Visualization of Areas of Interest in Component-Based System Architectures

Heorhiy Byelas
Technische Universiteit Eindhoven
h.byelas@tue.nl

Egor Bondarev
Technische Universiteit Eindhoven
e.bondarev@tue.nl

Alexandru Telea
Technische Universiteit Eindhoven
a.c.telea@tue.nl

Abstract

Understanding complex component-based systems often requires getting insight in how certain system properties, such as performance, trust, reliability, or structural attributes, correspond to the actual system architecture. Such properties can be seen as defining several 'areas of interest' over the system architecture. We present an interactive tool that efficiently and effectively combines visual presentation of component-based architectures with that of areas of interest. Our tool helps users investigate how various system properties correlate with each other and correspond to the actual architecture, while preserving the visual architecture layout familiar to designers. We validate our tool and the proposed techniques on a component framework used in the industry.

1. Introduction

Component based software engineering is a promising way towards reducing software development time and costs. Several methods exist that help system architects and developers to describe and understand component-based systems. Visual UML-like diagrams are used to describe the compositional aspects of the system, i.e. the various components, interfaces, and structural dependencies thereof [11][12]. Software metrics can effectively describe various aspects of complex architectures, e.g. resource usage, component complexity, system stability, or performance [3][6][7]. Metrics can answer complex, targeted questions, such as "which components are unstable or non-conforming to specific guidelines and requirements?" or "what happens if I change this component?" [10]

Components that share some common property are of particular interest in system analysis, e.g. "all high-reliability components", "all components using over 1 MB of memory", "all components introduced in the system version 2.3", or "all components in the same thread" [15]. We call such a set of components an *area of interest* (AOI). AOIs can be specified using various software metrics [4][6][7] which can be computed by existing analysis tools [2][18].

However, such metrics (defining AOIs) are usually shown to users in a tabular format. A better way is to visually combine the AOIs with the UML (architecture) diagrams, to let users correlate concerns (described by AOIs) with the system structure (diagrams). We present here an approach that combines architectural and AOI information for component-based systems in an integrated, interactive visualization. Users can smoothly navigate between views of classical UML diagrams and AOIs, while preserving the familiar diagram layout. We detail how to visualize multiple, possibly overlapping, AOIs, and demonstrate our approach on a real-life industrial component framework.

Section 2 presents related work in visualizing AOIs and diagram data. Section 3 details the new techniques we propose to combine AOIs and diagrams in an effective and efficient way. Section 4 presents our case study on an industrial component-based system. Section 5 discusses our findings and the lessons learnt. Section 6 concludes our paper and outlines directions for future work.

2. Related work

We describe our goal of visualizing areas of interest on component architectures with the 5-dimensional model of Marcus *et al.* [9]: task, audience, target, medium, and representation. Our **task** is to understand how various (non-)functional system properties, described in terms of *areas of interest*, correspond to the system architecture, described by UML-like diagrams. In this work, we assume areas of interest are specified by already-computed software metrics. Our **audience** covers system architects and

developers. Our visualization **target** is the system architecture (a set of component diagrams) enriched with metrics that describe various AOIs. The visualization **medium** is the standard PC display. Finally, the **representation** enriches the classical box-and-line UML-like diagram drawings with areas of interest, drawn as soft textured images using a novel technique.

UML modeling tools, e.g. Rational Rose [11] or Together [12], are the most accepted way to visualize software architectures. However, such tools have little support to add extra information to the picture, e.g. to support questions as "which components are in a given AOI?" One could show an AOI by marking its component icons with the same color. A related approach draws icons on components, scaled, colored, and shaped to show metric values [14]. Yet, this color or icon marking technique is hard to follow visually and non intuitive on large diagrams (see Figure 8 later in this paper). Another option is to move component icons that are in an AOI close to each other and draw a surrounding frame around them [5]. However, changing the layout is usually unacceptable: Diagrams are often built manually with great care to reflect various user semantics. Also, drawing both AOIs and components as 'boxes' (frames) easily leads to confusion between the two. Finally, both icon/color marking and framing scale very badly when showing several (overlapping) AOIs.

We describe next our novel approach whose goal is to overcome these limitations while showing AOIs.

3. Visualizing areas of interest

In designing a way to visualize AOIs, the following requirements must be met:

a) AOIs must not change the given diagram layout
b) AOIs should be drawn with minimal visual clutter, even when they overlap
c) AOIs and diagrams should not visually interfere, i.e. the two should be drawn in different ways
d) AOI drawing should be fast, even for large diagrams

As a design start point, we propose to render AOIs in a similar way human users draw them with pen on paper diagrams, i.e. as some vague, sketchy, rounded, shapes that surround the concerned components. We next present an automated two-step method that addresses all our requirements, as follows. First, we build a *skeleton* of the AOI using the components' geometric layout data (Sec. 3.1), thereby addressing requirement (a). Next, we draw the AOI using a graphics technique called *texture splatting* (Sec. 3.2). By controlling the various splatting parameters, we address requirements (b,c,d). All these techniques are described next.

3.1. Skeleton construction

The input of the first step is the set of components in a given AOI. For every component, we assume we have its geometric layout information, i.e. the position and size of its 2D rectangular bounding box. We now build a *skeleton* of the AOI as follows (see Figure 1, which illustrates the complete process on a simple AOI that contains three components).

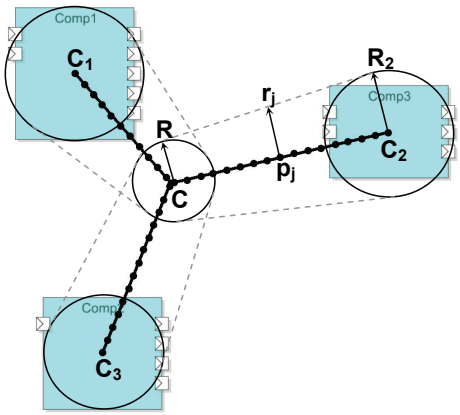

Figure 1: Geometric skeleton construction

First, we compute the center $C = \sum_{i=1}^{N} A_i C_i \Big/ \sum_{i=1}^{N} A_i$ of the AOI as the average of the component icons' centers C_i weighted by their areas A_i. Next, we compute a radius $R_i = 1/2 \max(w_i, h_i)$ for each component, where w_i and h_i are the width and height of the component's icon, and a radius $R = k \sum_{i=1}^{N} A_i R_i \Big/ \sum_{i=1}^{N} A_i$ for the center C as a fraction k of the average radius. For all images in this paper, we used k=0.8. Next, we sample every line segment CC_i with several points p_{ij} so that the distance between two consecutive points p_{ij} and p_{ij+1} is a small fraction δ of R. For all images in this paper, we used $\delta = 0.1 * R$. For every p_{ij}, we compute also a radius r_{ij} by linear interpolation between the radii R and R_i at the end of the segment CC_i. The geometric skeleton is now the complete set of points and radius values $\{(p_{ij}, r_{ij})\}_{i,j}$.

3.2. Texture splatting

We now use the skeleton to draw the AOI, as follows. First, we construct a so-called *splat*. This is a

radial function $T(x,y)=f\left(\sqrt{x^2+y^2}\right)$. T looks as shown by Figure 2a (dark=opaque, light=transparent). Here, f is the *profile*, or shape, of the splat. We shall use $f(x)=x^k$, so T increases linearly with the distance for k=1, exponentially for k>1 and logarithmically for k<1 (see Figure 2b). T is implemented as a transparency (also called alpha) texture using the OpenGL graphics library [17]. Hence, T=0 yields fully transparent pixels and T=1 fully opaque pixels. We now draw the AOI simply by rendering the texture T centered at every skeleton point p_{ij}, scaled by the radius r_{ij}, and colored by a user specified AOI color. Finally, we draw the component diagram itself as usually, atop of the AOI.

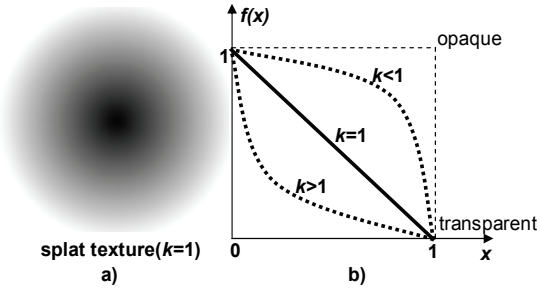

Figure 2: Splat texture (a) Texture profile (b)

Figure 3 shows the result of the texture splatting for the AOI of the components in Figure 1. Several properties of our method are visible here. First, the AOI is visually quite different (i.e, soft and round) from the component diagram, which is sharp and drawn with straight lines. This distinguishes the two visually, hence addressing requirement (c). Second, our splatting method is a robust, simple and fast way to draw a shape that contains all components in an AOI and also has a simple, predictable 'look', even for AOIs containing many components scattered all over a diagram. This addresses requirement (d).

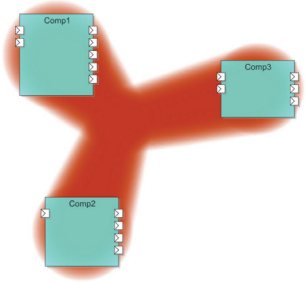

Figure 3: Area of interest drawn with splatting

By changing various parameters of the splat texture, we obtain different visual effects useful for different user scenarios. If we want to draw 'hard' AOIs with a sharp, precise, border, we set k<1 (e.g. k=0.3, Figure 4a). This is useful e.g. to show important system properties or metrics having a high confidence value. If we want to draw 'soft', fuzzy AOIs, we set k>1 (e.g. k=5, Figure 4b). This is useful e.g. to show less important properties, which should distract less the user's eye from the more important diagram drawing, or metrics having a low confidence value. Clearly, many other scenarios are possible too.

A second variation our users found useful during our case studies was to draw AOIs as *contours* instead of filled shapes. This is easily achieved in two passes. First, we draw the filled AOI using the splat textures, as described so far. Second, we draw the same AOI, using the same splat texture centered at the skeleton points, but now scaled to a smaller radius $d*r_{ij}$, and using the background color (e.g., white). Here, $d \in [0,1]$ controls the contour width: d=0 yields the filled shapes, while a d close to 1 yields a very thin contour. Just as before, k controls the contour sharpness. Figure 4(c,d) shows two examples of areas of interest drawn with contours, where we used a contour width d=0.8.

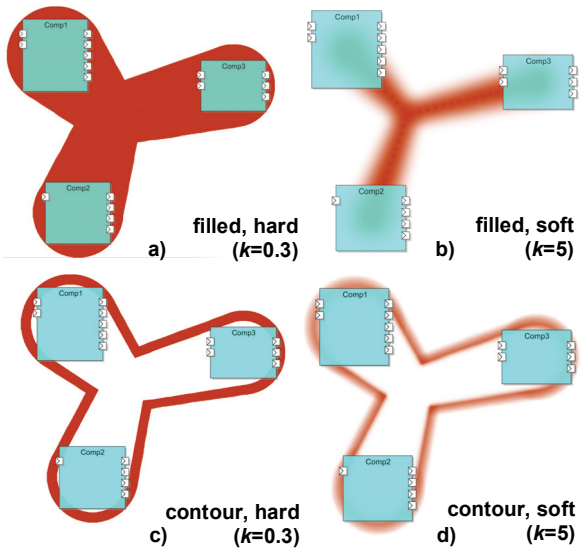

Figure 4: Area of interest drawing variations

3.3. Erasing overlapping components

The drawing method described so far does indeed guarantee that the drawn shape visually surrounds all components in the AOI. However, the drawn shape might surround, or overlap with, components which are *not* in the AOI, e.g. the marked one in Figure 5a. This is, of course, an undesired side effect.

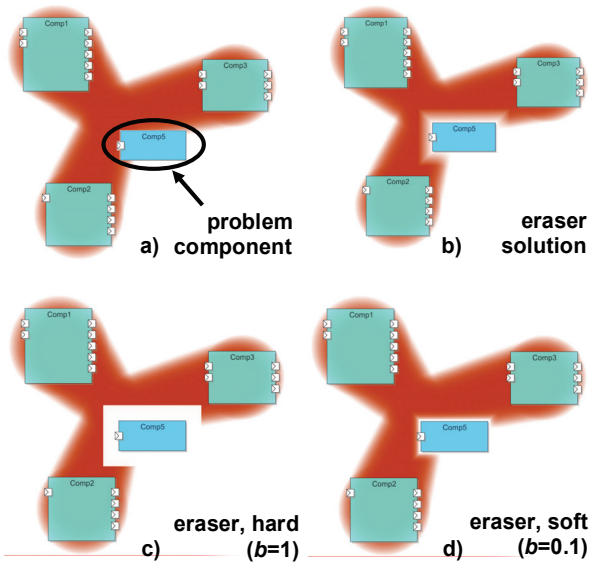

Figure 5: Erasing overlapping components

We solve this problem as follows. First, we draw all AOIs as described so far. Next, for all components not in any AOI, we draw an *eraser* texture. This is a transparency texture, like the splat texture (Figure 2) used to draw the AOIs, except that it has a rectangular (instead of radial) shape (see Figure 6a) and a profile given by a slightly different function. Instead of $f(x)=x^k$, we use now the following profile f (Figure 6b):

$$f(x)=\begin{cases} 1 & ,x<b \\ \left(\frac{x-b}{b}\right)^k & ,x\geq b \end{cases}$$

Using a fixed k=4 and varying b in [0,1] yields an eraser ranging from hard (b=1) to very soft (b=0.1), as shown in Figure 5. The value b=0.8 is a good default.

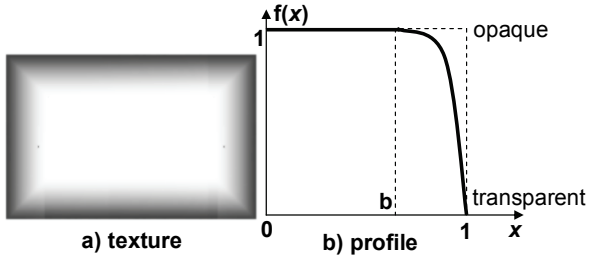

Figure 6: Eraser texture

Drawing the eraser texture mapped on background-colored (white) rectangles slightly larger than the components effectively erases the AOIs underneath, yielding the effect shown in Figure 5b. The component that was erroneously overlapping with the AOI appears now to be outside the AOI. As for the splat textures, we can control the eraser strength by the k parameter, yielding results from hard to soft (Figure 5c,d).

3.4. Drawing multiple areas

Drawing multiple AOIs is simple: We just apply the skeleton construction and rendering described so far, using different user-specified colors, for each area in turn. If desired, we let users specify *priorities* for every AOI. Next, we draw AOIs from low to high priority, thereby ensuring that high-priority AOIs will always be more visible, since drawn atop of low-priority AOIs.

4. Applications

We implemented all our visualization methods in the MetricView tool [14]. MetricView is an interactive software architecture visualization tool which combines metric and UML diagram data. We next present a case study where we used our visualization tool.

4.1. Context of assessment

Within the ITEA Trust4All project [13], we have developed a Real-Time Integration Environment (RTIE) toolset that provides design and development of embedded real-time, component-based systems, based on the Eclipse platform[2]. RTIE contains three tools: Repository, Composer and Quality Attribute Analyzer (QAA). The Repository provides storage, search and retrieval of third-party components, and also stores component models representing abstract specifications of the component quality attributes. The Composer allows the application designer to select and instantiate components and bind their provided and required interfaces, thereby specifying a desired system software architecture, all via point-and-click mouse operations in a GUI-based tool. The QAA tool performs design-time analysis and prediction of various quality attributes of the designed system, e.g. reliability, hardware resource usage, task delays and throughput. Finally, our extension of the MetricView tool reads the output of the Composer tool (i.e. a component diagram) and QAA (i.e. software metrics) and visualizes the composition together with areas of interest determined by the predicted quality attributes, using the techniques described in 3.

The RTIE toolset uses the ROBOCOP component model [8]. ROBOCOP was developed during a 5 year period by an international consortium joining industry and academia. ROBOCOP provides a generic, flexible, and resource-efficient set of mechanisms and tools to

implement, compose, deploy, and monitor component-based software applications for high-volume embedded appliances, e.g. mobile phones, set-top boxes, and embedded controllers. Overall, its component model is similar to Rubus [1] and PECOS [16].

4.2. Case study: The Car Media Center

We have used the RTIE toolset described above in a case study on a Car Media Center (CMC) real-time system. The CMC has the following functionality: (a) GPS-based car navigation; (b) radio and digital TV reception and rendering; (c) CD/DVD playback. We designed the CMC system with out RTIE toolset out of both proprietary and 3rd-party ROBOCOP components. Figure 7 is an actual snapshot from the Eclipse-based Composer GUI. It shows the CMC system design consisting of 28 component instances together with the connections between their various provided interfaces (drawn at the left of the component icons) and required interfaces (drawn at the right of the component icons).

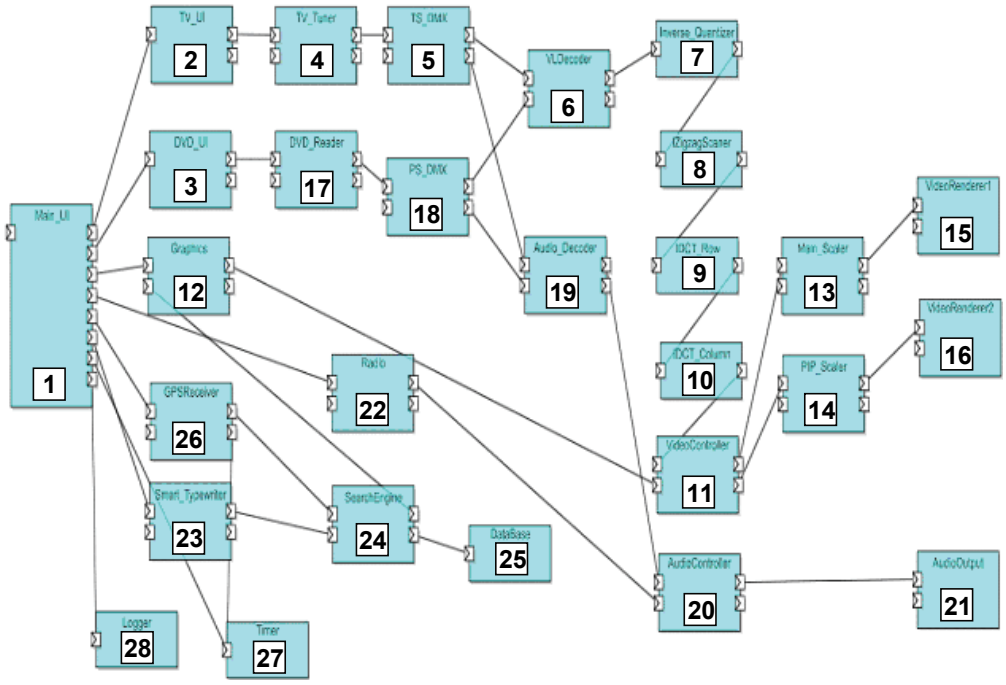

Figure 7: Car Media Center component-based design (GUI-based Composer tool snapshot)

Let us briefly explain the component functions in the CMC system. The Main_UI component (1) receives user input by polling the state of the buttons on the CMC dashboard. The TV_UI (2) and DVD_UI (3) components receive and process TV and DVD-related user commands. TV_UI sends the currently selected TV channel to the TV_Tuner (4) component that does the TV tuning. The transport bit stream of the chosen TV channel is sent to the TS_DMX (5) component, which de-multiplexes the stream into video and audio. The video stream is next processed by several video filters: VLDecoder (6, variable length decoder), Inverse Quantizer (7), IZigzag_Scanner (8, inverse zigzag scan), IDCT_row and IDCT_column (9, 10, inverse row/column discrete cosine transform). The decoded video stream is next sent to the VideoController (11) component, which specifies on which display to render the video. A second video stream comes to the VideoController from the Graphics (12) component carrying the graphical data (UI and navigation) coming from the Main_UI component. The VideoController outputs two video streams to the Main_Scaler (13, scales images to display size) or the PiP_Scaler (14, scales images to picture-in-picture format). Two VideoRenderer (15, 16) components perform the actual display rendering and update. The audio path starts from the TS_DMX (5) or DVDReader (17) and PS_DMX (18) components, goes to the AudioDecoder (19) and AudioController (20), and ends up in the AudioOutput (21) component, which controls the car loudspeakers. AudioController also accepts the audio stream from the Radio (22) component and decides which of the two streams to play. Finally, the car navigation is

implemented as follows. The user inputs an address via the Smart_Typewriter (23) component. The address is next sent to the SearchEngine (24) component, which finds the desired location by querying the DataBase (25) component, compares it with the current car location received from the GPSReceiver (26) component, and computes the best driving path. The driving path and instructions are sent to the Graphics component for video rendering and to the AudioController component for voice messages. Finally, the Timer (27) and Logger (28) components perform system-wide synchronization and logging.

4.3. System analysis with areas of interest

The architects of the CMC system were interested in several aspects. Among others, these were:

- How are component functions related to vendors?
- Which components are on the video or audio paths?
- Which components have user interface functions?
- Is performance-sensitivity related to functionality?
- How is availability related to functionality?

All aspects (vendor, performance, availability, etc) were represented by metric values obtained from the RTIE toolset. Based on the values of these metrics, our users created next several areas of interest, as follows:

Area	Explanation
A_1	Components produced by Vendor A
A_2	Components produced by Vendor B
A_3	Components on the video path
A_4	Components on the audio path
A_5	Availability-sensitive components
A_6	Performance-sensitive components
A_7	Interaction-sensitive (GUI) components

We first visualized these areas of interest (AOIs) using the standard metric icons provided by MetricView [14], by assigning different icon shapes and colors to every AOI. Components in one AOI thus share the same icon shape and color. Figure 8 shows the result. As expected, this visualization is not easy to follow. Next, we visualized the same AOIs, this time using our new splatting method. Figure 9 shows the result. We used here the same area colors as for the metric icons in Figure 8. Clearly, the AOIs, and their relations with the system structure, are now easier to follow. Looking at the Vendors and Paths visualizations, we see now easily that all video components (A_3) come from vendor A (A_1). Looking at the component functions (see 4.2), we concluded that vendor B (A_2) provided all the navigation (GPS)-related components. The Paths visualization also reveals some insight about the diagram layout, which was manually constructed by the designed in the RTIE Composer tool. Clearly, its upper part (A_3) contains the video path, its lower part (A_4) the audio path, and components are laid out from left (path begin) to right (path end). Comparing the Sensitivity visualization with the Vendors and Paths visualizations in Figure 9 answers further questions. We see that only the video components (A_1) are performance-sensitive (A_6). The interaction-sensitive components (A_7) are found only at the beginning of both video and audio paths (A_3,A_4). Only components from vendor B (A_2) have availability-related problems (A_5), except the video component '11' which is from vendor A. Finally, we locate two interesting components (VideoController and Main_Scaler, denoted '11' and '13' in both Figure 7 and Figure 9) which are both performance and availability-sensitive.

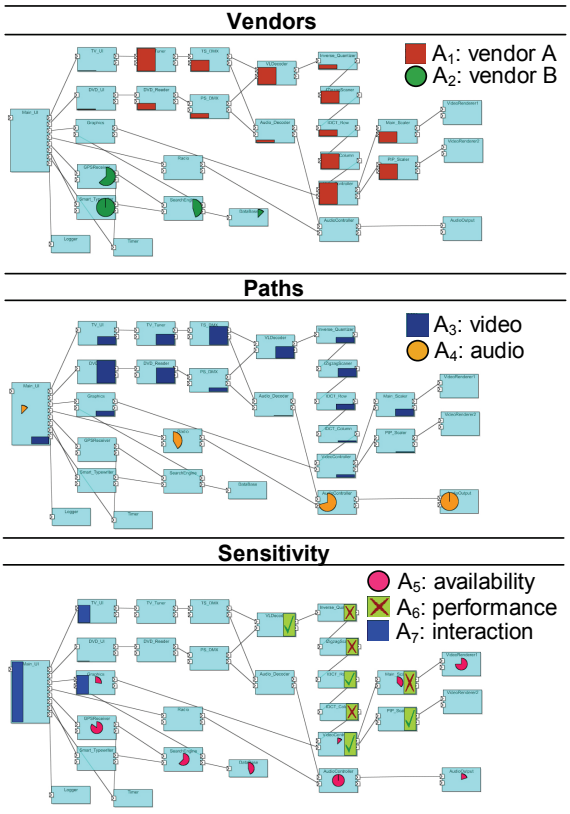

Figure 8: AOIs for the CMC system (icons)

The AOIs are easier to follow than metric icons for scenarios as described above. Yet, metric icons are better when one wants to visually compare individual metric values. We can easily combine the power of the

AOIs (showing *subsets* of interest of a system's architecture) with the metric icons (showing *individual* component properties), by displaying the two atop of each other in a 3D view. Figure 10a shows such a visualization, made with our modified MetricView tool. Here, we show the audio components as an AOI and the 'video processing power demands' metric as vertical metric bars (the bar heights show actual metric values).

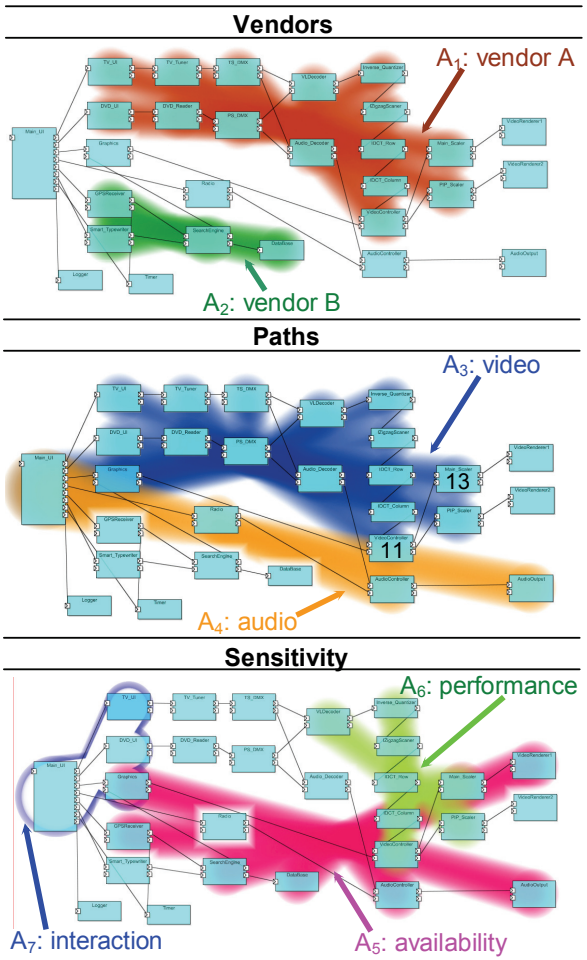

Figure 9: AOIs for the CMC system (splatted)

Finally, we mention that our AOI technique is not limited to *component* diagrams. Figure 10b shows an AOI rendered with our extended MetricView tool on a UML class diagram modeling a lift control software system, as well as various software metrics relevant to this application. This figure also illustrates the overall look of our tool's user interface, similar to [14].

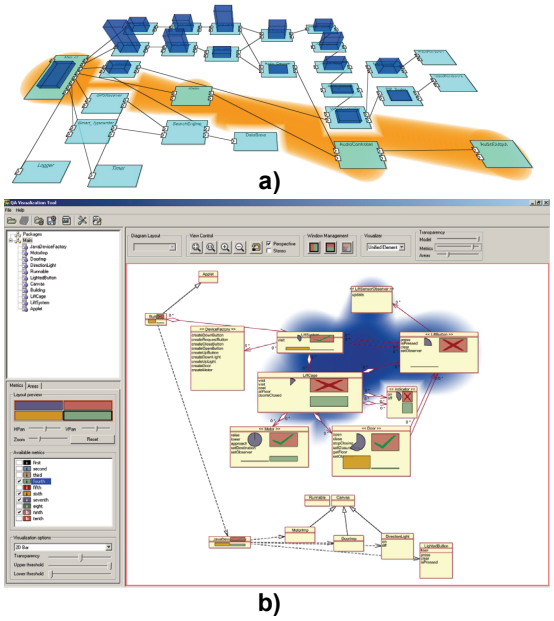

Figure 10: Diagrams, metrics, and AOIs

5. Discussion

To implement the material presented in this paper, we added a component diagram renderer to MetricView (which only supported class diagrams, state charts, and message sequence charts), and a renderer for AOIs. Our work was greatly eased by the choice to represent AOIs as metric values, which are already supported by the original MetricView tool. We designed the component diagram renderer to produce near-identical drawings with the Eclipse-based Composer GUI (compare Figure 7 with Figure 9). The AOI renderer was written in OpenGL in a few hundred lines of C++. Adding AOIs to other designer tools, e.g. [11][12], should be very easy, once one has access to the tool renderer code and this renderer supports an OpenGL-like API. As outlined in Sec. 3, rendering an AOI involves drawing a few tens (at utmost, hundreds) of transparency textures, an operation that OpenGL can do in real-time, even for very large diagrams. This enables users to interactively edit component diagrams, e.g. by dragging component icons in the tool GUI, while the AOI rendering is updated on-the-fly.

The technical description of the AOI rendering in Sec. 3 involves many parameters, which may suggest that users have to tune many values in the MetricView tool to get a useful visualization. This is not the case. We mentioned all these parameters just to make the explanation of our technique detailed and complete. In the MetricView tool GUI, users actually tune just a few parameters: AOI color, drawing mode (filled or contour), and AOI transparency, and use default values

for the rest. Using AOIs is very simple and intuitive. Making the pictures in Sec. 4.3 took a few minutes for users already familiar with our MetricView tool.

The greatest limitation of the AOI method presented here is that it cannot avoid undesired overlaps of the rendered areas. However, this is very hard to avoid, given that we impose ourselves not to alter the diagram layout. We are working to minimize the AOI overlap by means of more advanced skeleton designs.

6. Conclusions

We have presented a technique that adds areas of interest (AOIs) to the rendering of classical component diagrams. We implemented our work in MetricView, an existing metric-and-diagram interactive visualization tool. Throughout our work, users and their preferences stood central: First, we use the UML-like diagrams and graphical layouts familiar to architects and developers. Second, users can navigate between classical UML-like diagram drawing and the AOIs by the simple use of a transparency slider. Third, AOIs are defined easily and flexibly using software metric values. Since the original MetricView tool clearly separates metric and diagram specification, we can define and/or change any number of AOIs per user scenario without touching the diagram data and/or its XMI input format. This decoupling of concerns allowed us to integrate our visualization tool in the already existing RTIE toolset for component system design and simulation, all in a few hours. The only code we needed to write for this was a plug-in for MetricView to accept RTIE's metric output format.

We are already investigating several extensions of our AOI techniques. We look at ways to parameterize the AOI rendering (e.g. color and texture) by actual metric values, in order to display metric values of whole component sets. A second, more challenging, direction we are working on is a better AOI skeleton generation algorithm to provide better-looking, less overlapping, AOI shapes, targeted to support very large component diagrams with many complex-shaped areas of interest.

7. Acknowledgments

This research is part of the ITEA project Trust4All, which aims to develop models and methods to describe, evaluate, and assess trust and other non-functional parameters in component-based frameworks used in the middleware of high-volume embedded appliances [13]

8. References

[1] Articus Systems, *Rubus OS Reference Manual*, 1996

[2] Bondarev, E., Chaudron, M., de With, P. *A Process for Resolving Performance Trade-offs in Component-Based Architectures*, Proc. 9th Intl. Symposium on Component-Based Software Engineering, Springer LNCS, 2006, to appear

[3] Dumke, R. Schmietendorf, A. *Possibilities of the Description and Evaluation of Software Components*, Metrics News, vol. 5, 2000.

[4] Fenton, N., Pfleeger, S. *Software Metrics: A Rigorous and Practical Approach*, Chapman & Hall, London, 1998

[5] Gansner, E., North, S. C. *An open graph visualization system and its applications to software engineering*, Software: Practice & Experience, vol. 30, no. 11, J. Wiley & Sons, 2000, pp. 1203– 1233

[6] Gill, N., Grover, P. *Component-Based Measurement: A Few Useful Guidelines*, ACM SIGSOFT Software Engineering Notes, vol. 28, 2003, ACM Press.

[7] Goulão, M., Abreu, F. *Formalizing Metrics for COTS*, Proc. MPEC'04, Edimburgh, 2004

[8] ITEA, *ROBOCOP: Robust Open Component Based Software Architecture for Configurable Devices.* Public document, version 1.0, May 2002; available online at: http://www.hitech-projects.com/euprojects/robocop/

[9] Marcus, A., Feng, L., Maletic, J. I. *3D Representations for Software Visualization*, Proc. ACM SoftVis'03, ACM Press, 2003, pp. 27– 36.

[10] A. Möller, M. Åkerholm, J. Fredriksson, and M. Nolin, *Evaluation of Component Technologies with Respect to Industrial Requirements*, *Proc. EUROMICRO'04*, IEEE Press, 2004, pp. 56 – 63.

[11] Rational Rose: www.306.ibm.com/software/rational/

[12] Together: http://www.borland.com/together, 2005

[13] Trust4All ITEA project: http://www.win.tue.nl/trust4all/

[14] Termeer, M., Lange, C., Telea, A., Chaudron, M. *Visual exploration of combined architectural and metric information*, Proc. Vissoft'05, IEEE Press, 2005, pp. 21– 26

[15] Voinea, L., Telea, A., *A framework for interactive visualization of component-based software*, *Proc. EUROMICRO '04*, IEEE Press, 2004, pp. 567 – 574

[16] M Winter, T Genssler, *Components for Embedded Software – The Pecos Approach,* Proc. 2nd Intl. Workshop on Composition Languages, ECOOP'02, June 11, 2002

[17] Woo, M., Neider, J., Davis, T. Shreiner, D. *OpenGL Programming Guide*, 3rd edition, Addison-Wesley, 2001

[18] Wust, J. *SDMetrics: The software design metrics tool for UML*, http://www.sdmetrics.com, 2005