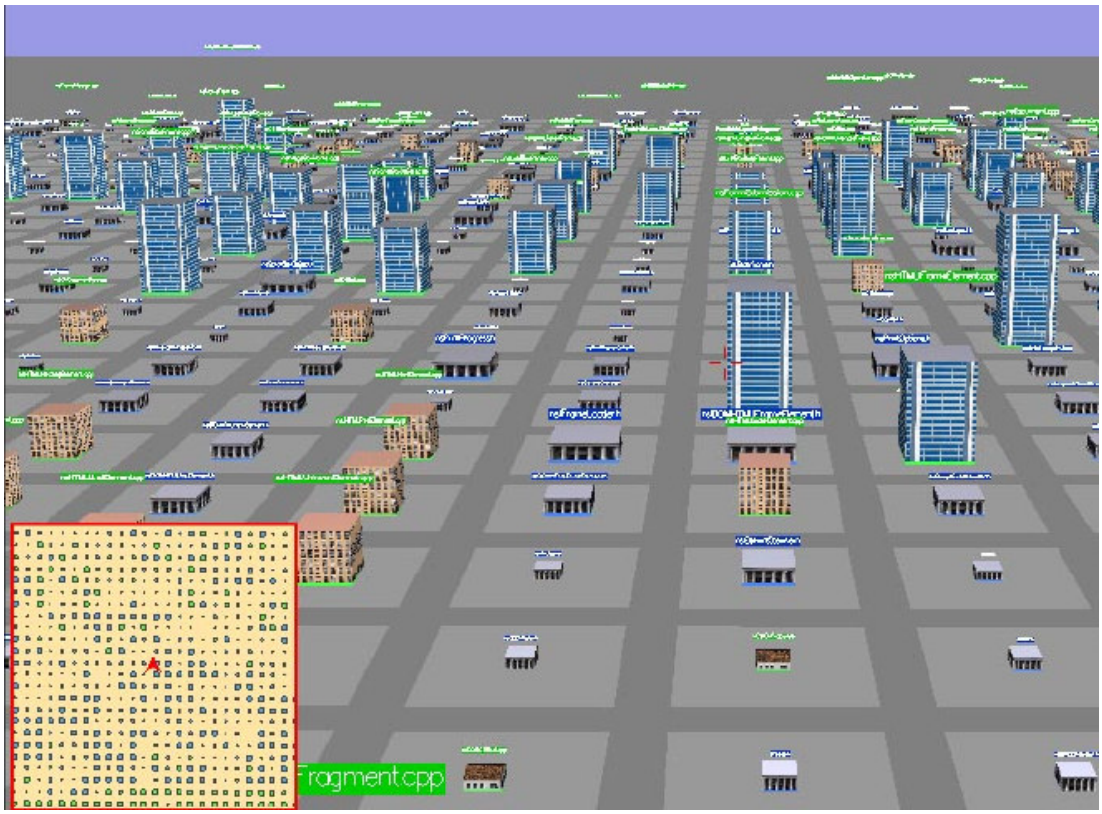

Figure 2.9: The file city - various glyphs for node representations [4]

2.2.4 Edges Visualization

One of the basic elements of diagrams are the links among nodes. Holten [38] came with the alternative representation of edges, which should help to reduce the visual clutter and can thus help users to orient easier. They developed five representations which combine the shape of the edge as well as the color.

To evaluate the proposed representations, they performed a user study which leads to following recommendations:

- Standard arrow representation (part (a) in Figure 2.10) should be avoided, because the performance of the users is quite low while using it. It is probably caused by the arrowheads, which cause occlusion problems and visual clutter.
- The best results was measured while using the tapered representation (f) in Figure 2.10 for directed graphs.
- For intensity based representation the dark-to-light representation is better than light-to-dark.
- Combining used factors (such as curving, changing colors etc.) for representation of the edges (multi-cue) does not seem to be better than using only one factor (single-cue).

The UML component diagram describes provided and required interfaces, which can be considered as directed edges. Thus above mentioned recommendadions can be used in component diagrams. Also the described intensity of edges can be used for indicating desired component connections metrics. Such as in case of clustered interfaces described in Section 3.4.3.

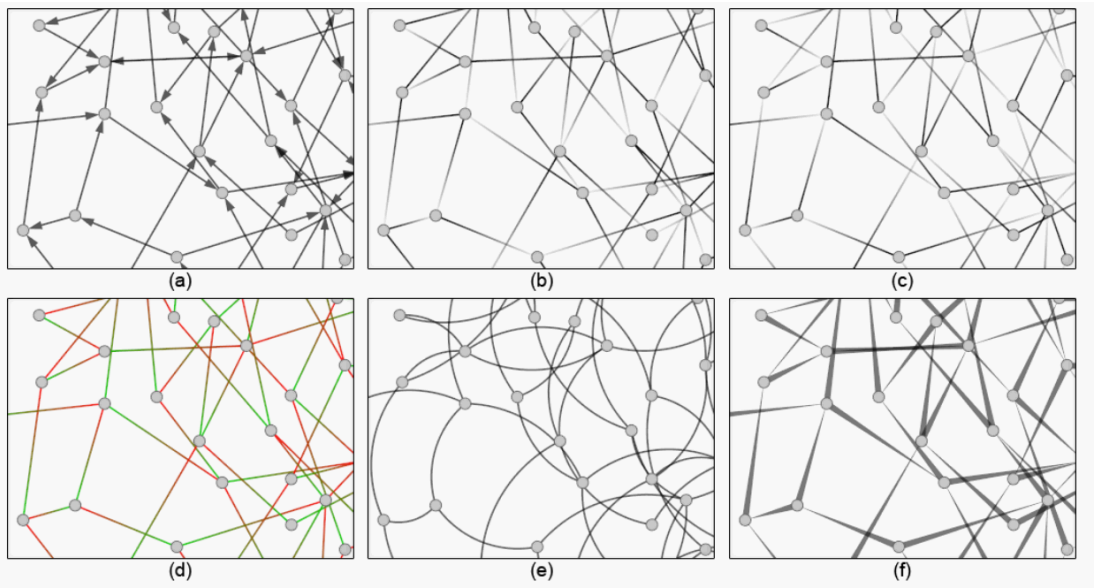

Figure 2.10: The six single-cue directed-edge representations used in the rst user experiment. (a) arrow; (b) light-to-dark; (c) dark-to-light; (d) green-to-red; (e) curved; (f) tapered [38]

Edge Bundling

Visualization of large node-link graphs usually suffer from visual clutter. One of the possible soultions of this problem can be using of edge bundling techniques which can reveal high-level edge patterns. The edge bundling can be applied for both general layouts of graphs or circle layouts. Holten [37] presented self-organizing approach to edge bundling. They model edges as flexible springs atracting each other. They also present rendering techniques to emphasize the bundling.

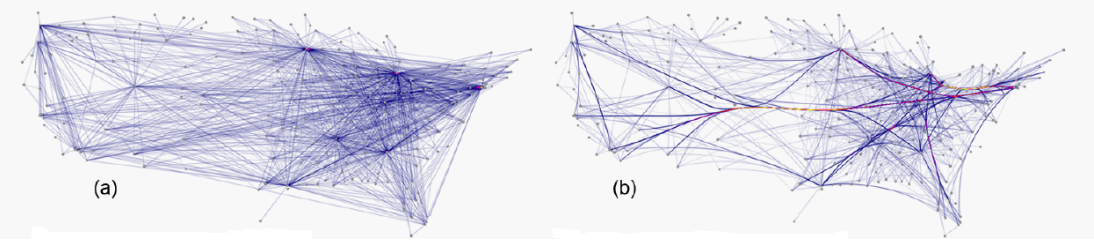

Figure 2.11: US airlines graph (235 nodes, 2101 edges) (a) not bundled graph (b) bundled graph [37]

Holten [36] also presents a technique of visualizing the elements in circle layout with the possibility to collapse elements. This collapsing and uncollapsing is fully animated, few steps are shown in Figure 2.12. Collapsing leads to replacing of all edges leading from all collapsed elements with one edge.

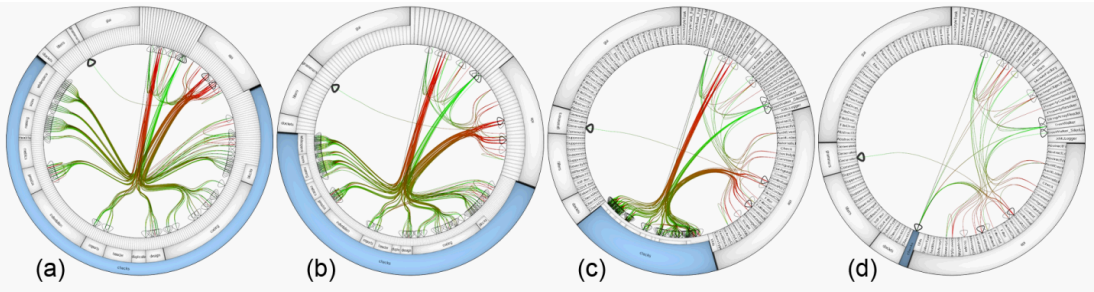

Figure 2.12: Steps of animation of collapsing the checks element (highlighted in blue) in (a) hides all of its children and lifts the relations pertaining to the children to the checks element, as shown in (d). [36]

Also the work of Gansner [30] presented a multilevel agglomerative edge bundling method. It minimizes ink needed to edges representation with respecting constraints on the curvature of the resulting splines. They declare that this method is able to bundle hundreds of thousands of edges in seconds. For comparison they provide the same graph as Holten [37], shown in Figure 2.13.

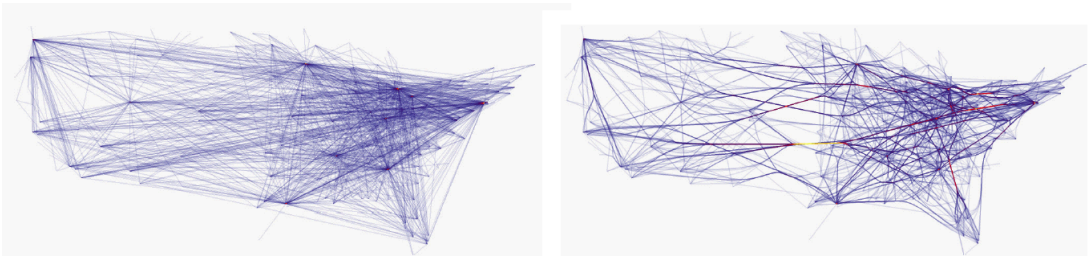

Figure 2.13: US airlines graph (a) not bundled graph (b) bundled graph [30]

2.2.5 Background Visualization

The background of visualized node-link diagram can be used for improving the navigation and understanding of visualized system. The work of Byelas [15] presented the tool using the areas of interest (AOI) technique in software diagrams. It investigates correlation of system properties while preserving the layout of dislpayed nodes. Several rendering modes of this technique are show in Figure 2.14.

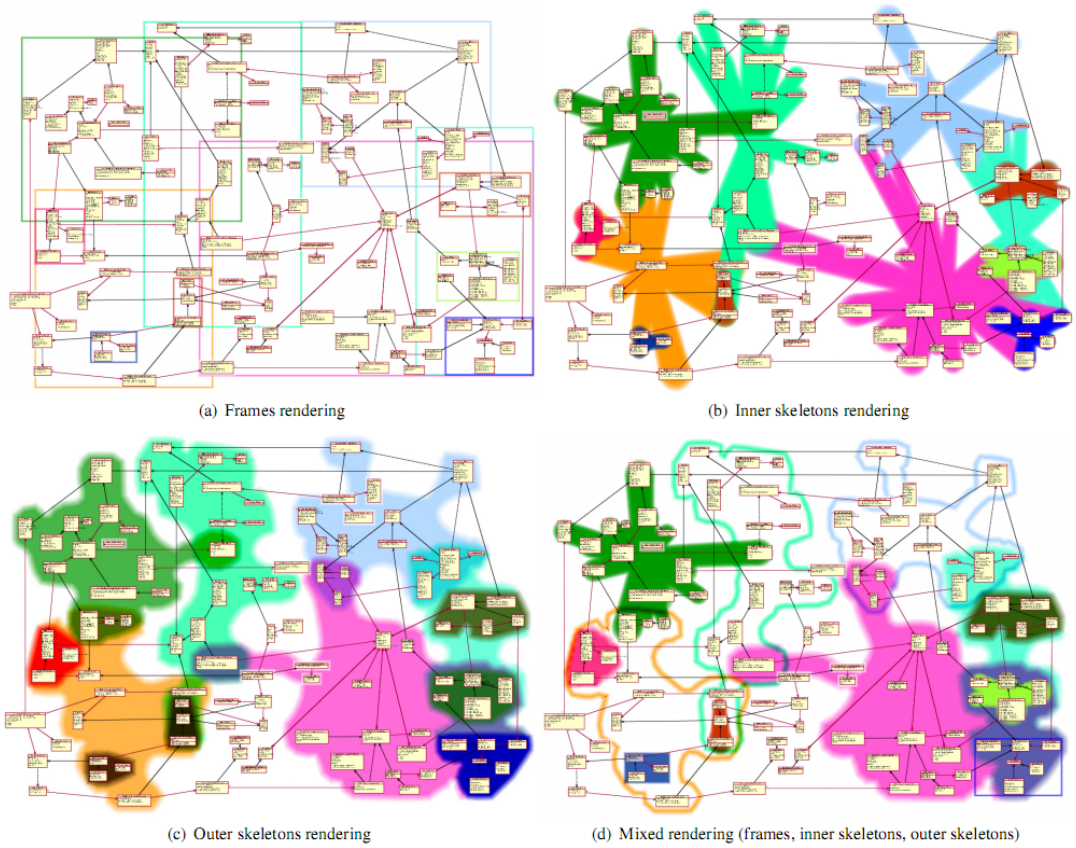

Figure 2.14: UML diagram with 12 AOIs, various rendering modes.[15]

Another work using diagram background is [29]. It describes the use of geographic maps to highlight clusters and neighborhoods. Although the work shows the similarities and recommendations arising from TV shows the idea could be adapted for the software visualization.

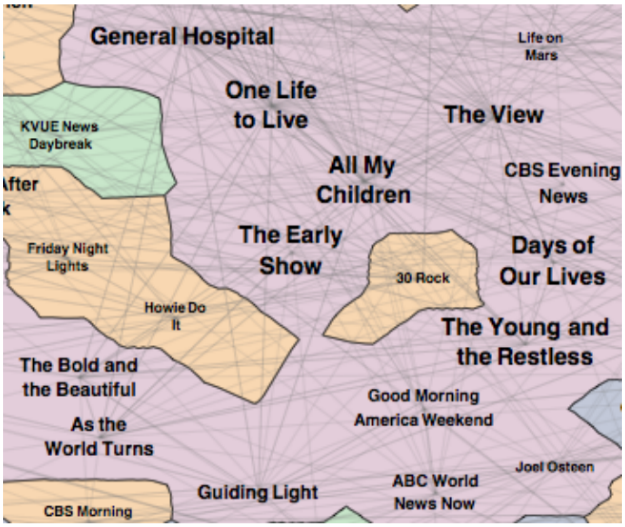

Figure 2.15: Background maps used for displaying clusers [29]

We provide deeper evaluation on current visualization tools in Section 3.2.

2.3 Nodes Clustering

In scope of this work the graph clustering can be used for reducing the amount of components in the displayed diagram. Although components usually represent relatively independent feature of a system, we can still find groups of components which represent even more global feature. For example a system can have several components for ensuring the security. So it is usually possible to find a group of components in a system which can be considered as a cluster. Cluster can be possibly collapsed into one node. Thus the number of nodes in whole diagram would be lowered and the understanding of the whole diagram become easier. The number of nodes in the whole diagram is lowered although the connections among components are usually still present and could be shown on demand.

Clusters can either be marked manually, in an automated way [19],[47], [10] or by a combination of those approaches [48]. The overview of clustering algorithms can found in [61], [71]. While using manual clustering user selects the nodes belonging to a cluster. When using an automated way, a diagram is considered as a graph. In this situation, the main factor for creating clusters usually are the interconnections (edges) among nodes. To improve the automated clustering

we can also use available metrics and information about components. These information can be eg. names of the packages (eg. org.package1.subpackage2) which can can help to form cluster even when the components are not even connected.

Important factor in choosing a clustering algorithm for certain implementation is the quality of the clustering result for the given domain. There are several metrics which can help to choose the appropriate clustering algorithm stated eg. in [10]. But even after using these metrics it is usually not clear which algoritms will give the best result for the general type of graph which component software application diagram can be.

Chapter 3

Concept of the thesis

The main apporach of this thesis lies in both using known techniques and using novel invented techniques for reducing complexity of large diagrams, which are generally node link graphs. This allows a user to explore the diagram and find desired information, design patterns or understand the architecture of a system. As useful existing techniques and approaches are considered ideas described in 2. Also using the hardware support for improving the insight into the data shown is described in 3.1. There are several relatively independent factors when visualizing complex software structures. By imporoving each of this factors and combining them together we achieve large increase of the insight into visualized system. The overall picture of the influencing factors important for the scope of this work are shown in Figure 3.1.

The screen size and resolution are described in 3.1. The factors from the software side of the Figure 3.1 are described in Section 2.2.

3.1 Using Large Projection Areas

The main idea of improving the diagram understanding by large projection areas is the fact that a person is generally able to see larger area than nowadays standard screen size. Thus the goal of this section is to provide an overview of current projection possibilites with respect to price of final solution which would increase the comfort of displayed diagram understanding.

Enlarging the projection area can be easily achieved by using projectors, but we also need to have high resolution to see the details. On the other hand a very high resolution on small projection area will not bring additional advantages. Thus one of the main requirements for the solution is to preserve reasonable ratio between the pixel size and projection area while covering whole user's persperctive.

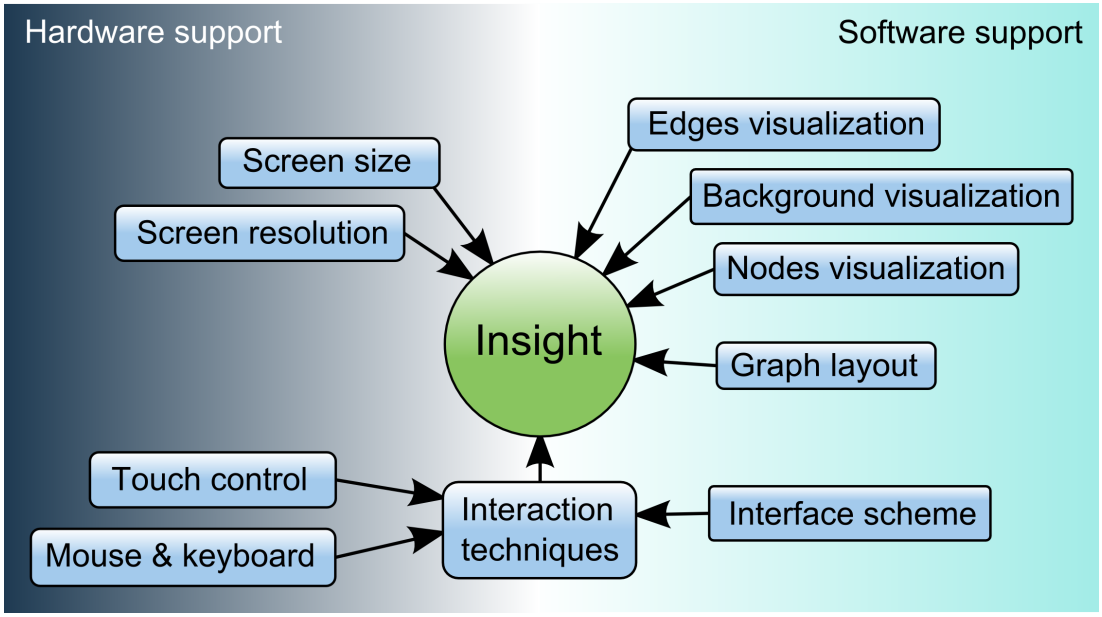

Figure 3.1: Factors influencing visualization in scope of this work

There are possibly four main ways of achieving the large viewing area:

- using high resolution projectors
- using several projectors composition
- using several monitors compostion

These solutions vary in price and comfort. While using multiple screens or projectors there is a possibility to use multiple interconnected computers as signal source. In case we have these interconnected computers we can save costs for the graphic adapters neccessary while using one computer. On the other hand there are further complications while using multiple computers such as delays or data throughput.

Current high resolution projectors can offer resolution around ten millions pixels. In the compostion of projectors we can theoretically achieve very large resolution, but currently offered products can provide hundreds of millions pixels. Using projectors is more expensive than using monitors, but it is possible to achieve a projection areas without seeing any visible grid. Current projecting devices for affordable price still do not exceed the abilities of humans eyes in a resolution criteria. The eye cannot be simply compared to projection devices by using only resolution metric, because there are many influencing factors in human's reception. On the other hand we can consider values between hunderds of millions and thousands of millions of pixels as comparable with humans abilities.

3.2 Component Software Visualization Tools Evaluation

In the following section, we describe the problems in visualization of component-based software related to its diversity, as well as different approaches to visualization of such structures. In Section 3.2.2 we suggest the criteria that can be used for evaluating tools able to visualize such software. These criteria are thoroughly discussed and evaluated from the views of different CBSE stakeholders. The application of these criteria is then presented on the example of IBM Rational Software Architect in Section 3.2.5.

3.2.1 User's Needs and Requirements

People involved in the component development and maintenance process need to visualize the component applications in a various ways. Visualization should help them to understand the system, analyze dependencies [46], extract and show desired properties, etc. These techniques are necessary especially when dealing with larger systems which consist from many (hundreds or thousands) components.

Graphical notation is one of the important aspects of visualizing component models. Many component models propose their own graphical notation while other ones assume a generic one like UML; this fragmented landscape can be seen as similar with the situation before UML became widely established for object-oriented languages.

Component Visualization Approaches

Components are by their nature more complex than classes in terms of their contractually specified interface features. Their models, visual syntax, supporting meta-data and tool functionalities should therefore be also more sophisticated. For example, the study [46] shows that architectural modeling would benefit from consolidated views, model consistency and defect checking, and its augmenting by metrics. Additionally, Kollman et al note that obtaining more abstract representations and providing advanced (semantically rich) model features are important for analysts [44].

Several works describe general criteria on analytical visualization tools, e.g. [67] or [45]; both of these works attempt to structure the criteria into categories for better orientation. [59] have further identified common desirable features or open issues which can be improved by visualization techniques. Visual notations can be in general analyzed or compared from the semiotic point of view, like in [62] or in [51], to understand the suitability of chosen symbols and layouts.

However we are not aware of any other method that would help to evaluate component architecture visualization tools. Favre et al discussed several issues with visualization of component-based software in [25]. While Favre covered all areas of component visualization, namely component models, components and their assemblies, he addressed only global issues of such visualization and he did not identified specific visualization tasks, however he provided a solid background and motivation for future work.

The options in modeling and visualizing component software architectures specifically are, cf. [49]:

1. component model-specific tool/notation;
2. generic component-aware tool/notation;
3. UML with profiles;
4. plain UML.

Component model-specific visualization means a visual notation (symbols and their meaning) supported by tools which are able to visualize only one or very few specific models. The motivation for this approach is the diversity of features provided by individual component models. The downside is that the specifics of the given notation can make it difficult for experts from different domains to read and understand the models. Examples of this approach are SaveCCM [32] or Palladio [6] component models.

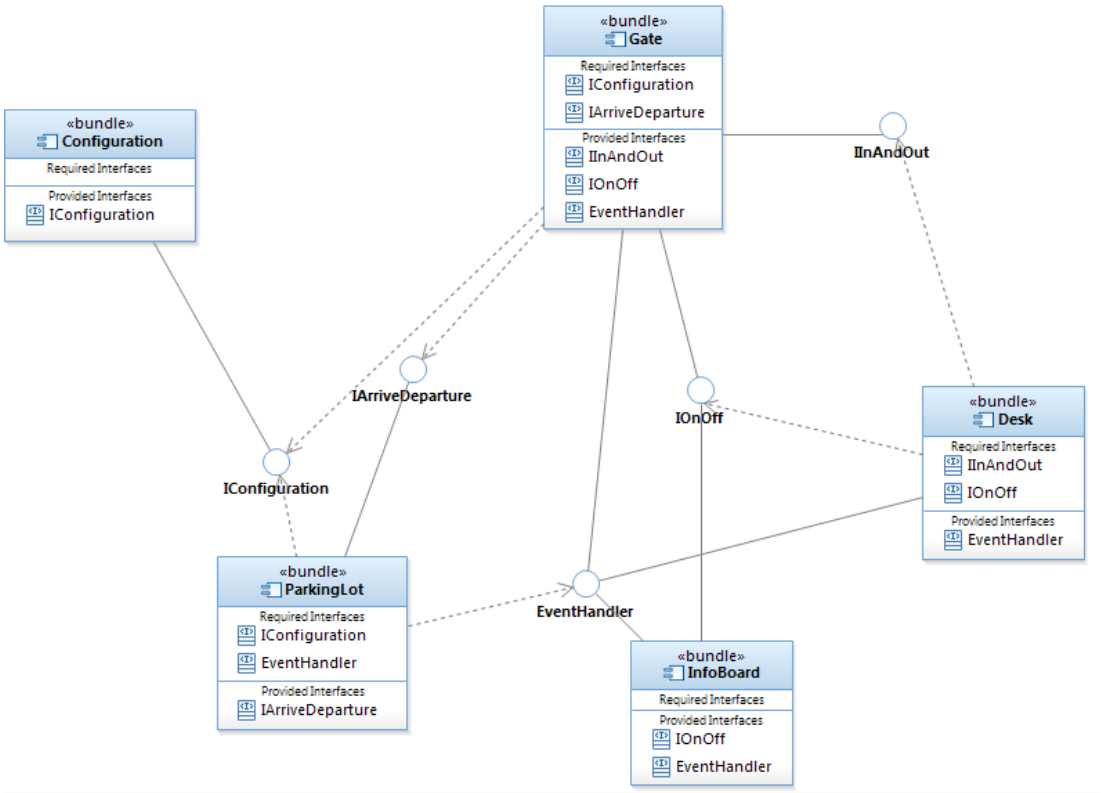

Figure 3.2: Example of plain UML2 Component Model

Secondly, we can use a universal component-aware visualization tool like SoftVision [66] which is either able to visualize any component model or can be extended for given component model needs. Related to this category is the use of UML [52] constrained by or extended with UML profiles which enable to further specify the semantics of existing model elements and create new ones on top of the core UML meta-model. Creation of profiles including introduction of icons for new model elements is supported by some tools, e.g. IBM Rational Software Architect, and many UML tools are able to use a pre-defined selection of profiles.

Finally, we can use plain UML, especially its component diagram (see Figure 2.7) and possibly class diagram. It may not capture the desired level of details necessary for full component modeling but provides a universal notation that is understood by most software engineers today. Moreover, the tool support is extensive (e.g. MagicDraw, Enterprise Architect, PowerDesigner or StarUML, to name just a few). However, this probably most common modeling approach "... lacks support for capturing and exploiting certain architectural concerns whose importance has been demonstrated through the research and practice of software architectures" [49] and supports only rudimentary analytical tasks.

Problems and Approaches Classification

In general, the options and benefits of a visualization of a component application are affected by: (a) the component model and its features; (b) visual notation's repertoire; (c) the capabilities of a tool used for visualization. Suitable visualization approaches have to be general enough to cover a wide range of component models while at the same time being able to capture all aspects of a concrete component model, in order to provide sufficient level of standardization while preserving precious information about the particular component-based applications. In visualization of component-based software it is therefore crucial to provide good notation and diagramming functionalities and beneficial to support more advanced features for architectural analyzes, data mining and visualization in general.

In this section we aim to define a suite of criteria that capture these features and emphasize the aspects important from CBSE point of view. These criteria should be suitable for the evaluation of visualization tools to indicate their fitness for advanced visualization of component-based software. Secondly these criteria can guide developers of current or new tools while considering implementation of new features, because each applied criterion increases the added value of the visualization tool.

#	*Functional criteria*	*Category*	*System architect (SA)*	*Compon. developer (CD)*	*Compon. assembler (CA)*
C00	Basic features	N/A	mandatory		
C01	Richness of component interface visualization	Data representation / Static	***	***	***
C02	Model extraction	Integration / Data mining	**	**	
C03	Component and architecture analysis	Data representation / Static		***	
C04	Finding matching variation / extension points	Data representation / Static	***	*	***
C05	Analysis and visualization of extra-functional properties	Data representation / Static	***	**	*
C06	Change analysis	Data representation / Dynamic and Evolution	***	**	*
C07	Analyzing differences between views	Operations / Comparison	**		**
C08	Traceability analysis	Operations / Searching	*	*	**
C09	Model querying and structural analysis	Operations / Searching		**	**
C10	Interactive components clustering	Operations / Searching	***		**
C11	Custom metrics and parameters visualization	Effectiveness / Benefits	**	*	**
C12	Diagram scalability and filtering	Effectiveness / Scalability	**		***

Table 3.1: Criteria and roles for component visualization

3.2.2 Criteria for Evaluating Tools

The criteria which we consider important for visualization tools targeted at component-based development are based on the general visualization rules and particular CBSE needs identified in the previous section. The criteria are summarized in Table 3.2.1; the list is structured using the general scheme proposed by [67] and related to roles specific to CBSE, cf. [65]. Individual criteria are

discussed in detail below.

The importance of each criterion for each role is indicated by stars, the scale is from none (not applicable) through one star for lowest importance to three stars for highest importance. Formula 3.1 describes the calculation of final rating s_r of given tool for one role.

$$s_r = \frac{\sum_{i=1}^{n}(w_i \cdot c_i)}{M \cdot \sum_{i=1}^{n}(w_i)} \tag{3.1}$$

Here w_i stands for the criterion importance and c_i represents the coverage of the feature by the given tool, on the scale from zero for "not present" to M for full coverage. Symbol n stands for the number of criteria and M equals three.

Criteria Description

We distinguish between basic and advanced criteria. As basic criteria we consider common tools features, which should be fulfilled in any case. As basic features we consider following:

- pan&zoom,
- diagram overview,
- adjusting the layout of a diagram,
- import&export,
- displaying model structure.

The brief description of advanced criteria follows.

Rich component interface visualization Represents the tool's ability to work with all properties and features specified by component model or framework.

Model extraction Describes the tool's ability to extract model from source code, deployment form or runtime representation, to a representation suitable for working with visualizing the gathered data.

Component and architecture analysis This criterion describes to what degree a tool is able to provide analyses of structures or behavior of components. There are many possible analyses, for instance for internal dependencies between provided and required interfaces or finding unused required interfaces or structures. Tools can also be able to check architecture style rules, detect design patterns or anti-patterns.

Finding matching variation/extension points The process of finding a variation or extension point in complex application can be very tedious. But if the tool is aware of the data types and structures it is displaying and is able to run basic queries internally, there is a possibility to offer users a feature which ease this process.

Analysis and visualization of extra-functional properties Extra-functional properties [41] can be either stored in a file or repository separately or can be gathered from the code or running system. Tools can also be able to compose the extra-functional properties of individual components into one property for the system or subsystem, and compare them in order to determine which component is better for a given purpose. There are also several ways of presenting the gathered data as a visualization in the diagram or them exporting into another tool.

Change analysis Represents the tool's ability to analyze the impact of the change (e.g. changed interfaces or relations), application's consistence and component's compatibility with other related components after the change.

Analyzing differences between views Although analyzing differences in textual data is a common task sufficiently solved by tools, differencing two graphical views is not a very common feature. It enables users to faster understand the changes made in the system.

Traceability analysis Important part of understanding the system is tracing through its dependencies. Although components should be treated as black boxes, composing the dependency along a chain of components from the individual internal dependencies between provided and required interfaces can be very useful. It enables users to predict the ripple effects of potential changes or understand the structure of the system.

Model querying and structural analysis Describes tool's ability to perform user specified or built-in operations generally needed to find desired information in the model. It comprises features from basic search to tool's own query language where the queries can be specified by user. Advanced features like structural analysis, model evolution prediction or design patterns and anti-patterns detection are also related to this criterion.

Interactive components clustering Diagrams of large applications become difficult to explore. One of the possible ways of improving the diagrams to be easier to understand is creating clusters of components which semantically represent a subsystem. Clusters can be minimized into symbols to lower the visual clutter of the application's diagram overview. These clusters can be found or suggested by tools automatically and/or adjusted by user manually.

Custom metrics and parameters visualization This criterion describes tool's ability to provide data and related operations, which would lead to vi-

sualization of desired metrics a parameters. Important part of this criterion is also the way in which the tool is able to visualize and customize the gathered data. There can be several data sources for the metrics and parameters. They can be stored in a file or repository separated from the diagram representation. Another way of gathering such data can be tool's own metrics measuring and composing capability.

Diagram scalability and filtering In case of large diagrams a tool should be able to handle the load and offer satisfactory response time. This criterion evaluates how the tool handles the problem of model complexity. It can be reduced for instance by multiple levels of displayed details or filtering highly connected parts suitable for detailed view.

In Table 3.2.1 we can see that most of the criteria are related with the component system architect or assembler and fewer are related with component developers. Component architects and assemblers need to have an overview of the whole system which can consist from hundreds or thousands of components and thus they need lot of analytical techniques and tools to ease their work.

3.2.3 Tools

This and following subsections describe the capabilities of current state-of-the-art tools for analyzing component applications in view of these criteria, in the form of a non-exhaustive survey. Primarily it describes the tools which provide interesting features besides basic component diagramming and focuses on those which introduce a novel look on component visualization.

In spite of the imperfections of plain UML component model we briefly present in subsection 3.2.4 selected tools which work with this model as the baseline. We also consider UML profiles as a separate point of view in subsection 3.2.5 because they provide an opportunity to represent various component models. Then we discuss visualization tools specific for some component models in subsection 3.2.6 which usually provide very good representation for the given model. Finally, in subsection 3.2.7 we sample tools which are able to represent any component model or at least support a high number of models or languages.

For each of the tools described in more detail we list the criteria from Table 3.2.1 it supports.

3.2.4 Plain UML Tools

The UML component diagram describes static application architecture and belongs to the structural diagrams category. It is able to show the components

themselves, their provided and required interfaces, associated artifacts and also composition hierarchy by putting (sub-)components inside other components.

There are many tools for drawing plain UML component diagrams, e.g. UMLet[1] or Dia[2].

MetricView

MetricView is a standalone tool which allows users to display custom metrics directly in the UML model (C11), as shown in Figure 3.3. Metrics visualization is among others useful for displaying extra-functional properties. This software has a version called MetricViewEvolution which is able to calculate metrics (C05), visualize evolution data (C06) and provide more views for UML model exploration. This tool also implements the area of interest technique [14] for UML diagrams (C12) which helps to highlight areas of concern in the diagram.

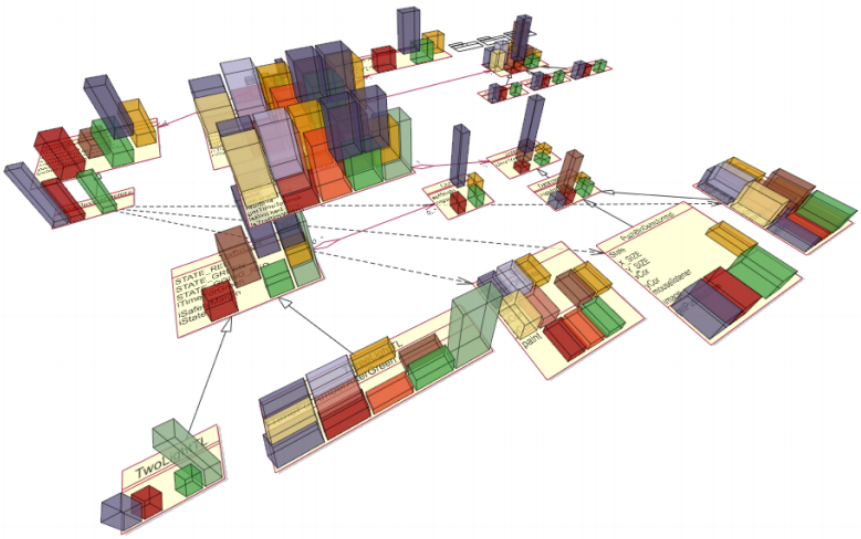

Figure 3.3: Metricview metrics visualization

3.2.5 Tools for UML Profiles

For purposes of component application modeling and visualization we can use UML profiles to describe the specifics of component model(s). Visualization and analytical features then depend on the tool's support for profiles.

[1] http://www.umlet.com/
[2] http://live.gnome.org/Dia

There are many tools which are able to work with UML profiles such as MagicDraw[3], StarUML[4], Borland Together Designer[5], Visual Paradigm for UML[6] or IBM Rational Software Architect (described in 3.2.5). Diagrams can usually be exchanged among such tools using XML Metadata Interchange (XMI), which should enable UML compliant documents exchange between tools.

Papyrus

Papyrus[7] is an component of the Eclipse Model Development Tools. It is able to work with UML2 exactly according to its definition and supports UML profiles very well. It it is able to customize its editors, model explorer and create user defined perspectives (C11, C12) in a way which provides users the look and feel comparable with domain specific language editors. Papyrus can download the following profiles UML profiles via its update site: MARTE, SysML, EAST-ADL, CCM and LwCCM (C05). The learning curve of this tool can be improved by using tutorials, videos or documentation provided.

IBM Rational Software Architect

IBM Rational Software Architect (RSA) is part of the Rational Rose tool family and it is build on the Eclipse platform. We chose RSA for this case study because it is not just a UML diagramming tool but rather represents a robust solution that supports model driven development, analytical work over different views on the same software and a lot more. All of these features are built on top of the UML meta-model.

RSA offers not only use of UML profiles but it is also possible to design new ones with it. This means that any component model can be represented with details limited only by the UML meta-model itself.

RSA supports all basic features needed for reasonable visualization of component-base software (C00), thus it is possible to use it for these purposes. Richness of contractual levels (C01) is achieved by using UML profiles, extension mechanism which – together with the option to define custom element icons – is powerful enough to model and reasonably well visualize most of kinds of component interface features.

RSA is able to trace dependencies, inheritance or ancestors by using several different features, thus covering the (C08) criteria in its full content. RSA enables

[3] http://www.magicdraw.com/
[4] http://staruml.sourceforge.net/
[5] http://www.borland.com/us/products/together/index.aspx
[6] http://www.visual-paradigm.com/product/vpuml/
[7] http://www.eclipse.org/modeling/mdt/papyrus/

model management for parallel development and architectural re-factoring – split, combine, compare and merge models and model fragments, thus (C07) criteria is also fully covered.

For model analysis and model metrics there is a special plug-in, called *The Model Metric Analysis Plug-in* which covers the criteria of (C11). This plug-in enables to create Kiviat diagrams ("spider charts"), perform interactive analysis of model and asses the results. RSA is able to create data sets (queries) to extract a defined set of information from UML models. This feature is accessed by using RSA extended with BIRT project[8], which also enables to create reports and sub-reports, these features covers the criteria (C09).

It may seem that model extraction (C02) is supported, because RSA can reverse-engineer class diagrams from Java, C++ and .NET source code. However, this ability does not work on component-based software and component diagrams. No other criteria is fulfilled.

Evaluation of RSA

Detailed overall value of IBM Rational Software Architect's component visualization capabilities is calculated by using Formula 3.1 and is summarized in Table 3.2.

#	c_i	**SA**	**CD**	**CA**
C01	2	***	***	***
C07	3	**		**
C08	3	*	*	**
C09	2		**	**
C11	2	**	*	**
s_r	12	**0,26**	**0,29**	**0,41**

Table 3.2: Assessment of RSA using our criteria

We can conclude that RSA does not fully cover the desiderata of component application visualization but still offers features, from which component assemblers can benefit the most.

3.2.6 Specific Component Model Visualization Tools

From the tools available for the many existing component models, we selected two representatives with direct support for model visualization.

[8] www.eclipse.org/birt/phoenix/

Save-IDE

Save-IDE[9] is an Integrated Development Environment (IDE) which can be used for the development of component-based embedded systems in the SaveCCM component model. Among others it uses formal specification and analysis of behaviors for designing systems (C05). It also enables internal component analysis (C03). Components' visualization in IDE is shown in Figure 3.4.

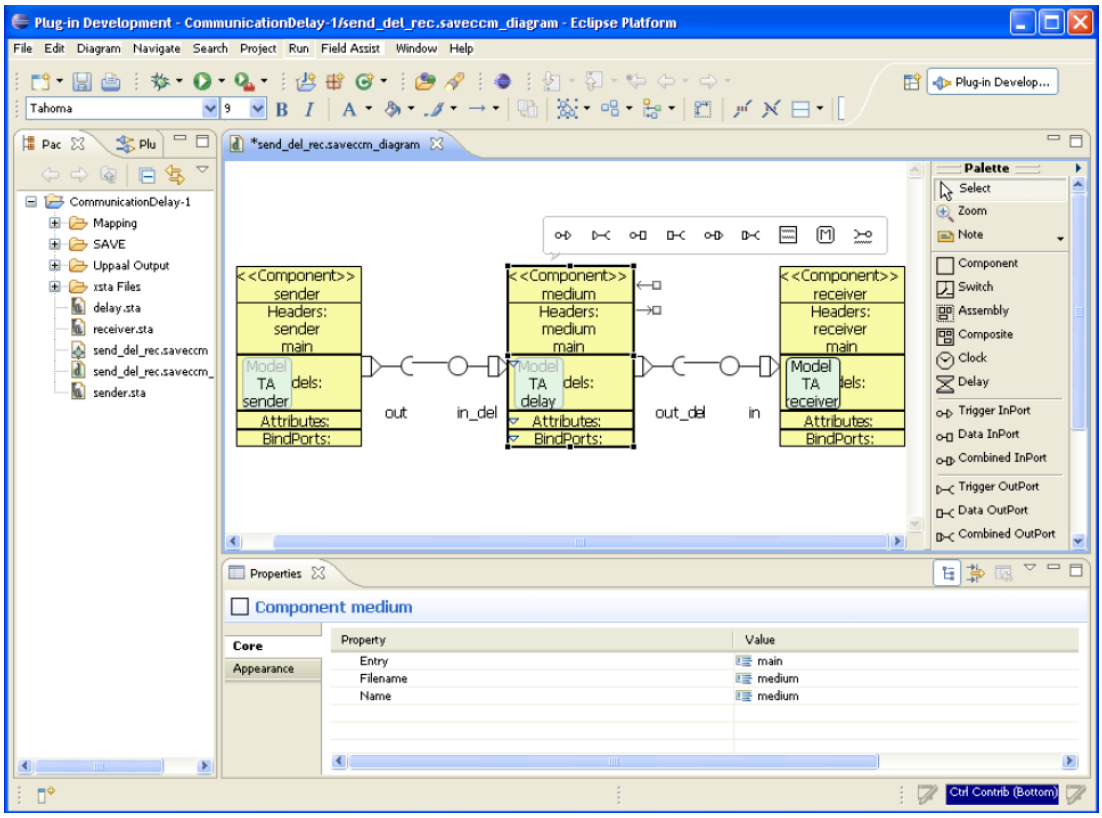

Figure 3.4: Save-IDE visualization

SOftware MOdel eXtractor (SoMoX)

SoMoX[10] is a tool for reverse engineering (C02) of the Palladio component model. Palladio can execute analysis (C05) of software performance, reliability, and maintenance properties on its component-based applications. It is also able to extract the components from source code (C10) written in various languages. Reverse engineering results in the creation of basic and composite components, component interface and service signatures, ports (roles), assembly and delegation connectors

[9] http://save-ide.sourceforge.net/
[10] http://www.palladio-simulator.com/tools/add_ons/somox/

and behaviour model. The extracted models enable quality analysis and help to understand analyzed system.

Plug-in Dependency Visualization

Plug-in Dependency Visualization[11] is a plugin for the Eclipse IDE. It enables to extract (C02), visualize and analyse the dependencies (C08) among core and user installed Eclipse plugins, called bundles. The dependency graph helps to understand the system by providing reasonable cognitive support. The user is able to select several options of highligting the bundles. For example, it is possible to show the shortest dependency path between two selected components. Example of application visualized by this tool can be seen in Figure 3.7

3.2.7 Generic Component Model-aware Visualization Tools

There are very few tools in this category, and often there is little information available about them.

SoftVision

SoftVision is a software visualization framework described in [66] which is able to interactively explore relations between data structures, as shown in Figure 3.5. It can scale the model to visualize large complex datasets (C12).

This tool enables users to define the structure of the component model used in a given component based system and thus visualize any component model (C02). If the needs of users differ for each component model, SoftVision provides elements customizability (C11). Thanks to this feature the user is able to create applications which suits well for exploration of given architecture. It also enables to write a custom scenario model which helps users better analyze the system by creating custom map, edit and filter operations (C09).

[11] http://www.eclipse.org/pde/incubator/dependency-visualization/

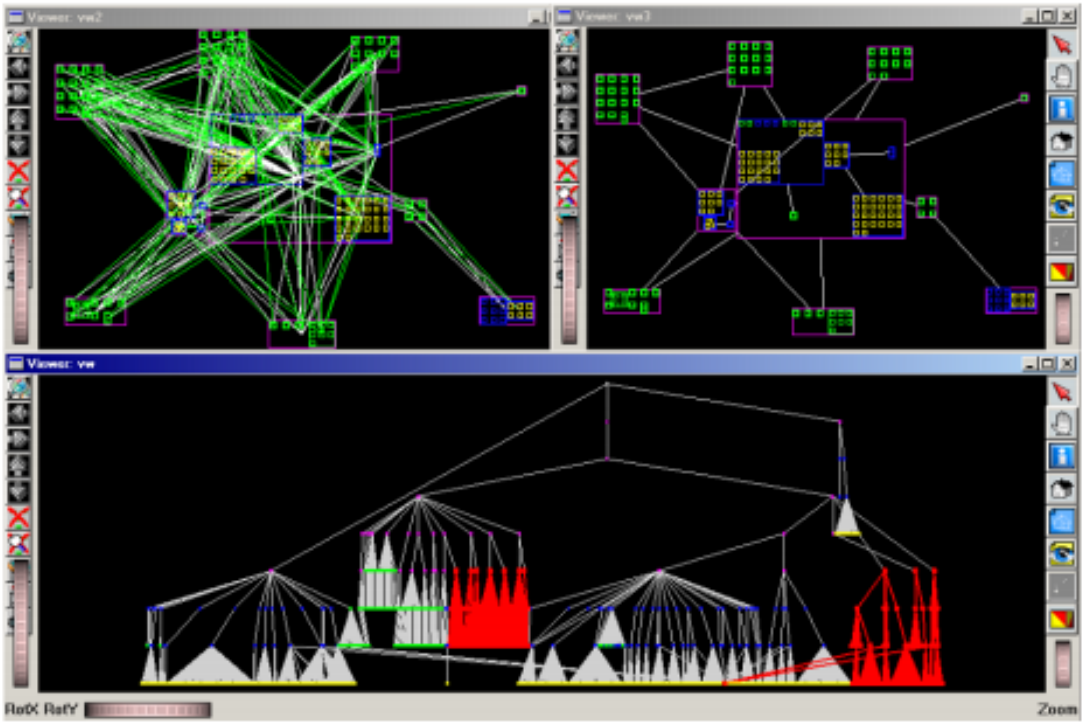

Figure 3.5: Softvision visualization [66]

Sec.	Approach	Visual syntax	Tool support	Compon. features fully captured	Simplicity of preparations before use	Requir. coverage
3.2.4	Plain UML component diagram	** well-known	*****	*	*****	**
3.2.5	UML profiles	**** tool dependent	***	****	**	**
3.2.6	Specific component model	model dependent	model dependent	***** model dependent	*****	*
3.2.7	Generic	custom	*	** tool dependent	**	***

Table 3.3: Comparison of approaches to component modeling

3.3 Viewport Technique for Surroundings Exploration

Visualization techniques which handle the complexity, such as off-screen rendering [27], can help to understand a diagram, even it is complex. This section describes a novel approach called viewport which attempts to reconcile the above mentioned contradictory requirements and helps to explore the dependencies among components in an intuitive way. This technique should help to work with complex diagrams (hundreds or thousands of components) by highlighting details of the important parts of the diagram and their related surroundings without losing the global perspective. To avoid visual clutter it uses clusters of interfaces and components.

3.3.1 Viewport for Component Diagrams

The proposed technique shows the graph (standard UML component diagram) zoomed-out to provide the appropriate overview of the complete architecture, with elements displayed without details. Besides that it shows selected components in detail inside a *viewport area* plus all their relations with other components in the diagram in an interactive border area (see Figure 3.6). These relations are for each component clustered into two sets: all provided interfaces (displayed as "lollipops") and all required interfaces (displayed as "sockets").

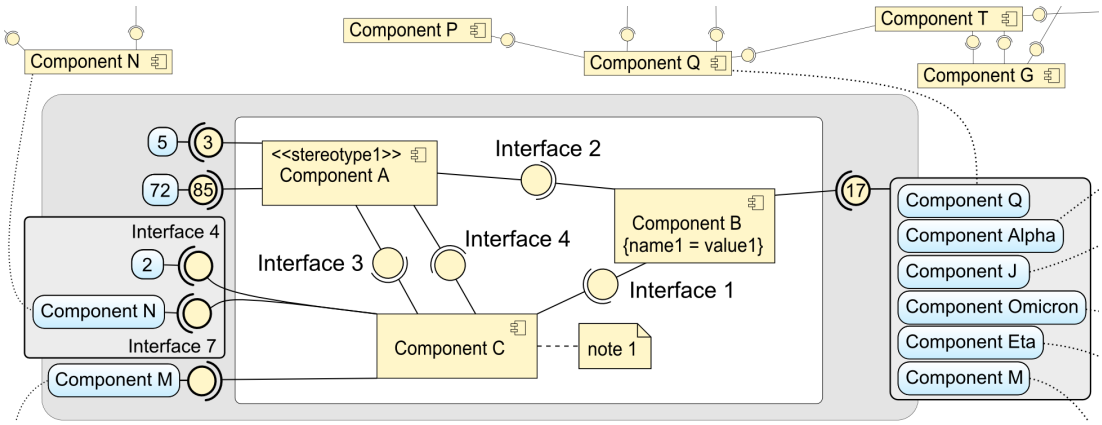

Figure 3.6: Viewport for component diagrams

These interfaces are then connected to clustered proxy components, visually represented as rectangles with rounded corners. Each rectangle represents one or more components. Numbers inside the clustered interfaces and proxy components represent a desired metric, e.g. the number of elements clustered in a given

symbol. One of the key factors of our approach will be the interactivity of the border area, which should comprise user manipulation with clustering of interfaces or components, layout adjustments and selecting the components shown in the viewport.

The viewport technique should enable to explore and understand the dependencies in large diagrams by showing the context of a selected diagram subset. The clustering shall reduce the visual clutter otherwise caused by large number of relations. The proxy elements should reduce the need for the disorienting pan&zoom otherwise necessary while exploring dependencies and provide user relevant information in one place. The viewport can either be placed on a given position in the diagram (there can be more viewports in a diagram) or have a fixed position on the screen.

3.4 Lowering Visual Clutter

In the following section we describe the problem of the visual clutter first. After that we describe a related work in Section 3.4.1. Then in Section 3.4.2 we present a novel technique which helps to reduce the visual clutter in large graphs. In Chapter 4, we describe the work in progress related to the implementation of the proposed technique as well as the future work.

This section focuses on the problem with highly connected components and the clutter caused by their connection visualization.

Very often, only a small amount of components is connected to a large number of other components. It results in a lot of lines going only from few components as can be seen in Figure 3.7, where is shown part of Eclipse[12] structure in Plugin Dependency Visualization tool[13]. Such components are often, among developers, informally called "God Objects". Having such objects (components), the user is limited to recognize other connections in their surrounding area and trace the connections themselves. Another side effect of these components is that they fill a lot of space, thus exhausting one of the essential resources in the visualization which can be used for easing the work with large component diagrams.

3.4.1 Coexisting Approaches

Visual clutter can be reduced by many techniques, such as bundling [37], sampling [58], clustering [18] etc. The whole taxonomy of these techniques has been

[12]Popular IDE, see http://www.eclipse.org/

[13]http://www.eclipse.org/pde/incubator/dependency-visualization/

described by Ellis and Dix in [24]. We provide a short description of those techniques which are related to our work.

The clutter caused by the lines is often reduced by edge bundling [35] (see also Section 2.2.4). Although this approach reduces the clutter, it can be difficult to trace the dependencies between connected nodes leading through the edge bundles. The visual clutter can be also lowered by using node clustering as mentioned in Section 2.3. Another influencing factor is the chosen layout algorithm (as described in Section 2.2.2), which can ease orientation in both clustered graphs [26] or a non-clustered ones [57], [31]. In the following section, we describe our approach to how to reduce the clutter in this problem.

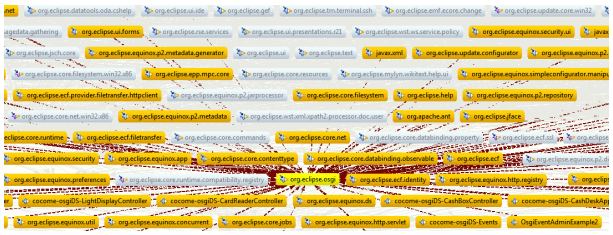

Figure 3.7: Wide Amount of Lines From One Component

3.4.2 Proposed Technique

The proposed technique reduces the visual clutter by removing the components with a large number of connections from the main diagram into a, so called, *separated components area* (abbreviated to SeCo) placed on the border of a window. This essentially marks the component as a "familiar one". The user may then concentrate on and continue getting familiar with the rest of the system.

When a user moves components from the main diagram to this area, the lines between these components and remaining components are elided. Instead of them a representing visual symbol is used in the diagram area. It reduces the number of lines in the graph not reducing the information provided. Obviously, components with a high number of connections are the most beneficial to be moved, because they reduce the high number of lines from the graph. For instance, a user may displace a component implementing a logger. Such a component is probably used by most of components in the system and its displacement reduces the graph complexity. We assume both automatic and manual component selection may be used. In the automatic case, all components with the number of connections overcoming a certain threshold are displaced. In the manual use, a user drags-and-drops the components from the main graph to the SeCo.

In the following paragraphs, we describe in detail the individual parts of the whole visual design used by this technique.

3.4.3 Separated Components Area (SeCo)

SeCo is a part of the application window. It can be placed on left or right side of the window, because current screens have wide aspect ratio and thus using these sides will not deform the rest of the viewing area as much as using the top or bottom side. The wireframe of the application window is shown in Figure 3.8.

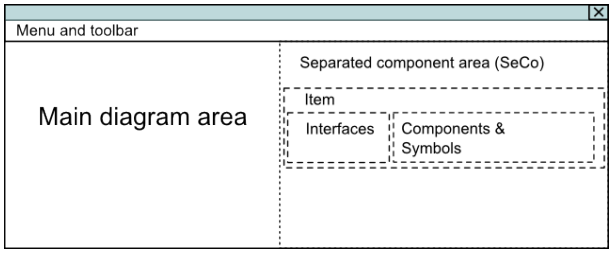

Figure 3.8: Overall Layout of the Application Window

Items

SeCo consists of a list of items. Each item consists of components, interfaces and one corresponding symbol (see Section 3.4.3). Components placed in SeCo have displayed relations with the rest of the components in the diagram on the border between diagram area and SeCo.

We distinguish between two situations corresponding to an item's internal layout of components and the representing symbol. In the first situation, if there is only one component in the item, interfaces are directly connected to the component and the symbol is behind the component as shown in Figure 3.10. In the second situation, the item consists of more components which form a group. In this case, the interfaces are directly connected to the symbol and the components are shown behind the symbol (Figure 3.12). The former situation stresses the display of the interfaces-component connections while the latter situation stresses the space saving. Groups are described more in detail in Section 3.4.3.

Symbols and Delegates

The purpose of symbols is to create clear and easily recognizable key which uniquely identifies one item within SeCo. Symbol should be small enough to save space anywhere it is used. The user should be able to choose its own symbols. We have chosen letters for the demonstration of the idea, but it can be any other symbol or an icon.

To keep the information about the connections in the main area when lines are removed, we use so called delegates. They represent the connection between a

given component and the corresponding item placed in SeCo. In the diagram, they are shown as small rectangles neighbouring the displayed components and containing the symbol which corresponds to the connected item (see Figure 3.9).

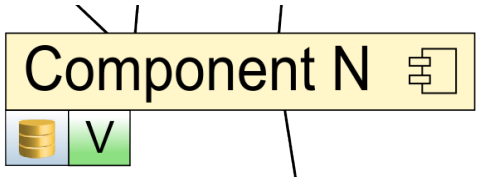

Figure 3.9: Delegates in the Diagram Area

By clicking on a delegate, the connections, interfaces and components involved in the relations are shown and/or highlighted. Display of the delegates in the diagram area can be toggled by clicking on the symbols in SeCo. The item indicates the state when delegates are shown by dark background as shown in Figure 3.10. The indication of the state when delegates are shown can be also differentiated by a checkbox, or other graphical element. We have chosen different background color in order to save screen space.

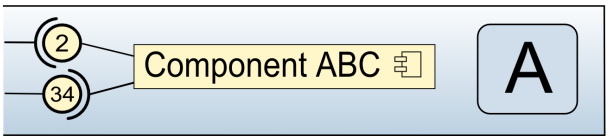

Figure 3.10: Item Design When Showing its Delegates

Interface Clustering

For each component shown in SeCo, interfaces are clustered into two sets: all provided interfaces (displayed as "lollipops") and all required interfaces (displayed as "sockets"). This is shown in Figure 3.10. Numbers inside the clustered interfaces represent the number of elements clustered in the given symbol. It helps to minimize the space which these components fill.

The clustered interfaces are by default not connected to the rest of the diagram by any lines which reduces the amount of lines in the diagram area. The connections (resp. lines) appear only when interacting with one of the sides of the connection included in clustered interfaces or the interface itself.

There are two kinds of interaction with clustered interfaces. First is a simple showing of the connections lines and highlighting of the components involved after user hovers with mouse cursor on the clustered interface. Second is a showing of the details of all interfaces including names, connections and highlighting of the involved components. It is launched by mouse click on the clustered interface. It is shown in Figure 3.11 for Interface 4. In a case a component from the diagram

area connected to an inspected interface would not be visible in the current diagram area view, it does not make sense to show the connection line and thus a proxy component is shown instead. This situation is shown in Figure 3.11 by the rectangle with rounded corners – Component N. It is shown in Figure 3.11 for Interface 4.

In the case a component from the diagram area connected to an inspected interface would not be visible in the current diagram area view, it does not make sense to show the connection line and thus a proxy component is shown instead. This situation is shown in Figure 3.11 by the rectangle with rounded corners for Component N.

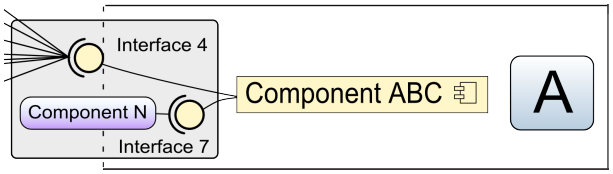

Figure 3.11: Interface Details

Component Groups

It is possible that a particular functionality of the system is implemented by several components. In a case this functionality is used by a large number of other components in the system, it can be represented as a group in SeCo.

All components from such a group are then replaced by one delegate in the diagram. It saves space in the diagram and also helps to create semantic clusters of components. It consequently improves understanding of the whole system where user may e.g. find cliques of components first. These may be then grouped and displaced from the graph to continue a study of the remaining graph.

The group symbols visually differ in component symbols and colours. A group symbol is larger in the size compared to the case of a single component, to denote the fact the group shows a large number of components. It is thus possible to show two additional categories of clustered interfaces. These categories contain all provided interfaces not used by any other component in a diagram and analogically all required interfaces which no other component provides. In the case of single components it is better to show these interfaces only on demand and thus save the space. The group is shown in Figure 3.12. Showing not used interfaces can easily inform the user about potentially missing components and thus prevent the future deployment problems.

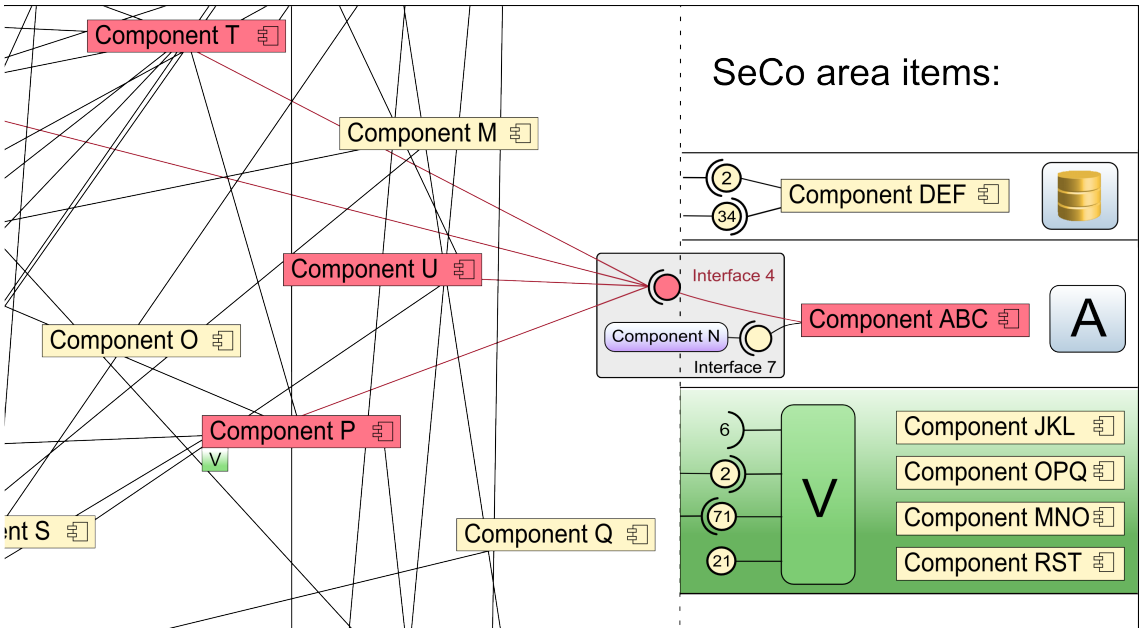

Figure 3.13: Application Layout with Example Diagram

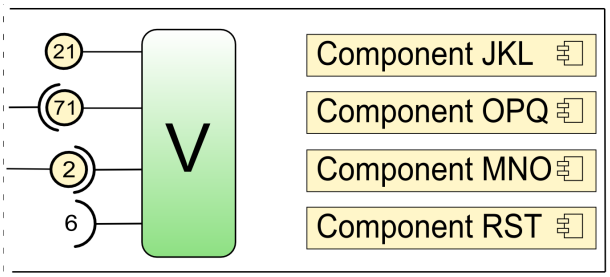

Figure 3.12: Group of Components Represented by a Group Symbol

When showing delegates in the diagram area for a given group or an item, its appeareance changes. We have chosen different the background color for demonstrating this item's state as shown in Figure 3.13 on group with *symbol* "V". This situation is equivalent to the situation of one component.

3.4.4 Discussion and Examples

In a lot of situations, there are components in the system which are connected with large number of other components. Thus, they are suitable candidates to be removed from the diagram area and moved to SeCo. In other cases we can use the SeCo features to form groups of components. These groups can serve as named categories according to which the user can classify the rest of the components in the diagram area and thus form a logical units of an investigated system.

Table 3.4 shows several systems with components having large number of connections. The table lists each system per a line with columns denoting the number of components, clustered and non-clustered connections among the components respectively. While non-clustered connections represent UML-like drawing separately connecting each individual provided-required interface pair, clustered connections collapse all connections between two components into two sets: all provided interfaces and all required interfaces.

System	Components	Clustered	Non Clustered
Nuxeo	202	698	1425
CoCoME	37	125	188
OpenWMS	65	232	642
Eclipse	378	533	1079

Table 3.4: Several Systems with the Number of Components and Connections

Several experiments using the proposed technique were performed, based on the data in the table. In one of them only 7 Nuxeo components have been removed from the diagram area leading to 241 and 431 lines remaining in the graph for the clustered and non-clustered versions, respectively. Therefore, the graphs were reduced of about 67% of lines in both cases.

These numbers show that by using the proposed technique, significant visual clutter reduction may be achieved. Visual effect of the results is shown in Figures 3.14 and 3.15, using circle layout for clarity.

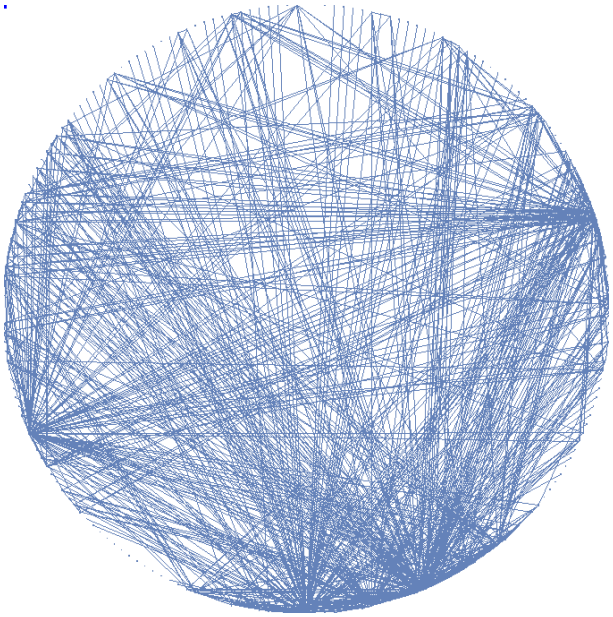

Figure 3.14: Nuxeo Before the Reduction

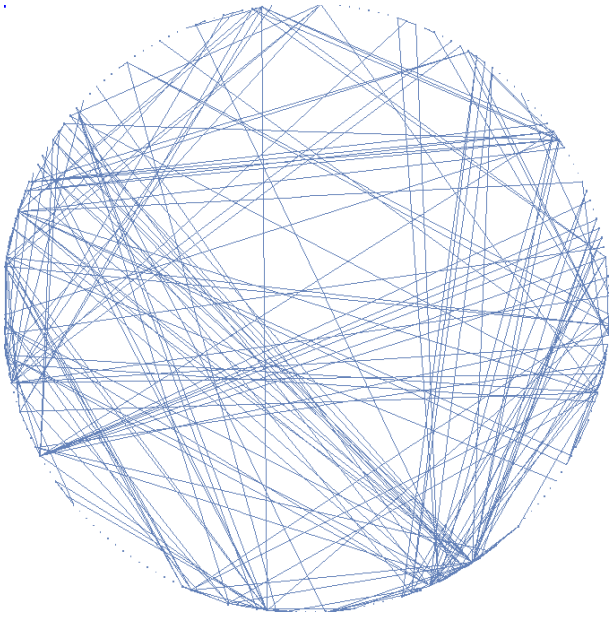

Figure 3.15: Nuxeo After the Reduction

3.4.5 Component Application Visualizer

We are currently implementing this technique in HTML5 technology as a plug-in extension to the ComAV tool [63]. ComAV is a versatile an extendible platform for visualization and reverse-engineering of component-based applications.

It offers the possibity to use multiple component models (currently OSGi, EJB 3 and SOFA 2 are supported) and different visualization styles. It uses component-model independent data format to store a reverse engineered structure of component-based applications and as an input for any visualization plug-in.

We plan to integrate a viewport technique [39] into this application to enable users to form relevant clusters comfortably. ComAV is thus a suitable tool to test this new technique as it has powerful reverse-engineering features supporting OSGi, EJB 3 and SOFA 2 component models. Consequently, it can easily analyse a structure of hundreds of components.

3.4.6 Techniques' Implementation

For implementing above decribed techniques a graph framework can be used, because the component diagram can be considered as a graph. There are many available graph frameworks both commercial or free to use. Their list for Java can be found at [3]. We focused on framework's ability to interact in a short time with the user while displaying large amount of elements. The other important

abilities were available documentation, graph layouts, customizability and size of the community around framework.

At first we chose JUNG, JGraph and Zest frameworks as suitable for visualization of large graphs for further comparison. Mentioned frameworks are able to work with both directed and undirected graphs. They also provide GUI for work with the displayed graph and layout functions. After testing these three frameworks we decided for JUNG. The framework and reasons of our decision are described in following section.

JUNG

The JUNG [2] stands for Java Universal Network/Graph Framework. It is a software library written in Java and compatible with Swing. It supports both directed and undirected graphs as well as hypergraphs and graphs with parallel edges. It allows users to annotate graphs, entities, and relations with metadata. It is also possible to use Java applets as shown in Figure 3.16.

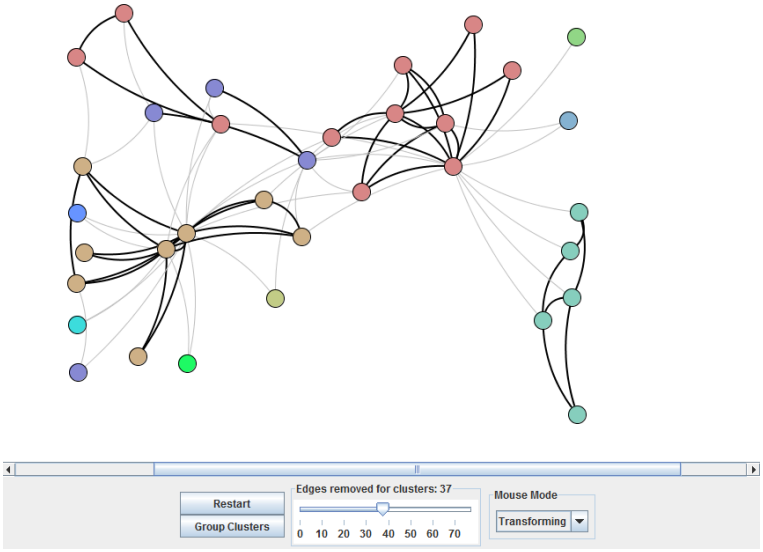

Figure 3.16: Example of JUNG applet showing both clustering and layout (Fruchterman-Reingold) algorithms.

We chose the JUNG framework as the most suitable for our needs, because of good documentation and overall functions. Showing 5000 nodes interconnected with 5000 edges lasts 28 seconds in JUNG framework whereas it takes 1180 seconds in JGraph framework.

We implemented a prototype showing the large component diagram. We have customized the connections lines to be shown as connected interfaces. Although this framework is very good in graph visualization, we considered the imple-

mentation of described techniques problematic, mainly due to large amount of specific customizations. Thus we decided not to use a framework and implement the desired techniques by using basic drawing primitives.

HTML5 + Java EE

We decided to implement the desired features as a web application to be easier to use. As backend technology we use servlets from Java EE technology, mainly because of Java implementation of ComAV tool [63]. For frontend we use HTML5, JavaScript, jQuery [14] framework and CSS3. These technologies are widely supported and provide desired features seamlessly integrated in the web page, whereas the JUNG uses applets which cannot be connected to the rest of the web page so easily as the previously mentioned technologies. We use canvas and SVG elements from HTML5 to represent the nodes of the diagram. Although HTML5 techonology is still not fully supported [15] by all main browsers, it provides uploading of multiple files, which is used for uploading components. Also desired features such as SVG support or Canvas are likely to be stable in the future.

The tool is able to load and visualize the components reverse-engineered by the ComAV tool. A user first pick component on a local machine a uploads them to the server. The ComAV tool creates the model of the application and the CoCA-Ex tool shows the application diagram in the webpage.

It currently has following features:

- removing nodes with the highest degree to the SeCo area (as single components or as one group),
- searching and highlighting components in the diagram according to given name,
- panning and zooming,
- manual layout adjusting,
- connected elements highlighting,
- delegates showing (as described in Section 3.4.3),
- symbols using,
- groups using.

[14] http://jquery.com/
[15] http://caniuse.com/

It provides standard features such as panning and zooming. There are two modes of manipulating the components with appropriate icons in the toolbar. First mode is for moving components (A) where the user can manually adjust the layout of the diagram. Second mode (B) serves for removing components from the diagram area to the SeCo area simply by clicking on the desired components which should be removed. Last two icons in the toolbar serve for the automatic removal of a configured amount of components from the diagram to the SeCo area. The tool is currently configured to remove 15% of most connected components. The icon (C) is used for removing these components and adding them to SeCo area as individual items. The next icon (D) creates one group for all of them.

CoCA-Ex offers a fulltext search in components' names. In Figure 3.17, one can see the search for a word "relations". Seven components in the diagram contain this word as indicated by the number seven (F). Matching components are highlighted by orange color (E).

If one clicks on the provided interfaces of a component in SeCo, these interfaces and connected components become highlighted by green color. An example is shown on dependency between the *Nuxeo Common* component's provided interfaces (Y) and *Nuxeo Platform Imaging API* component (G). Similarly, for interfaces required by components in SeCo highlighting by yellow color is used. It is demonstrated on dependency between *Nuxeo URL API* component (H) and *Nuxeo ECM Web Platform UI* component's required interfaces (Y).

For several components from the SeCo area (those with symbols' background highlighted by different colors (S)) there are delegates shown in the diagram area, e.g. (K). For inspecting interfaces, the tool offers highlighting of a connection by a red color and showing the interfaces involved in the connection (P), as shown in the green tooltip. Each individual component shown in SeCo has its own button (R) to remove it back to its original position in the diagram area.

Our preliminary experiences with this tool show that it is able to handle large diagrams without problems. Also we discovered requirements for this tool which should be implemented to ease the work to a user. We describe this requirements in Chapter 4.

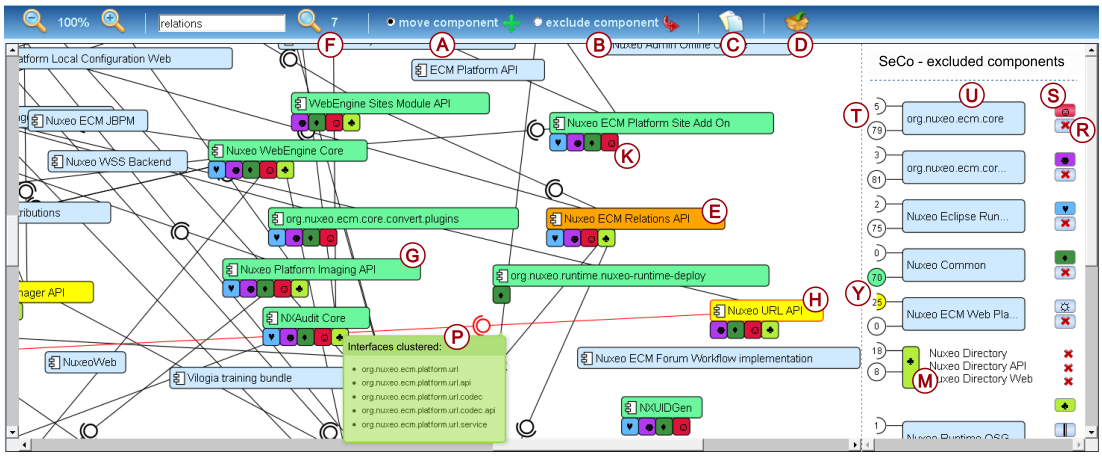

Figure 3.17: Nuxeo system loaded into CoCA-Ex application

Chapter 4

Future Work

Future work will focus on research on new techniques, their implementation and evaluation. It will also focus on the CoCA-Ex tool improvements in following features:

- layout integration,
- clustering integration,
- clustered interfaces exploration,
- viewport technique integration.

Furhermore we will investigate options for automatic suggestion of diagram parts suitable for displaying in viewports. Important part of the future reasearch will also be an evaluation of the CoCA-Ex tool.

We believe that the presented ideas can be generalized to be used in other domains, where one suffers from visual clutter caused by the large number of nodes and connection lines. Thus one part of the future work will be to provide examples of these applications including the technique adaptations.

Chapter 5

Conclusion

In this work we suggested several criteria for evaluating tools targeted at visualization of component-based software. These criteria can be used on existing visualization tools as we presented on the example of IBM Rational Software Architect, which was evaluated with quite satisfactory results. On the other hand, this case shows that even advanced visualization tools currently address only a few of the needs related to component visualization.

The proposed criteria can thus also serve as a guideline for efforts towards better visualization of component-based applications. Currently the main problem behind the lack of such efforts can be due to relatively low usage of components. However, their importance continues to rise and future visualization tools should address these topics to a broader extent.

We also presented a viewport technique which can help to form clusters and ease the process of exploring selected components surroundings.

We also described a technique which helps to reduce the amount of lines in the diagram, by removing the selected components from the diagram area (shown in Figure 3.13). It uses a *separated components area* where the selected components are shown, and symbolic delegates which represent the connections instead of lines.

These techniques are among other benefits useful in the reverse engineering process when the user is interactively getting familiar with the whole diagram.

Finally we provided an implementation of part of the invented techniques in CoCA-Ex tool which is based on HTML5, CSS3, JavaScript, jQuery and Java EE technologies.

Bibliography

[1] Handbook of graph drawing and visualization, 2012.

[2] Java universal network/graph framework, 2012.

[3] The stony brook algorithm repository, 2012.

[4] S. Alam and P. Dugerdil. EvoSpaces Visualization Tool: Exploring Software Architecture in 3D. In *WCRE '07: Proceedings of the 14th Working Conference on Reverse Engineering*, pages 269–270, Washington, DC, USA, 2007. IEEE Computer Society.

[5] F. Bachmann, L. Bass, C. Buhman, S. C. Dorda, F. Long, J. Robert, R. Seacord, and K. Wallnau. Volume ii: Technical concepts of component-based software engineering, 2nd edition. Technical report, CMU/SEI - Carnegie Mellon University/Software Engineering Institute, 2000.

[6] S. Becker, H. Koziolek, and R. Reussner. The palladio component model for model-driven performance prediction. *Journal of Systems and Software*, 82(1):3 – 22, 2009. Special Issue: Software Performance - Modeling and Analysis.

[7] B. B. Bederson and A. Boltman. Does animation help users build mental maps of spatial information? In *Proceedings of the 1999 IEEE Symposium on Information Visualization*, INFOVIS '99, pages 28–, Washington, DC, USA, 1999. IEEE Computer Society.

[8] A. Beugnard, J.-M. Jézéquel, N. Plouzeau, and D. Watkins. Making components contract aware. *Computer*, 32(7):38–45, 1999.

[9] A. Beugnard, J.-M. Jzquel, and N. Plouzeau. Contract aware components, 10 years after. In J. Cmara, C. Canal, and G. Salan, editors, *WCSI*, volume 37 of *EPTCS*, pages 1–11, 2010.

[10] R. A. Bittencourt and D. D. S. Guerrero. Comparison of graph clustering algorithms for recovering software architecture module views. In *Proceedings of the 2009 European Conference on Software Maintenance and Reengineering*,

CSMR '09, pages 251–254, Washington, DC, USA, 2009. IEEE Computer Society.

[11] P. Brada. The cosi component model: Reviving the black-box nature of components. In *Proceedings of the 11th International Symposium on Component-Based Software Engineering*, CBSE '08, pages 318–333, Berlin, Heidelberg, 2008. Springer-Verlag.

[12] F. J. Brandenburg, M. Himsolt, and C. Rohrer. An experimental comparison of force-directed and randomized graph drawing algorithms. pages 76–87. Springer-Verlag, 1996.

[13] T. Bures, P. Hnetynka, and F. Plasil. SOFA 2.0: Balancing advanced features in a hierarchical component model. In *SERA*, pages 40–48. IEEE Computer Society, 2006.

[14] H. Byelas, E. Bondarev, and A. Telea. Visualization of areas of interest in component-based system architectures. In *Proceedings of the 32nd EUROMICRO Conference on Software Engineering and Advanced Applications*, pages 160–169, Washington, DC, USA, 2006. IEEE Computer Society.

[15] H. Byelas and A. Telea. Visualization of areas of interest in software architecture diagrams. In *Proceedings of the 2006 ACM symposium on Software visualization*, SoftVis '06, pages 105–114, New York, NY, USA, 2006. ACM.

[16] K. Cassell, C. Anslow, L. Groves, P. Andreae, and S. Marshall. Visualizing the refactoring of classes via clustering. In M. Reynolds, editor, *Australasian Computer Science Conference (ACSC 2011)*, volume 113 of *CRPIT*, pages 63–72, Perth, Australia, 2011. ACS.

[17] C. Chen. Graph drawing algorithms. In *Information Visualization*, pages 65–87. Springer London, 2006. 10.1007/1-84628-579-8_3.

[18] K. Chen and L. Liu. A visual framework invites human into the clustering process. In *Scientific and Statistical Database Management, 2003. 15th International Conference on*, pages 97 – 106, july 2003.

[19] Y. Chiricota, F. Jourdan, and G. Melançon. Software components capture using graph clustering. In *Proceedings of the 11th IEEE International Workshop on Program Comprehension*, IWPC '03, pages 217–, Washington, DC, USA, 2003. IEEE Computer Society.

[20] A. Cockburn, A. Karlson, and B. B. Bederson. A review of overview+detail, zooming, and focus+context interfaces. *ACM Comput. Surv.*, 41(1):2:1–2:31, Jan. 2009.

[21] I. Crnkovic, M. Chaudron, S. Sentilles, and A. Vulgarakis. A classification framework for component models. In *Proceedings of the 7th Conference on Software Engineering and Practice in Sweden*, October 2007.

[22] I. Crnkovic;, S. Sentilles, A. Vulgarakis, and M. R. Chaudron. A classification framework for software component models. *IEEE Transactions on Software Engineering*, 37:593–615, 2011.

[23] T. Dwyer, B. Lee, D. Fisher, K. I. Quinn, P. Isenberg, G. Robertson, and C. North. A comparison of user-generated and automatic graph layouts. *IEEE Transactions on Visualization and Computer Graphics*, 15(6):961–968, Nov. 2009.

[24] G. Ellis and A. Dix. A taxonomy of clutter reduction for information visualisation. *Visualization and Computer Graphics, IEEE Transactions on*, 13(6):1216 –1223, nov.-dec. 2007.

[25] J.-M. Favre and H. Cervantes. Visualization of component-based software. In *Proceedings of the 1st International Workshop on Visualizing Software for Understanding and Analysis*, pages 51–, Washington, DC, USA, 2002. IEEE Computer Society.

[26] Q. Feng. Algorithms for drawing clustered graphs, 1997.

[27] M. Frisch and R. Dachselt. Off-screen visualization techniques for class diagrams. In *Proceedings of the 5th international symposium on Software visualization*, SOFTVIS '10, pages 163–172, New York, NY, USA, 2010. ACM.

[28] T. M. J. Fruchterman and E. M. Reingold. Graph drawing by force-directed placement. *Softw. Pract. Exper.*, 21(11):1129–1164, Nov. 1991.

[29] E. Gansner, Y. Hu, S. Kobourov, and C. Volinsky. Putting recommendations on the map: visualizing clusters and relations. In *Proceedings of the third ACM conference on Recommender systems*, RecSys '09, pages 345–348, New York, NY, USA, 2009. ACM.

[30] E. R. Gansner, Y. Hu, S. C. North, and C. E. Scheidegger. Multilevel agglomerative edge bundling for visualizing large graphs. In G. D. Battista, J.-D. Fekete, and H. Qu, editors, *PacificVis*, pages 187–194. IEEE, 2011.

[31] S. Hachul and M. Jnger. Large-graph layout algorithms at work: An experimental study. http://jgaa.info/ vol. 11, no. 2, pp. 345369, 2007.

[32] H. Hansson, M. Akerholm, I. Crnkovic, and M. Tarngren. SaveCCM - a component model for safety-critical real-time systems. In *EUROMICRO*, pages 627–635. IEEE Computer Society, 2004.

[33] D. Harel and Y. Koren. A fast multi-scale method for drawing large graphs (full version). In *Journal of Graph Algorithms and Applications*, pages 183–196. Springer-Verlag, 2000.

[34] M. Himsolt. Comparing and evaluating layout algorithms within graphed. *J. Visual Languages and Computing*, 6:255–273, 1995.

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

[36] D. Holten, B. Cornelissen, and J. J. van Wijk. Trace visualization using hierarchical edge bundles and massive sequence views. *Visualizing Software for Understanding and Analysis, International Workshop on*, 0:47–54, 2007.

[37] D. Holten and J. J. van Wijk. Force-directed edge bundling for graph visualization. *Comput. Graph. Forum*, 28(3):983–990, 2009.

[38] D. Holten and J. J. van Wijk. A user study on visualizing directed edges in graphs. In *Proceedings of the 27th international conference on Human factors in computing systems*, CHI '09, pages 2299–2308, New York, NY, USA, 2009. ACM.

[39] L. Holy and P. Brada. Viewport for component diagrams. In M. J. van Kreveld and B. Speckmann, editors, *Graph Drawing*, volume 7034 of *Lecture Notes in Computer Science*, pages 443–444. Springer, 2011.

[40] L. Holy, J. Snajberk, and P. Brada. Evaluating component architecture visualization tools - criteria and case study. 2012.

[41] International Standard Organization (ISO/IEC). Informational technology – product quality – part 1: Quality model. International Standard ISO/IEC 9126, June 2001.

[42] T. Kamada and S. Kawai. An algorithm for drawing general undirected graphs. *Inf. Process. Lett.*, 31(1):7–15, Apr. 1989.

[43] C. Klein and B. B. Bederson. Benefits of animated scrolling. In *CHI '05 extended abstracts on Human factors in computing systems*, CHI EA '05, pages 1965–1968, New York, NY, USA, 2005. ACM.

[44] R. Kollman, P. Selonen, E. Stroulia, T. Systä, and A. Zündorf. A study on the current state of the art in tool-supported uml-based static reverse engineering. In A. van Deursen and E. Burd, editors, *Proceedings of the 9th Working Conference on Reverse Engineering (WCRE 2002)*. IEEE Computer Society, November 2002.

[45] A. Kuhn, D. Erni, P. Loretan, and O. Nierstrasz. Software cartography: thematic software visualization with consistent layout. *J. Softw. Maint. Evol.*, 22:191–210, April 2010.

[46] C. F. Lange, M. R. Chaudron, and J. Muskens. In practice: UML software architecture and design description. *IEEE Software*, 23(2):40– 46, April 2006.

[47] S. Mancoridis, B. S. Mitchell, and C. Rorres. Using automatic clustering to produce high-level system organizations of source code. In *In Proc. 6th Intl. Workshop on Program Comprehension*, pages 45–53, 1998.

[48] F. McGee and J. Dingliana. Visualising small world graphs - agglomerative clustering of small world graphs around nodes of interest. 2012.

[49] N. Medvidovic, D. S. Rosenblum, D. F. Redmiles, and J. E. Robbins. Modeling software architectures in the unified modeling language. *ACM Trans. Softw. Eng. Methodol.*, 11(1):257, January 2002.

[50] P. Merle and J.-B. Stefani. A formal specification of the Fractal component model in Alloy. Research Report RR-6721, INRIA, 2008.

[51] D. Moody and J. van Hillegersberg. Evaluating the visual syntax of UML: An analysis of the cognitive effectiveness of the UML family of diagrams. In D. Gaevic, R. Lmmel, and E. Van Wyk, editors, *Software Language Engineering*, volume 5452 of *Lecture Notes in Computer Science*, pages 16–34. Springer Berlin / Heidelberg, 2009.

[52] Object Management Group. UML Superstructure Specification, 2009.

[53] OMG. CORBA components. OMG Specification formal/02-12-06, Object management Group 2006, 2006.

[54] OMG. UML 2.4 specification. OMG document ptc/2010-11-14, Object Management Group 2011, 03 2011.

[55] OSGi Alliance. OSGi service platform v4.2. Core specification, OSGi Alliance 2009, 2009.

[56] F. Plasil and S. Visnovsky. Behavior protocols for software components. *IEEE Trans. Software Eng*, 28(11):1056–1076, 2002.

[57] H. C. Purchase, M. McGill, L. Colpoys, and D. Carrington. Graph drawing aesthetics and the comprehension of uml class diagrams: an empirical study. In *Proceedings of the 2001 Asia-Pacific symposium on Information visualisation - Volume 9*, APVis '01, pages 129–137, Darlinghurst, Australia, Australia, 2001. Australian Computer Society, Inc.

[58] D. Rafiei. Effectively visualizing large networks through sampling. In *Visualization, 2005. VIS 05. IEEE*, pages 375 – 382, oct. 2005.

[59] Ratneshwer and A. K. Tripathi. Dependence analysis of software component. *SIGSOFT Softw. Eng. Notes*, 35:1–9, July 2010.

[60] R. Rosenholtz, Y. Li, and L. Nakano. Measuring visual clutter. *Journal of Vision*, 7(2), August 2007.

[61] S. Schaeffer. Graph clustering. *Computer Science Review*, 1(1):27–64, 2007.

[62] K. Siau and Y. Tian. A semiotic analysis of unified modeling language graphical notations. *Requirements Engineering*, 14:15–26, 2009. 10.1007/s00766-008-0071-7.

[63] J. Snajberk, L. Holy, and P. Brada. Comav - a component application visualisation tool. In *Proceedings of International Conference on Information Visualization Theory and Applications*. SciTePress, 2012.

[64] Sun Microsystems. Enterprise JavaBeans(TM), version 3.0. EJB Core, Sun Microsystems, 2006, 2006.

[65] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley / ACM Press, 3rd edition, 2002.

[66] A. Telea and L. Voinea. A framework for interactive visualization of component-based software. In *Proceedings of the 30th EUROMICRO Conference*, pages 567–574, Washington, DC, USA, 2004. IEEE Computer Society.

[67] A. Telea, L. Voinea, and H. Sassenburg. Visual tools for software architecture understanding: A stakeholder perspective. *IEEE Softw.*, 27:46–53, November 2010.

[68] B. Tversky, J. B. Morrison, and M. Betrancourt. Animation: can it facilitate? *Int. J. Hum.-Comput. Stud.*, 57(4):247–262, Oct. 2002.

[69] J. VANWIJK and W. NUIJ. Amodel for smooth viewing and navigation of large 2d information spaces. In *IEEE Trans. Visual. Comput. Graph. 10, 4,*, page 447458. IEEE, 2004.

[70] C. Walshaw. A multilevel algorithm for force-directed graph drawing. In *Proceedings of the 8th International Symposium on Graph Drawing*, GD '00, pages 171–182, London, UK, UK, 2001. Springer-Verlag.

[71] J. Wu, A. E. Hassan, and R. C. Holt. Comparison of clustering algorithms in the context of software evolution. In *Proceedings of the 21st IEEE International Conference on Software Maintenance*, ICSM '05, pages 525–535, Washington, DC, USA, 2005. IEEE Computer Society.