Visualization of Areas of Interest in Software Architecture Diagrams

H. Byelas[*]
Technische Universiteit Eindhoven

A. Telea[†]
Technische Universiteit Eindhoven

Abstract

Understanding complex software systems requires getting insight in how system properties, such as performance, trust, reliability, or structural attributes, correspond to the system architecture. Such properties can be seen as defining several 'areas of interest' over the system architecture. We visualize areas of interest atop of system architecture diagrams using a new technique that minimizes visual clutter for multiple, overlapping areas for large diagrams, yet preserves the diagram layout familiar to designers. We illustrate our proposed techniques on several UML diagrams of complex, real-world systems.

CR Categories: I.3.4 [Graphics Utilities]: Graphics editors—Paint systems; D.2.2 [Design Tools and Techniques]: Modules and interfaces—Computer-aided software engineering

Keywords: UML diagrams, metrics, areas of interest, architecture visualization

1 Introduction

UML (or similar) diagrams are among the methods of choice for system architects and developers to describe and understand software architectures and designs, e.g. the structural and functional relations between the various interfaces, components, objects or roles in a system [IBM 2005; Borland 2005]. Complementing diagrams, software metrics effectively describe various aspects of complex systems, e.g. system stability, resource usage, design complexity, or performance [Dunke and Schmietendorf 2000; Gill and Grover 2003; Goulao and Abreu 2004]. Metrics can help answering targeted questions, e.g. "which components are unstable or non-conforming to specific guidelines and requirements?" or "what happens if I change this component?" [Möller et al. 2004] Software elements that share a 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" [Voinea and Telea 2004]. We call such a set of elements an area of interest (AOI). AOIs can be defined using software metrics [Fenton and Pfleeger 1998; Gill and Grover 2003; Goulao and Abreu 2004] which can be computed by existing analysis tools [Wust 2005; Bondarev et al. 2006, to appear]. Such AOIs, and their underlying metrics, are usually shown to users in a tabular format. We argue it is better to visually combine AOIs and UML (architecture) diagrams, to let users correlate concerns (described by AOIs) with system structure (diagrams). We present an approach that visualizes AOIs on UML-like diagrams in a simple to follow, scalable, and non-intrusive way. Users can easily navigate between views of classical diagrams and AOIs, yet preserve the familiar diagram layout. Our technique scales well when visualizing multiple, overlapping, AOIs on large diagrams, and works for any UML-like diagrams. This paper is structured as follows. Section 2 reviews related work in visualizing AOIs and diagram data. Section 3 presents our new techniques that render AOIs on diagrams effectively and efficiently. Section 4 presents several applications of our AOI drawing on real-life diagrams from the industry. Section 5 discusses our findings and the lessons learnt. Section 6 concludes the paper with directions of future work.

2 Related Work

We describe our AOI visualization on (UML) diagrams with the 5-dimensional model of Marcus *et al.* [Marcus et al. 2003]: task, audience, target, medium, representation. Our task is to understand how various (non)functional system aspects (the AOIs) map on some system description (the UML-like diagrams). Our audience includes mainly software architects. Our visualization target is a set of UML-like diagrams, together with AOIs specified as already computed software metrics for the diagram elements [Fenton and Pfleeger 1998]. The visualization medium is a modified UML diagram viewer [Termeer et al. 2005] that combines rendering diagrams and AOIs. Finally, the representation enriches classical box-and-line diagram drawings with AOIs drawn as smooth, soft-textured, shapes with a new technique.

Modeling tools, e.g. Rational Rose [IBM 2005] or Together [Borland 2005], are accepted ways to visualize UML diagrams, but have little support for metric data, and still less for drawing metric-defined AOIs. Drawing AOIs as boxes AOI yields high visual clutter, as illustrated by the 12 AOIs drawn on the diagram in Fig. 16. Other tools, e.g. Rigi [Tilley et al. 1994] or MetricView [Termeer et al. 2005], can show an AOI by marking its elements with with icons scaled, colored, and shaped to show metric values. Yet, inferring AOIs from such markers is very hard for large diagrams having more than a few, overlapping, AOIs. One could also move all diagram elements in an AOI close to each other and draw a surrounding frame [Gansner and North 2000]. However, diagrams are often laid out manually with great care. Changing the layout every time one changes the AOIs destroys the user's 'mental map', a well known fact in information visualization. Methods such as metaballs [Rilling and Mudur 2002], H-BLOB [Sprenger et al. 2000], and 2D implicit surfaces [Balzer and Deussen 2005] draw AOIs as smooth shapes around their respective elements. Such shapes are computed as isosurfaces of some potential function, or distance field, based on the elements' locations. However, it is hard to control both the smoothness and tightness of isosurfaces. The isosurface shapes and, worse, connectivity, highly depend on a correct isovalue, which cannot be easily chosen automatically [Sprenger et al. 2000]. Finally, distance fields and isosurfaces are computationally expensive. Recently, a method based on texture splatting has been proposed that efficiently draws smooth AOIs of controlled shape with minimal user intervention [Byelas and Telea 2006], using a technique called texture splatting. However, this method is limited in handling complex, overlapping AOIs on large diagrams.

[*]e-mail:h.byelas@tue.nl
[†]e-mail:alext@win.tue.nl

3 New Proposal

We build on the texture splatting idea of [Byelas and Telea 2006], keeping its main features (high speed and minimal user input). Additionally, we add new techniques to solve the problems of the original method, i.e. we can easily handle many complex-shaped, overlapping, AOIs, and offer a simple and intuitive AOI shape smoothness and tightness control. Overall, the features of our method are as follows:

1. AOIs do not change a given diagram layout
2. There is low visual clutter between (overlapping) AOIs and diagrams, and AOIs themselves
3. AOIs drawing is real-time, even for large diagrams

Our AOI visual design tries to mimic the way humans draw them with pencil on paper diagrams, as vague, sketchy, imperfect shapes that surround the concerned elements. We construct such shapes in two steps. First, we build an AOI skeleton from the elements' size and position (Sec. 3.1). Next, we draw the AOI using a graphics technique called texture splatting (Sec. 3.2). These techniques are described next.

3.1 Skeleton Construction

In Section 4, we shall compare, on real-world UML diagrams, the quality of the AOIs drawn using the skeleton proposed here and the skeleton proposed by the original AOI drawing method [Byelas and Telea 2006] respectively. To distinguish the two, we call the skeleton proposed by [Byelas and Telea 2006] an inner skeleton (since it is located at the center of the elements' bounding box), and the skeleton we propose here an outer skeleton (since it is located at the periphery of the elements' bounding box). We first briefly sketch the inner skeleton method (see also Fig. 1). For a diagram with elements e_i having geometric centers c_i, the inner skeleton is the line set (c_i, C), where $C = \sum_i A_i c_i / \sum_i A_i$ is the area-weighted barycenter of the elements (A_i is the area of element c_i). Given element e_i, with bounding box of width w_i and height h_i, a radius $R_i = \max(w_i, h_i)$ is computed for e_i and a radius $R = k_R \sum_i A_i c_i / \sum_i A_i$ for the center C as a fraction k_R of the average radius. Setting the value for k_R is explained in Section 3.2. Next, every line segment (c_i, C) is sampled with several points p_{ij} spaced with some small distance $\delta = |p_i - p_{i+1}|$, e.g. $\delta = 0.1R$. 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 (c_i, C). The inner skeleton is the set of points and radius values $\{(p_{ij}, r_{ij})\}$.

We now explain the outer skeleton construction. This has three steps (see Fig. 2 b-d). We start with the 2D bounding boxes $(b_{1i}, b_{2i}, b_{3i}, b_{4i})$ of the elements e_i in the AOI (Fig. 2 a). We first compute the convex hull $C = \{q_i\}$ of the corner points $\{b_{ij}\}$, yielding the result in Fig. 2 b. This is the tightest convex polygon that encloses all our element bounding boxes, i.e. a possible approximation for an AOI shape. Still, we would like smoother, tighter fitting, shapes. To obtain this, we first subsample C (Fig. 2 c) such that the average distance δ between consecutive points $|q_i - q_{i+1}|$ is a given, small fraction of the convex hull perimeter $|C| = \sum_i |q_i - q_{i+1}|$. In practice, we set $\delta = 0.01|C|$.

Next, we deform the subsampled contour q_i so that it fits tighter the elements inside and, at the same time, yields a smoother curve than the convex hull (Fig. 2 d). We deform the contour by moving every point q_i to q_i':

$$q_i' = q_i + \varepsilon_n \vec{n} + \varepsilon_s \frac{q_{i-1} + q_{i+1}}{2} \tag{1}$$

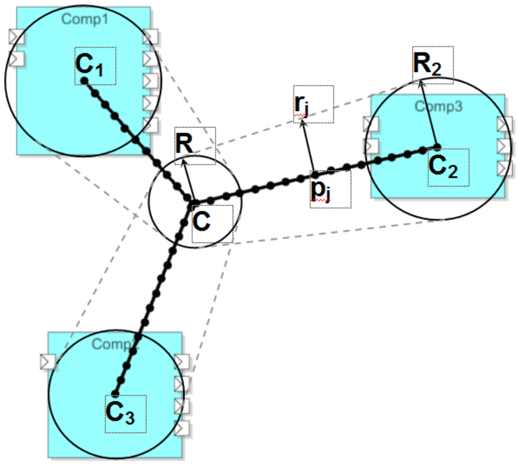


Figure 1: Construction of inner skeleton

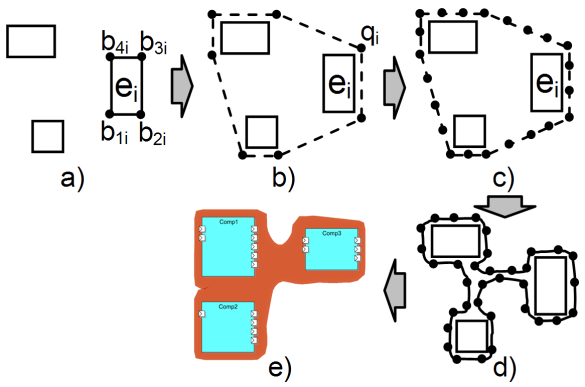


Figure 2: Construction of outer skeleton

Here, $\vec{n}$ is the normal to the line segment $(q_{i-1}q_{i+1})$. Assuming $\{q_i\}$ are specified in counterclockwise order, q_i will be moved inwards inside C. This serves two purposes. First, q_i moves perpendicular to the contour with a distance ε_n which shrinks the contour, making it tighter. Second, q_i moves towards the center of the line segment $(q_{i-1}q_{i+1})$ with distance ε_s. This is the well-known geometric Laplacian smoothing [Taubin 2000] with factor ε_s applied to our contour, which guarantees to remove contour sharp corners. We do the move in Equation 1 only if

$$d = \min\left(\min_{|j-i|>1} |q_i - q_j|, \min_j |q_i - p_j|\right) > 2\delta \tag{2}$$

i.e. the contour point q_i is farther from all element corners p_j and other contour points q_j (except its immediate neighbors q_{j-1}, q_{j+1}) than a distance 2δ. This test prevents the contour to self intersect during deformation. We move all points until we reach a user-set stop criterion or a maximum number of iterations N_{max}. Different stop criteria model different contour properties, as follows:

- Stopping when the deformed contour area $A(C)$ reaches a fraction $f_A < 1$ of the initial contour area controls the tightness of the AOI shape. Smaller f_A values mean tighter areas. Stopping after a given number of iterations $N < N_{max}$ does roughly the same and is also cheaper to implement.
- Stopping when the deformed contour length $|C|$ reaches a fraction $f_C > 1$ of the initial contour length controls the smoothness of the AOI shape. Larger f_C values mean less smooth contours.

Figure 8 shows several deformation steps for a simple AOI, starting from the convex hull until a quite tight shape, reached after 20 iterations. The parameter setting $\varepsilon_n = 0.005|C| = 0.5\delta, \varepsilon_s = |q_{i-1} - q_{i+1}|/4$, $N \in [5..20]$ and $f_C \in [1,2]$ give very good results in practice for all configurations (shape, position, and number of diagram elements). Besides preventing self-intersection, we must also prevent the contour to become too sparsely sampled, due to the contour length increase during deformation. We do this by checking the distances $|q_i - q_{i+1}|$ and $|q_i - q_{i-1}|$ between the moved point q_i and its neighbors. If these exceed 2δ, we insert a new contour point halfway between q_i and the respective neighbor. Checking for the contour becoming too densely sampled is not needed, as we know from shape processing that motion of lines (the initial convex hull edges) in (smoothed) normal direction always stretches the contour [Costa and Cesar 2001].

Fast convex hull and deformation computations are crucial for an efficient outer skeleton construction. We use the Triangle geometric library [Shewchuk 1996] which provides a state-of-the-art convex hull implementation. For the deformation, the distance testing in Equation 2 must be done very efficiently. A naive implementation would use $O(NC(NC+E))$ operations per deformation step for NC contour points and E elements, which is too slow for real-time performance. We solve this by using a fast spatial search structure that locates the nearest point q_j to the moving point q_i in $O(log(NC+E))$ operations, using kd-trees [Arya et al. 1998]. All in all, these choices let us deform complex contours containing hundreds of elements (E) and hundreds of contour points (NC) in sub-second time.

3.2 Drawing Areas by Skeleton Splatting

We now use the skeleton to draw the AOI, as follows. First, we construct a so-called splat. This is a radial function $T(x,y) = f(\sqrt{x^2+y^2})$. T looks as shown by Fig. 3 a (dark=opaque, light=transparent). Here, f is called the splat *profile*, or shape. 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 Fig. 3 b). We implement T as a transparency (also called alpha) texture with the OpenGL graphics library [Woo et al. 2001]. Hence, $T = 0$ yields fully transparent pixels and $T = 1$ fully opaque pixels.

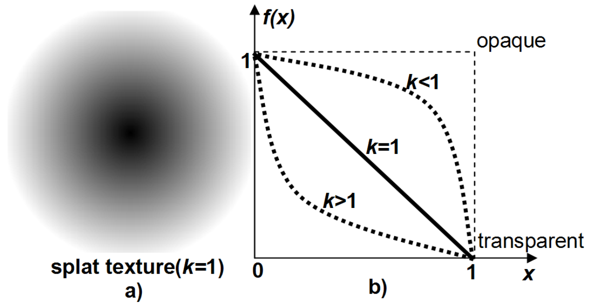


Figure 3: Splat texture(a) and texture profile (b)

The inner skeleton method [Byelas and Telea 2006] rendered the AOI by drawing the texture T centered at every skeleton point p_{ij}, scaled by the radius r_{ij}, and colored by a user specified AOI color. Figure 5 a shows the result of this method for a simple diagram containing five elements and two areas of interest.

Several properties of this method are visible here. First, the AOI is visually quite different (i.e, soft and round) from the diagram

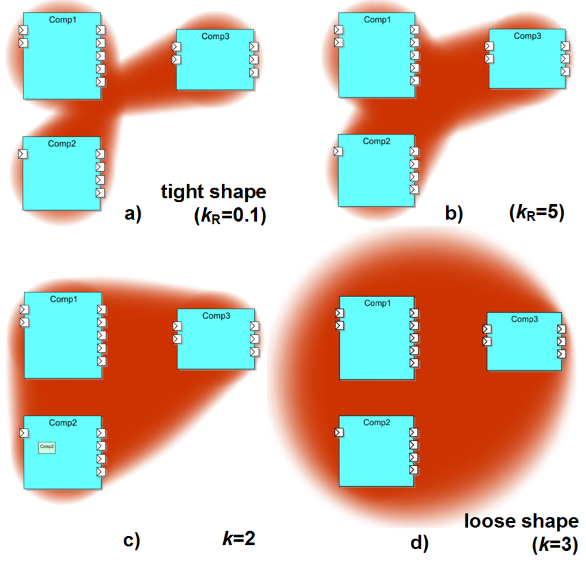
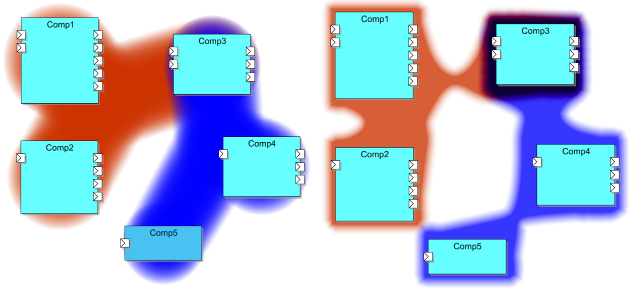


Figure 4: Tightness and smoothness control (inner skeleton)

(drawn with sharp, straight lines). This distinguishes the two visually. Splatting the inner AOI skeleton is a robust, simple and fast way to draw a shape that contains all elements in an AOI and has a simple, predictable 'look'. A first simple, but effective, improvement we propose with respect to the original method [Byelas and Telea 2006] is the choice of the radius factor k_R (introduced in Sec. 3.1). The original method set k_R to a small constant value $k_R = 0.1$. We let users vary k_R to control the tightness and smoothness of the AOI shape. Small ($k_R \in [0.1, 0.5]$) values yield the typical tight star-shaped AOIs shown by the original method (Fig. 4 a). Large ($k_R \in [1,3]$) values yield rounded, softer shapes (Fig. 4 d). In-between k_R values balance the trade-off between the shape smoothness and tightness (Fig. 4 b,c).



(a) inner skeletons (b) outer skeletons

Figure 5: Areas of interest drawing styles

However useful, the inner skeleton AOI drawing has a major problem: It scales quite poorly for diagrams having overlapping AOIs of complex shapes (see e.g. Fig. 15 (bottom). The main problem of the inner skeletons is that they have a fixed, star-like, topology, i.e. a center connected to the elements' centers. Inner skeletons work quite well for small-size AOIs (e.g. Fig. 5) or AOIs whose convex hull is close to a regular n-sided polygon, but not that well for more complex shapes (e.g. Fig. 16). Our outer skeletons solve this problem. Figure 5 b shows the same AOI as in Fig. 5 a, this time drawn using the outer skeleton, as explained next.

We draw the AOIs using the outer skeleton in two steps. First, we triangulate the deformed contour $\{q_i\}$ (Sec. 3.1) and render the re-

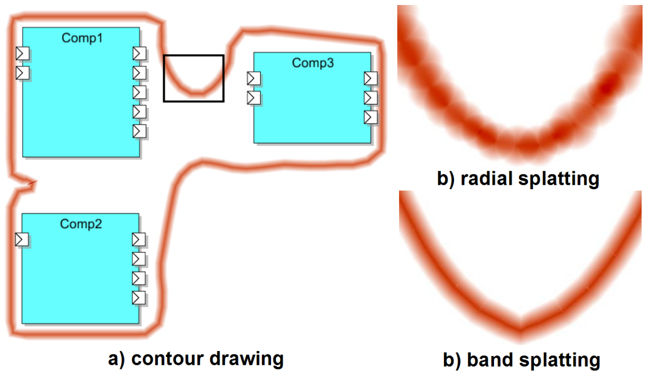


Figure 6: Contour splatting: (a) contour; details with (b) radial and (c) band splatting

sulting triangles in the area's color. This takes care of the area itself. Next, we would like to draw a soft, fuzzy contour, similar to the effect in Fig. 5 a for the inner skeleton drawing. We first tried the same idea of splatting the contour points with the radial texture. However, this requires a very high number of splats (roughly, one every few contour pixels) to produce relatively smooth border, which is quite inefficient. And the quality is still poor (see Fig. 6). The contour in Fig. 6 a is rendered with splats. We can see on the zoomed-in detail (Fig. 6 b) that, even though we are using a high splat density, the border looks jagged. We solved this problem by designed a better rendering method for the outer skeleton, as follows. We first offset the contour points q_i outwards along the contour normal $\vec{n}$:

$$q'_i = q_i + \varepsilon_n \vec{n} \tag{3}$$

Here, $\vec{n}$ and ε_n are the same as in Equation 1. This creates a narrow band along the contour (Fig. 7 a). Next, we create a 'band' texture $T(x,y) = f(x)$ (Fig. 7 b) where f is the same profile as for the splat texture (Fig. 3) and use it to render the border quadrilaterals $(q_i q_{i+1} q'_{i+1} q'_i)$. This yields the soft border effect (Fig. 5 b) which looks very much like the soft edges of the inner skeleton rendering (Fig. 5 a). Compare also Fig. 6 c (drawn with the new method and using about 80% less contour point samples) with Fig. 6 b (drawn with radial splats).

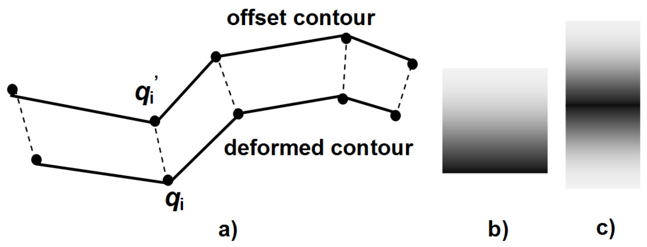


Figure 7: Soft border splatting for outer skeletons

By controlling the various splatting parameters, we obtain visual effects useful for different user scenarios. If we want to draw 'hard' AOIs with a sharp, precise, border, we set $k < 1$ for the texture profile (e.g. $k = 0.3$, Fig. 9 a). 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. $k = 5$, Fig. 9 b). This is useful e.g. to show less important properties, which should not distract the eye from the more important diagram drawing, or metrics having a low confidence value. If we want to draw tight areas, we use a low k_R value (e.g. 0.1) in inner skeleton mode, or more deformation iterations (e.g. 10..20) in outer skeleton mode. Conversely, if we want looser, more rounded areas, we use a high k_R value (e.g. 2) in inner skeleton mode, or fewer deformation iterations (e.g. 5) in outer skeleton mode. Clearly, many other scenarios are possible too.

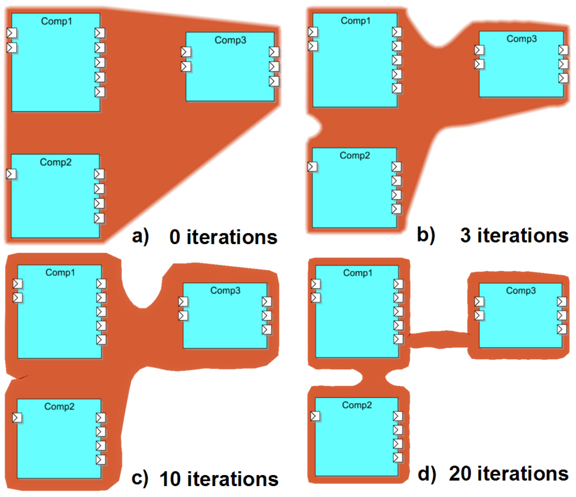


Figure 8: Controlling tightness and smoothness (outer skeletons)

A second variation our users found very intuitive and useful during our case studies (Sec. 4) was to draw AOIs as *contours* instead of filled shapes. For the inner skeletons, this is done 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. $d \in [0,1]$ controls the contour width: $d = 0$ yields the filled shapes, and $d \approx 1$ yields a very thin contour. As before, k controls the contour sharpness. Figure 9 (c,d) shows two examples of areas of interest drawn with contours with a contour width $d = 0.8$.

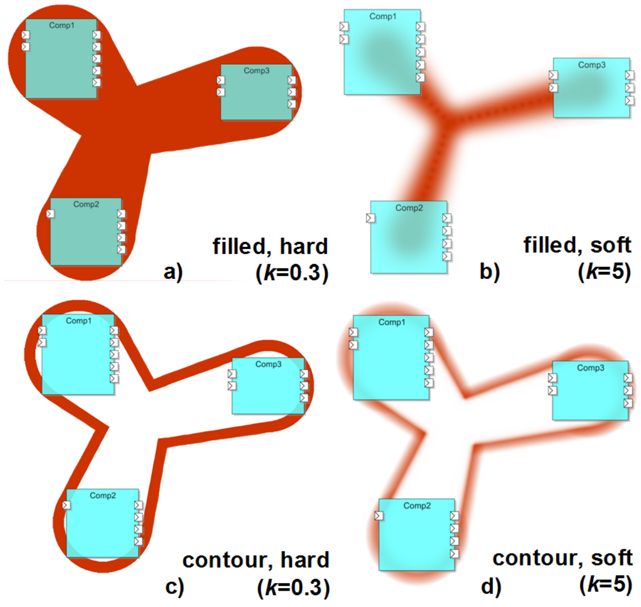


Figure 9: Filled and contoured areas (inner skeletons)

However, contour drawing using the inner skeletons has the unpleasant property that it erases the inside of the contour. This leads to undesired effects when e.g. drawing multiple, overlapping AOIs, as shown in Fig. 11 a. This problem is easily solved when drawing AOI contours using the outer skeleton (Fig. 11 b,c). We do this by simply skipping drawing the triangulation and drawing only the soft contour band, this time using a mirrored band texture (Fig. 7 c) to make the border look symmetric. As shown in Fig. 11 b, we can now easily understand which elements are in which AOI, e.g. the upper-right element is in both AOIs. After our users experimented with this display mode one some large diagrams (see Sec. 4), they required the same intuitive display of overlapping AOIs also in filled area mode, not only contour mode. We solved this request using outer skeletons by using a special blending mode, as follows. First, we render the background black. Next, we render all AOIs using $1 - RGB_i$, where RGB_i is the actual color of area i, in additive OpenGL blending mode. After all areas are rendered, we negate the image. The resulting color will be (see also Fig. 11)

$$RGB = 1 - (\max \sum_i (1 - RGB_i)) \qquad (4)$$

The above means that areas are rendered as before where they do

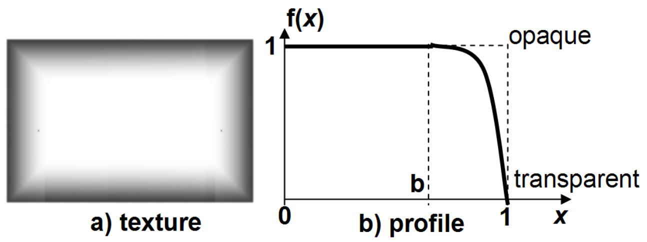
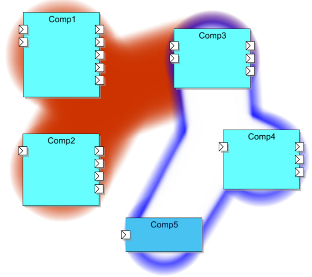
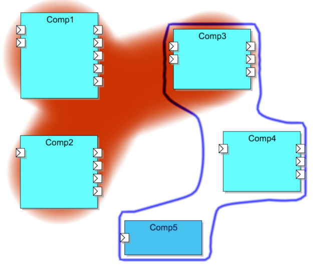
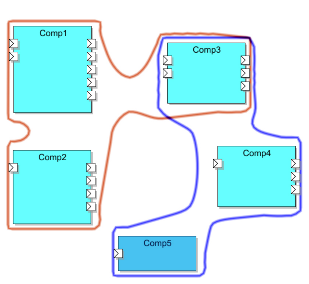


Figure 10: Eraser texture design

not overlap. Overlap regions have a color equal to the subtractive blending of the overlapping areas' colors, i.e. the darker they are, the more areas overlap there (Figs. 5 b, 11 d).


(a) Contour-area overlap problem


(b) Contour-area overlap solution


(c) Two overlapping contours

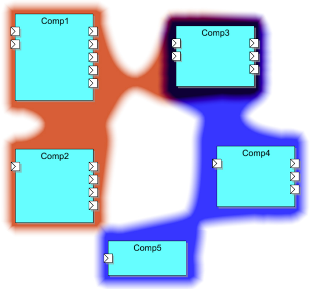

(d) Two overlapping areas

Figure 11: Drawing overlapping areas of interest

3.3 Erasing Overlapping Components

Both inner and outer skeleton drawing methods described so far guarantee that the drawn shape visually surrounds all elements in the AOI. However, the drawn shape might surround, or overlap with, elements which are not in the AOI, but close to it, e.g. the marked one in Fig. 12. This is, of course, an undesired side effect. As explained in Sec. 1, one of our hard constraints is to never modify the diagram layout. Hence, we must find some other solution to visually show that the problem elements are actually not in the AOI they visually interfer with. We solve this problem as follows. First, we draw all AOIs as described so far. Next, for all elements not in any AOI, we draw an eraser texture. This is a transparency texture, like the splat texture (Fig. 3 a) used to draw the AOIs, except that it has a rectangular (instead of radial) shape (see Fig. 10 a) and a profile given by a slightly different function. Instead of $f(x) = x^k$, we use now the following profile f (see also Fig 10):

$$f(x) = \begin{cases} 1, & x < b \\ \left(\frac{x-b}{b}\right)^k, & x \geq b \end{cases} \qquad (5)$$

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 Fig. 12. The value $b = 0.8$ is a good default.

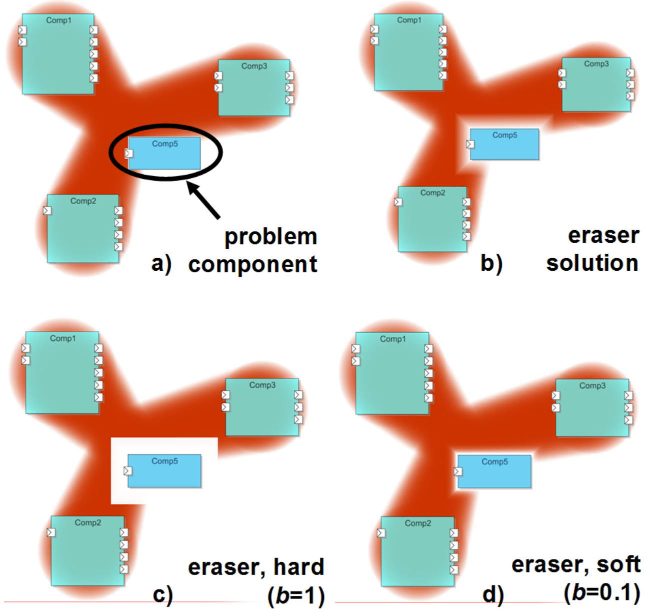


Figure 12: Erasing incorrectly overlapping elements

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 Fig. 12. The element 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 ranging from hard to soft (Fig. 12 c,d). We had a good thought about whether this erasing technique is sufficient and/or appropriate for visually marking elements as being outside of an AOI. Another possibility we investigated was to constrain the AOI shape to 'avoid' including such elements. After various experimentation, however, we found this route to be too complex, since the layout configurations met in practice on large diagrams with may AOIs are very difficult to handle in a geometric fashion. However limited at the first sight, our eraser technique *always* delivers controlled, visually expected results, which are easy to interpret for the users.

4 Applications

We present below two case studies we did to assess the effectiveness of our area of interest visualizations. In both situations, we targeted a group of industrial users that had a set of UML diagrams representing some system design. Areas of interest are defined using either metrics (Sec. 4.1) or reverse engineering analysis tools (Sec. 4.2). These data are input into our AOI visualizer, which was built atop of MetricView, an interactive UML and metric visualization tool [Termeer et al. 2005]. The users employed this tool to discuss various questions they had on the designs in their diagrams. We monitored the users' discussions and also analyzed their feedback, and used this to (iteratively) improve the design of our AOI visualization methods.

4.1 Application 1: The Car Media Center

Within the ITEA Trust4All project [ITEA 2005], a Real-Time Integration Environment (RTIE) was built. RTIE provides design and development of embedded real-time, component-based systems, e.g. e.g. mobile phones, car navigators, or set-top boxes [Bondarev et al. 2006, to appear] using the ROBOCOP component model [ITEA 2002]. RTIE provides a composer tool, for visual system assembly from components, and a quality assessment and analysis (QAA) tool, for design-time analysis and prediction of system attributes, e.g. reliability, hardware resource usage, and throughput. Our visualization reads the composer tool output (a component diagram) and QAA tool output (attributes) and visualizes the composition and areas of interest determined by the predicted attributes.

With the RTIE toolset, our users have developed a Car Media Center (CMC) real-time system with the following functionality: GPS-based car navigation, radio and digital TV reception and display, and CD/DVD playback. Figure 13 is a snapshot from the Eclipse-based composer GUI, showing the CMC system design consisting of 28 components and the connections between provided interfaces (at the left of icons) and required interfaces (at the right of icons). The CMC has a dataflow-like design. Let us briefly explain the component functions in the CMC system. The Main_UI component (1) receives user input by polling the buttons on the car 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) 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 show 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. 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.

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?

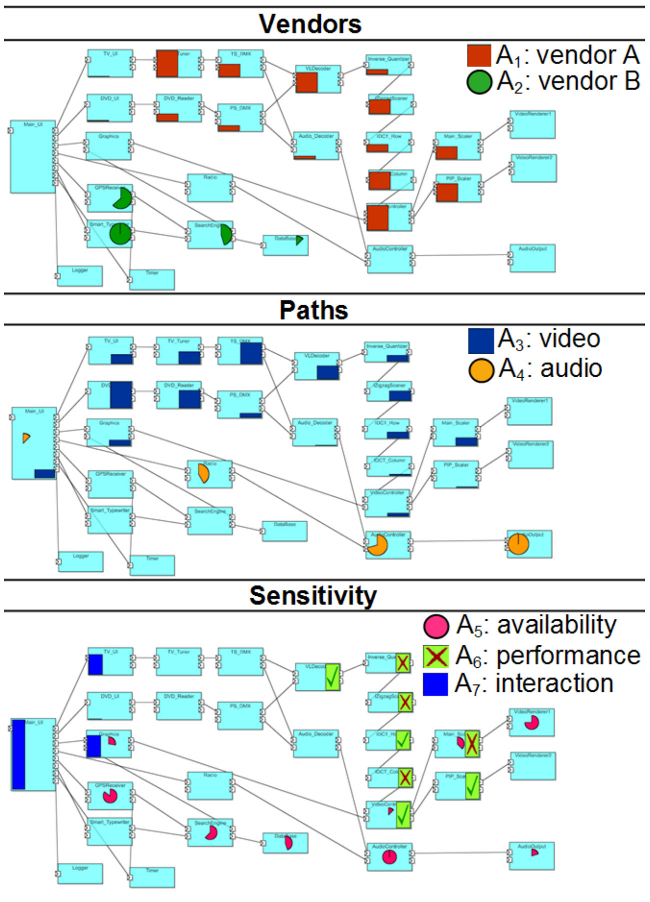


Figure 14: AOIs for the Car Media Center system (icons)

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 shown in Table 1.

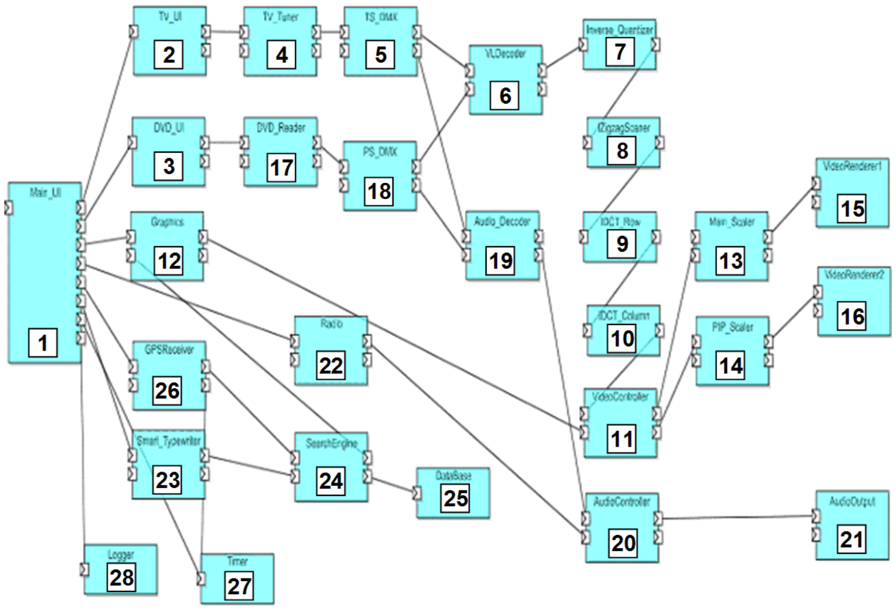


Figure 13: Car Media Center component-based architecture (snapshot from the visual composition tool)

Area	Meaning of components in the area
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

Table 1: Areas of interest for the CMC architecture

Our users first tried to visualize these areas of interest (AOIs) using the standard metric icons provided by MetricView [Termeer et al. 2005], by assigning different 'marker' icon shapes and colors to every AOI. Components in one AOI thus share the same marker shape and color, which are chosen in some way so that they look different for different area. Figure 14 shows the result. As expected, this visualization is not very easy to follow. Next, the users tried to visualize the same AOIs, this time using AOIs rendered with the original inner-skeleton-based splatting method [Byelas and Telea 2006]. Figure 15 shows the result, which uses the same area colors as for the markers in Fig. 14. The AOIs 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). Studying the component functions, described earlier, we concluded that vendor B (A_2) provided all the navigation (GPS)-related components.

Still, this visualization has some problems. In the 'Paths' view (Fig. 15 middle) it is not quite clear whether the leftmost component is in both the video (A_3) and audio (A_4) areas. Also, in the 'Sensitivity' view (Fig. 15 bottom), it is not clear how the availability (A_5) and performance (A_6) areas overlap exactly. Moreover, the star-shaped form of the AOIs is somehow visually distracting. It suggested (at least to one user) there is something special about the element(s) located at the star center, which is of course not the case. We used next our new outer-skeleton based rendering method (Fig. 15 b). In Fig. 15 b (middle), we see a dark area around component 1 (leftmost). This says, as explained in Sec. 3.2, it is in two areas (i.e. video and audio). The advantage of the new visualization is even clearer comparing Fig. 15 b (bottom) with Fig. 15 a (bottom). The dark areas show now easily the overlap of A_5 (availability) with A_6 (availability) and A_7 (performance). Comparing the 'Sensitivity' with the 'Vendors' and 'Paths' views answers further questions. We see that only 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 video component '11' which is from vendor A. Finally, we locate three interesting components (VideoController, MainScaler and PiP_Scaler, i.e. 11,13, and 14 in Fig. 15) which are both performance and availability-sensitive.

Finally, we mention that we can show diagrams, metric icons, and areas of interest together in a single view, if desired. Figure 16 illustrates this with a snapshot from the actual visualization tool. Clearly, such images can become overcrowded. We solve this by the original MetricView idea [Termeer et al. 2005], i.e. let users adjust the transparencies of diagrams, metrics, and AOIs separately. In typical scenarios, users would make the diagram fully opaque (salient) and use a 50% transparency for either metrics or AOIs.

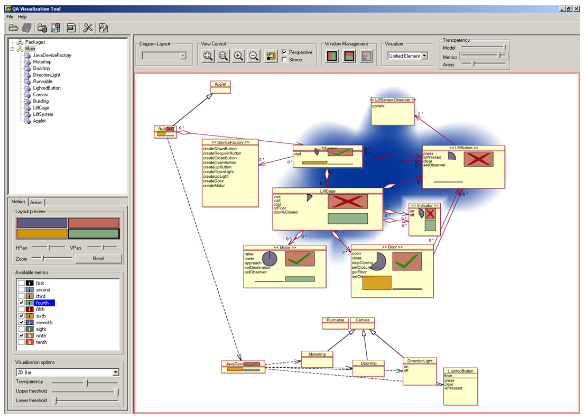
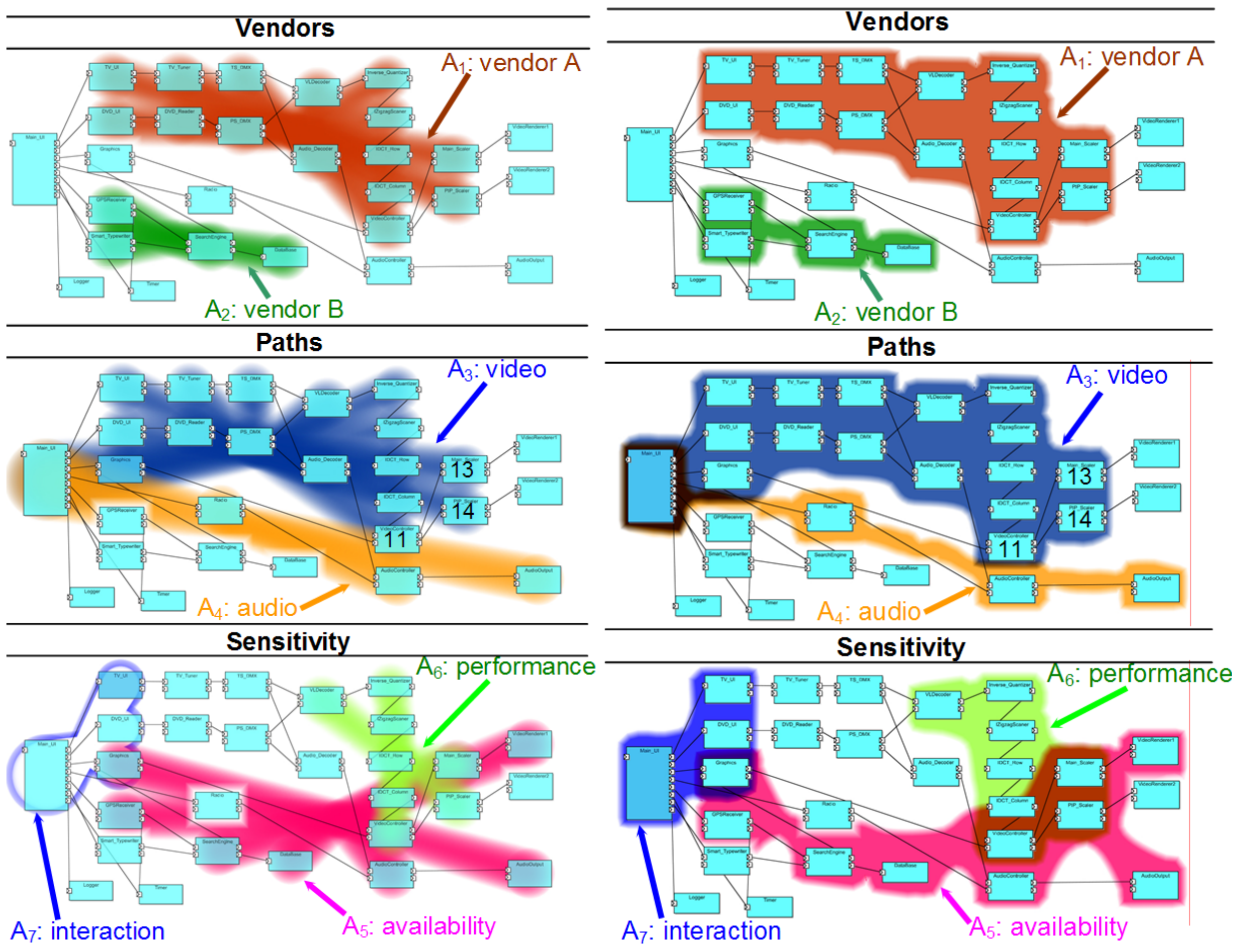


Figure 16: Combining diagrams, metrics and AOIs



(a) inner skeleton rendering (b) outer skeleton rendering

Figure 15: AOIs for the Car Media Center system. Compared to Fig. 14, areas are easily visible

4.2 Application 2: Reverse Engineered UML

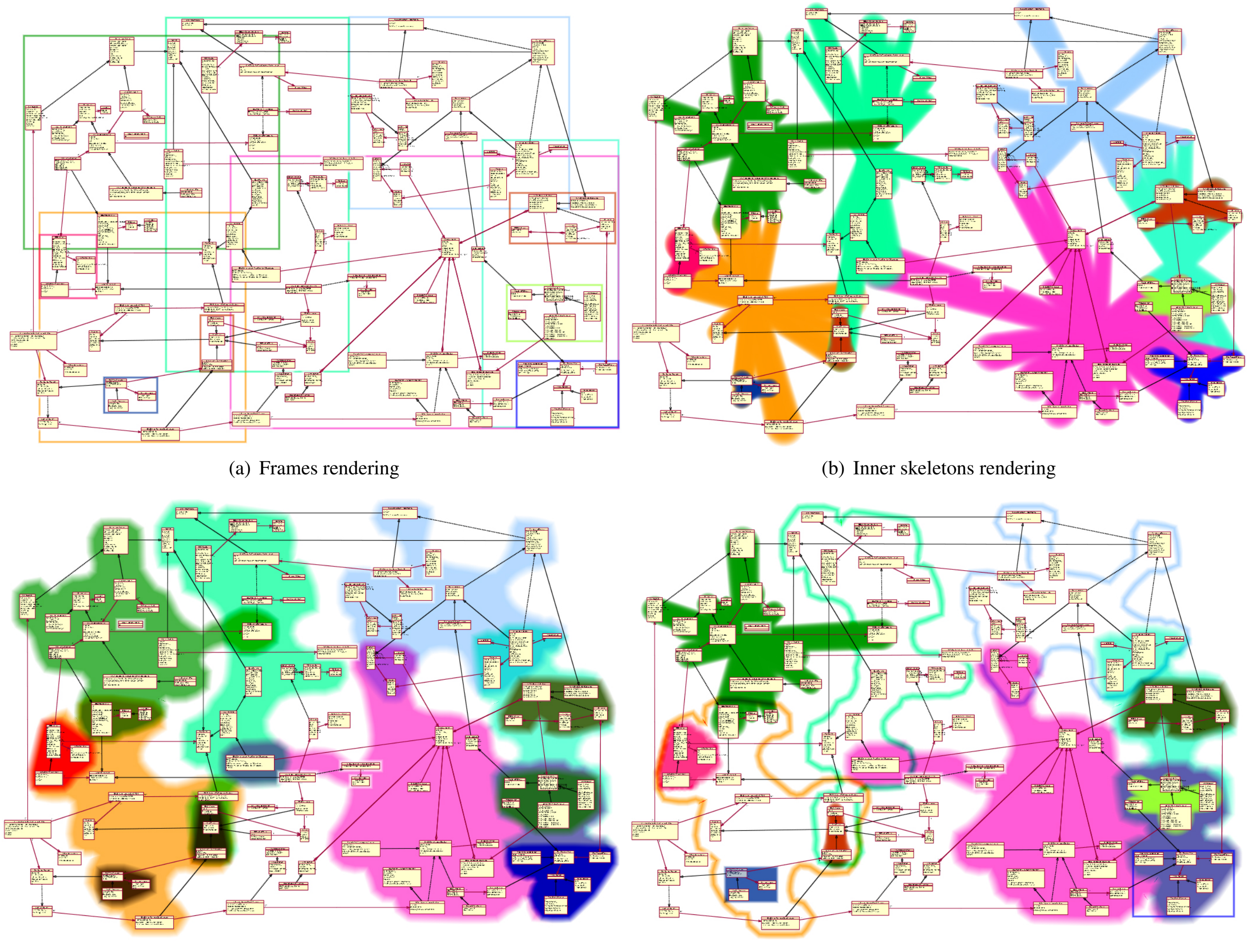
The users for our second application work in a laboratory for quality of software [LaQuSo 2006]. They extracted UML from industrial C++ code, using the Columbus tool [Ferenc et al. 2002], and identified several high-level design elements in terms of groups of classes. They used our AOI visualizations as a medium to see and discuss their results during this reverse engineering process. First, they tried to show AOIs by icon marking (similar to Fig. 14). The results were poor, since their diagrams are quite large. Next, they tried to display AOIs using rectangular frames (Fig. 17 a). Again, the result is hard to understand, since there is a lot of visual cluttering. They tried our improved outer skeleton method (Fig. 17 c). Finally, they settled for a mixed rendering (Fig. 17 d), containing inner skeletons (e.g. upper-left), outer skeletons (vast majority), and even rectangular frames (lower-left). Clearly, the three rendering techniques can nicely co-exist in the same visualization. Although this use case has 12 areas of interest on a quite complex diagram, the users could construct a 'meaningful and useful' (in their own words) visualization in under a few minutes, using basically the default parameter settings. The only settings they actually wished to change were the hues for each area, and whether an area is to be drawn as full or contour. One aspect the users were interested in was finding out whether certain design principles, such as nesting of subsystems, were violated or not. This can be easily checked in our visualization, as subsystem nesting corresponds one-to-one to visual AOI nesting.

We do not detail here the actual application-specific meanings of the AOIs in this UML diagram (in this case, an industrial controller), given the limited space. However, we believe the readers themselves can see the usefulness of visualizations such as Fig. 17(right) for their own diagrams and scenarios.

5 Discussion

The technical description of the AOI rendering in Sec. 3.2 involves many parameters, which may suggest that users must tune many values in the actual tool to get useful visualizations. This is luckily not the case. We mentioned all these parameters just to make the explanation of our technique detailed and complete. In the actual 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 Section 4 took just a few minutes for users already familiar with UML editors.

The main contribution of this paper is the outer skeleton technique for rendering AOIs. In practice, this technique minimizes unnec-


(a) Frames rendering

(b) Inner skeletons rendering

(c) Outer skeletons rendering

(d) Mixed rendering (frames, inner skeletons, outer skeletons)

Figure 17: Reverse-engineered UML diagram with 12 AOIs, various rendering modes. (c,d) are our favorites.

essary visual overlap of the AOIs as desired, in almost all cases, even for complex diagram and area layouts. Hence, the question arises whether the original inner skeleton technique is still useful. Our answer is positive. We noticed that, for small or middle-sized AOIs having a convex hull close to a regular *n*-sided polygon, inner-skeleton AOI rendering produces more rounded, more 'natural' shapes than the outer skeleton technique. Geometrically, there is an interesting connection between the inner and outer skeletons. For such diagrams and a very fine sampling rate δ and large number of deformation iterations (see Sec. 3.1, the outer skeleton converges to the inner skeleton. Since the contour points move inwards with unit normal speed, until they get very close to points coming from other locations of the contour, the outer skeleton converges to the so-called medial axis of the convex hull [Telea and van Wijk 2002]. This result is interesting, since it predicts the kind of shapes our method will produce.

The outer-skeleton AOI renderer was written in OpenGL in under 500 lines of C++ and added to the MetricView tool [Termeer et al. 2005]. Adding AOI rendering to other tools, e.g. Rational Rose [IBM 2005] or Together [Borland 2005], should be very easy, once one has access to the tool renderer code. Rendering an AOI involves drawing a few hundred transparency textures, which OpenGL can do in real-time. This enables users to interactively edit component diagrams, e.g. by dragging element icons in the tool GUI, while AOIs are re-rendered on-the-fly. Compared to the significantly more complex implementation, delicate parameter setting, and less robust visual behavior of algorithms like H-BLOB [Sprenger et al. 2000], we can say our method is indeed an effective and efficient way to visualize areas of interest.

A most recent version of our tool is downloadable from `http://www.win.tue.nl/∼alext/ARCHIVIEW`

6 Conclusions

We have presented a technique that adds areas of interest (AOIs) to the rendering of classical UML-like diagrams, based on splatting hald-transparent textures on a geometric skeleton computed from the diagram layout. We borrowed the texture splatting principle from an existing work on AOI rendering [Byelas and Telea 2006]. Our contribution is as follows. We added a completely new 'outer skeleton' concept which visibly improves upon the original 'inner skeleton' from [Byelas and Telea 2006], by reducing visual clutter and making regions of overlapping areas of interest better visible. We proposed a new splatting scheme, using band textures, to complement our new outer skeleton shapes, yielding the same type of amooth results as the radial splat texture did in combination with the inner skeletons. We designed a blending mode that combines colors in regions where several areas of interest overlap, in order to make such regions easily visible. Finally, we applied the complete design to UML class diagrams, whereas [Byelas and Telea 2006] targeted only component diagrams.

Throughout our visual design, 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 metric and diagram specification are separate, we can define and/or change any number of AOIs per user scenario without touching the diagram data and/or its XMI specification format. We conducted two experiments where actual system architects and analysts used our AOI visualizations as a means to explore and exchange ideas about the system represented by the diagrams. In this sense, the soft-shaped, AOI images can be seen as exploration-time annotations showing fuzzy concerns atop of the crisply drawn, ground-truth UML diagrams.

As future work, we are investigating ways to parameterize the AOI rendering (e.g. color, softness, and texture) by actual metric values, to display such metric values for whole diagram element sets. Moreover, we are looking at ways to visualize the temporal dimension, i.e. changing areas of interest atop of changing diagrams.

Acknowledgments

This research is part of the ITEA project Trust4All, which develops 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 [ITEA 2005]

References

ARYA, S., MOUNT, D., NETANYAHU, N., SILVERMAN, R., AND WU, Y. 1998. An optimal algorithm for approximate nearest neighbor searching. *J. of the ACM 45*, 891–923. `www.cs.umd.edu/mount/ANN`.

BALZER, M., AND DEUSSEN, O. 2005. Exploring relations within software systems using treemap enhanced hierarchical graphs. In *Proc. VISSOFT*, IEEE Press, 89–94.

BONDAREV, E., CHAUDRON, M., AND DE WITH, P. 2006, to appear. A process for resolving performance trade-offs in component-based architectures. In *Proc. 9th Intl. Symposium of Component-Based Software Engineering*, Springer LNCS.

BORLAND. 2005. Together. `www.borland.com/together`.

BYELAS, H., AND TELEA, A. 2006. Visualization of areas of interest in component-based architectures. In *Proc. EUROMICRO SEAA - Component-Based Software Engineering*, IEEE Press. `www.win.tue.nl/alext/ALEX/PAPERS/EUROMICRO06/paper.pdf`.

COSTA, L., AND CESAR, R. 2001. *Shape Analysis and Classification: Theory and Practice*. CRC Press.

DUNKE, R., AND SCHMIETENDORF, A. 2000. Possibilities of the description and evaluation of software components. *Metrics News 5*.

FENTON, N., AND PFLEEGER, S. 1998. *Software Metrics: A Rigorous and Pracical Approach*. Chapman & Hall.

FERENC, R., BESZÉDES, A., TARKIAINEN, M., AND GYIMÓTHY, T. 2002. Columbus – reverse engineering tool and schema for c++. In *Proc. ICSM*, IEEE Press, 172–181.

GANSNER, E., AND NORTH, S. 2000. An open graph visualization system and its applications to software engineering. *Software: Practice & Experience 30*, 11, 1203–1233.

GILL, N., AND GROVER, P. 2003. Component-based measurement: A few useful guidelines. *ACM SIGSOFT Software Engineering Notes 28*.

GOULAO, M., AND ABREU, F. 2004. Formalizing metrics for *COTS*. In *Proc. MPEC'04*, Edimburgh.

IBM. 2005. *Rational Rose*. `www.306.ibm.com/software/rational`.

ITEA. 2002. *ROBOCOP*: A robust open component-based software architecture for configurable devices. Public document, version 1.0. available at `www.hitech-projects.com/euprojects/robocop`.

ITEA. 2005. *Trust4All* project. `www.win.tue.nl/trust4all`.

LAQUSO. 2006. Laboratory for quality of software. Eindhoven University of Technology, the Netherlands. `www.laquso.com`.

MARCUS, A., FEND, L., AND MALETIC, J. I. 2003. 3d representations for software visualization. In *Proc. ACM SoftVis*, 27–36.

MÖLLER, A., AKERHOLM, M., FEDERIKSSON, J., AND NOLIN, M. 2004. Evaluation of component technologies with respect to industrial requirements. In *Proc. EUROMICRO'04*, IEEE Press, 56–63.

RILLING, J., AND MUDUR, S. P. 2002. On the use of metaballs to visually map code structures and analysis results onto 3d space. In *Proc. WCRE*, IEEE Press, 299–306.

SHEWCHUK, J. R. 1996. Triangle: Engineering a 2d quality mesh generator and delaunay triangulator. In *Proc. Applied Computational Geometry*, ACM Press, 124–133.

SPRENGER, T., BRUNELLA, R., AND GROSS, M. 2000. H-blob: A hierarchical clustering method using implicit surfaces. In *Proc. Visualization*, IEEE Press, 61–68.

TAUBIN, G. 2000. Geometric signal processing on polygonal meshes. In *EUROGRAPHICS STAR Reports*.

TELEA, A., AND VAN WIJK, J. J. 2002. An augmented fast marching method for computing skeletons and centerlines. In *Proc. VisSym*, IEEE Press, 151–158.

TERMEER, M., LANGE, C., TELEA, A., AND CHAUDRON, M. 2005. Visual exploration of combined architectural and metric information. In *Proc. VISSOFT*, IEEE Press, 21–26.

TILLEY, S., WONG, K., STOREY, M., AND MÜLLER, H. 1994. Programmable reverse engineering. *Intl. J. Software Engineering and Knowledge Engineering 4*, 4, 501–520.

VOINEA, L., AND TELEA, A. 2004. A framework for interactive visualization of component-based software. In *Proc. EUROMICRO SEAA - Component-Based Software Engineering*, IEEE Press, 567–574.

WOO, M., NEIDER, J., DAVIS, T., AND SHREINER, D. 2001. *OpenGL Programming Guide, 3rd edition*. Addison-Wesley.

WUST, J. 2005. *SDMetrics*: The software design metrics tool for *UML*. `www.sdmetrics.com`.