

# Using Multilevel Call Matrices in Large Software Projects

Frank van Ham

Department of Mathematics and Computer Science  
Technische Universiteit Eindhoven  
The Netherlands  
e-mail:fvham@win.tue.nl

## Abstract

Traditionally, node link diagrams are the prime choice when it comes to visualizing software architectures. However, node link diagrams often fall short when used to visualize large graph structures. In this paper we investigate the use of call matrices as visual aids in the management of large software projects. We argue that call matrices have a number of advantages over traditional node link diagrams when the main object of interest is the link instead of the node. Matrix visualizations can provide stable and crisp layouts of large graphs and are inherently well suited for large multilevel visualizations because of their recursive structure. We discuss a number of visualization issues, using a very large software project currently under development at Philips Medical Systems as a running example.

**CR Categories:** H.5.2 [Information Systems and Presentation]: User Interfaces—Graphical User Interfaces (GUI) D.2.4 [Software Engineering]: Software/Program Verification—Validation;

**Keywords:** software visualization, multilevel visualization, call matrix

## 1 Introduction

Visualization can play an important role during the complete life-cycle of software systems, from the global design of a new system to the reverse engineering of legacy systems. Diagrams are a standard aid during the design of a system. In the maintenance of large legacy systems, for which no queryable design information exists or for which the design information is outdated, visualization can provide more insight into the structure of the system by aiding in the detection of patterns or features.

We focus on the use of visualization for the stage in between, where a high-level architectural design is transformed into an implementation. In large systems, the high-level architecture of a system is designed by a small team of system architects, and implemented by a substantially larger team of programmers. A first problem is that it is impossible and often not even desired for the architects to manage, or even specify, every little implementation detail, due to the size of the system. This might lead to subsystems being implemented with an interface that is too wide, subsystems

growing too big or calls being made between subsystems that are not supposed to communicate, for example. Determining these potential problems as early in the development process as possible can save much time later on.

A second problem when dealing with very large software systems is that even the top architects have difficulties to maintain a birds eye view of the entire architecture. Although they often have general knowledge of most subsystems, knowing exactly how all pieces fit together can be a daunting task. In this case software visualization can also help, serving as a useful memory externalization and communication tool.

We use a very large software development project currently in progress at Philips Medical Systems as a running example. The project deals with the redesign of a medical imaging platform. To indicate the scale of the project: It is one of the largest software engineering efforts ever undertaken by Philips, consisting of well over 25,000 classes (including instantiated template classes), implemented in over 3 million lines of code, with a comparable amount of testing code. Approximately 300 programmers and 12 architects have been working on the project since 1997. Architects working on the system were interested in the following aspects:

- To what degree does the current implementation conform to the architecture specification? Are there any calls made that are not allowed?
- Which subsystems are affected if a subsystem is changed? Or, in other words, which subsystems call or are called by the changed subsystem?
- What are the changes in the system over time?

The next sections describe a visualization tool that is able to deal with the scale of the project and can provide answers to the questions stated above. Section 2 describes the problem and the approach we took, section 3 covers related work and section 4 deals with some visualization issues. We discuss practical results in section 5 and conclude in section 6.

## 2 Approach

The system we are considering is hierarchically decomposed into different components. Every component  $x$  has an integer abstraction level  $A(x)$ . Given a total of  $N$  levels, 0 is the highest possible level of abstraction and  $N - 1$  the lowest. The system under consideration has five different abstraction levels named System, Layer, Unit, Module and Class. Every component  $x$  is contained in exactly one other component  $P(x)$ , unless  $A(x) = 0$ . Components also call each other: predicate  $C(x, y)$  is true if a call exists from component  $x$  to component  $y$  and false otherwise. Note that calls are directed so  $C(x, y) \neq C(y, x)$ .

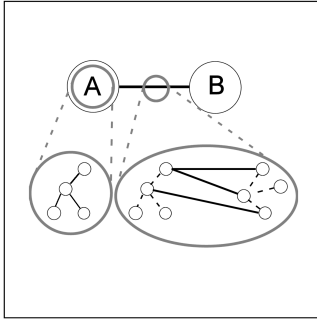


Figure 1: When displaying all calls made between nodes A and B, we have to zoom both A and B

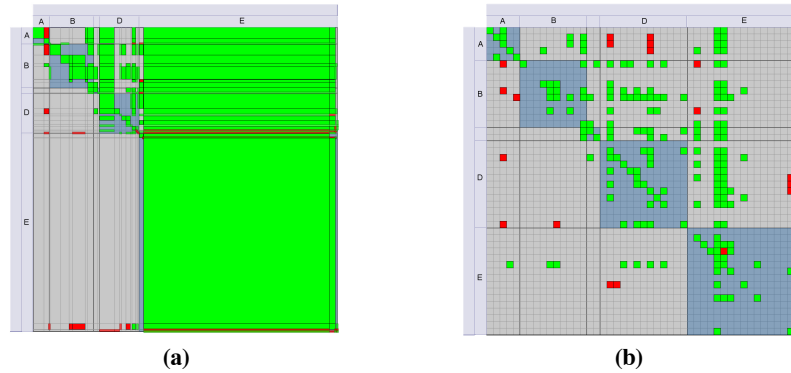


Figure 2: Recursive matrix subdivision according to relative component size (a), Matrix subdivision according to number of subcomponents at a specific abstraction level (b)

The software architecture under consideration here has the following properties:

- **Strictly hierarchical.** Components form a tree structure in which  $A(P(x)) = A(x) - 1$  for every component  $x$  with  $A(x) > 0$ .
- **Layered calls.** Calls can only be made between two components at the same level in the hierarchy. Or equivalently:  $C(x,y) \Rightarrow A(x) = A(y)$
- **Call aggregation.** Calls are aggregated upward through the hierarchy. For every call between two components, there exists a call between their respective parent components. That is:  $C(x,y) \Rightarrow C(P(x),P(y))$ .

The actual call data and composition hierarchy are extracted from the C++ code by a system developed at Philips Research Laboratories [Postma 2003]. It consists of a number of PERL scripts and takes about 7 hours to extract data. Calls that are not interesting from a software engineering perspective, such as callback interfaces, Microsoft Foundation Classes calls or standard C++ calls, are filtered from the data. Data extraction was performed at set intervals (usually monthly) leading to a series of datasets.

Providing a meaningful, multi-level, stable and interactive layout is by no means an easy task. Traditionally, system architects are quite fond of *node-oriented* visualizations, such as the node-link diagram. In these visualizations, the node (or component in this case) and its characteristics are the prime objects of interest. Unfortunately, node-link diagrams usually cannot provide meaningful and readable layouts for more than a hundred nodes at a time.

Node-link diagrams also have a problem with multilevel viewing. This is inherent to their visual language. Consider a hierarchical call graph, such as the one defined above. Components are displayed as nodes, calls between components as edges between nodes. If we want to view all calls that are made within component A, we can suffice with displaying only A at a lower level of abstraction. If however, we want to view all calls that are made between two different components A and B, we have to show both A and B at a lower abstraction level (See figure 1).

Finally, node link diagrams are not always as stable as we would like. Addition or removal of a single node can lead to a radically different layout. Although methods to alleviate this problem exist, a stable layout cannot be guaranteed. Because of these issues, node link diagrams are not very suitable for the purpose discussed here.

### 3 Related work

In this paper we opt for a *link-oriented* visualization and the use of call matrices. Call and, more general, adjacency matrices have been used before in graph visualization as an alternative to the traditional node link diagrams in [Abello and Krishnan 1999; Abello and Korn 2002; Becker et al. 1995; Stolte et al. 2002; Ziegler et al. 2002] amongst others. More specifically, inter-class call matrices have been proposed [de Pauw et al. 1998] to visualize calls between different classes of a relatively small object oriented system. Colors represented the frequency of calls between different classes. The user could also bring up a detailed view in a separate display, showing calls between different methods in a class. We expanded on this idea by using multiple levels in the hierarchy, allowing us to apply it to much larger samples. The different views have also been integrated by providing a smooth zoom transition between them, similar to the SHriMP [Storey et al. 2001] interface. This gives the impression of a single coherent information space.

### 4 Visualization

A call matrix is an  $M \times N$  matrix in which each row and column represents a subcomponent. A cell  $[i, j]$  in the matrix represents a call from component  $i$  to component  $j$ . If such a call exists a cell is filled, else it is empty. From a visualization viewpoint, a hierarchical call matrix has the following desirable properties:

- **Uniform visual representation:** The only visual element used in the visualization is a matrix-cell. This allows us to avoid the previously mentioned zooming inconsistency. Zooming into a node A corresponds to zooming in on the cell  $[A,A]$ , zooming into edges from node A to node B corresponds to zooming in on cell  $[A,B]$ .
- **Recursive structure:** The visual representation of call  $C(i, j)$  is by definition contained in the visual representation of call  $C(P(i),P(j))$ . This makes it very easy to construct multiscale visualizations.
- **Stability and predictability:** Contrary to other popular graph visualization methods (such as force directed methods), addition of a call is guaranteed to never radically change the layout, since every call has its own prerreserved section of visualization space.

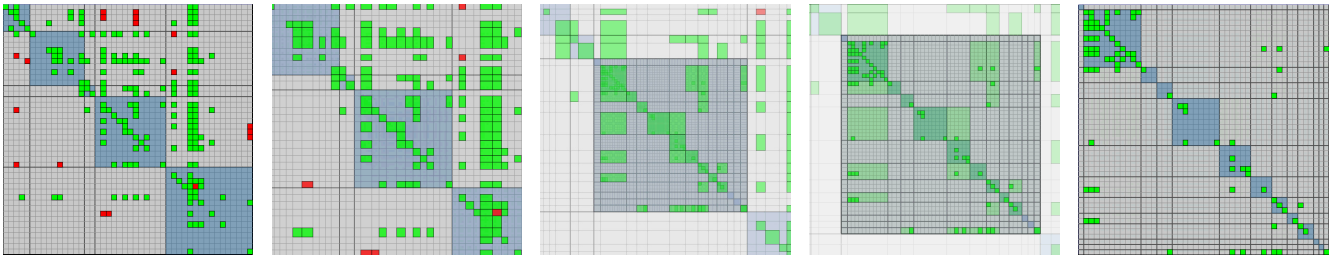


Figure 4: Series of frames showing the transition from the cell in figure 2b to the cell in figure 3

The construction of a matrix visualization is not difficult as such, but a careful design is required for an optimal result. In this section we will elaborate on some of the subtleties in the construction of a hierarchical call matrix.

#### 4.1 Matrix subdivision

Each cell  $M[i, j]$  in the matrix represents the calls between components  $i$  and  $j$ . Since it would be very practical if the dimensions of  $M[i, j]$  depend on the sizes of components  $i$  and  $j$ , it makes sense to first define the size of a component. The most obvious choice here is to recursively define a component's size by summing the sizes of its subcomponents. For components at the lowest level of abstraction, which don't have any subcomponents, one can take a suitable size statistic such as the number of source lines or number of class methods. We could then try to visualize all abstraction levels in a single display, leading to a treemap-like [Johnson and Shneiderman 1991] recursive subdivision. Unfortunately, this means we have to represent almost 25,000 items at the lowest level, which is well above the limit of modern day raster displays. We therefore limit the number of abstraction levels displayed simultaneously. In practice displaying two abstraction levels at once is a nice compromise between showing detail and maintaining overview. This leads to a visualization as displayed in figure 2a. Large subcomponents take up an area of visualization space that is the square of their size suppressing small but possibly important components. The wide variety in row and column sizes (ranging from a few pixels to almost half of the visualization space) also makes the resulting matrix messy and hard to 'read'.

Since we are displaying only a limited number of abstraction levels at a time, a much better option is to use the number of currently visible subcomponents as a size measure instead. Figure 2b shows a visualization using this size measure, resulting in a much more regular, grid-like subdivision of the visualization space. One of the disadvantages of using this measure is that zooming into a matrix cell distorts the sizes of its subcells, since the number of visible subcomponents increases non uniformly. This has as a consequence that matrix cells may look different depending on the abstraction level they are viewed on (Figure 3). A second disadvantage is that the

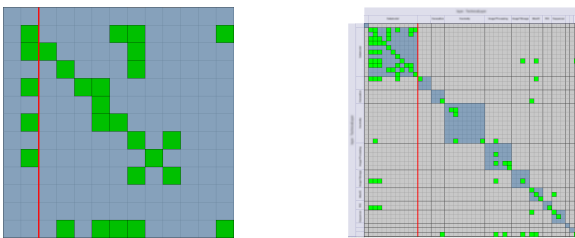


Figure 3: Identical matrix cells at different levels of visual abstraction

displayed size of a component is no longer proportional to the size of that component relative to the system. This can clearly be seen in Figure 2, where component  $E$  (note that actual, more descriptive component names have been removed for security reasons) takes up well over half of the system in figure 2a, and roughly one third of the system in figure 2b. Both these problems can be solved by applying appropriate visualization techniques, as we shall see in the following sections.

#### 4.2 Multilevel zooming

A common problem in multilevel visualizations is that users are provided with different levels of visual abstraction, without immediately comprehending how these different levels interrelate. This problem is aggravated by the fact that the visual representation of a single matrix cell depends on the current level of data abstraction. To illustrate this, figure 3 shows a cell from figure 2b displayed at different levels of visual abstraction. If nothing were done to remedy this, users would quickly lose context when navigating the structure. This leads to information fragmentation instead of a consistent mental map. The use of a smooth, continuous semantic zoom from one abstraction level to another can be of great help in this case [Stolte et al. 2002; Stasko 1998].

The best results were obtained with a combination of linear interpolation and crossfade. Interpolation can be done in a straightforward fashion by animating every gridline in the high level representation to their position in a lower level representation. Figure 3 shows the positions of a single gridline (red) at two different levels of abstraction. The final effect is that grid cells expand or contract, depending on the number of subcomponents they contain. At the same time we perform a crossfade (that is, the transparency of one representation is increased, while the other representations transparency is decreased) to avoid visual elements suddenly appearing out of nowhere. Figure 4 shows a series of frames from a zoom operation, a full animation can be viewed at [van Ham 2003]. We can use the same procedure for panning operations. Additionally, independent zooms into one axis of the matrix are supported in a similar manner. Users can indicate they want to zoom in on only one dimension, for example by clicking a column header, and the system expands that particular column.

Zoom/pan trajectories are computed using a novel method [van Wijk and Nuij 2003]. A smooth and efficient path of the virtual camera is derived analytically, such that the perceived velocity of the moving image is constant. Apart from semantic zooming, the visualization system also supports traditional geometric zooming into sections of a matrix cell, which comes in handy since many parts of the matrix are very sparsely populated. Since it is possible that some matrix cells occupy less than a pixel on screen (depending on the geometric zoom of the user), any cell containing a call is at least rendered at a preset minimum size. This avoids user having to hunt for cells containing calls.

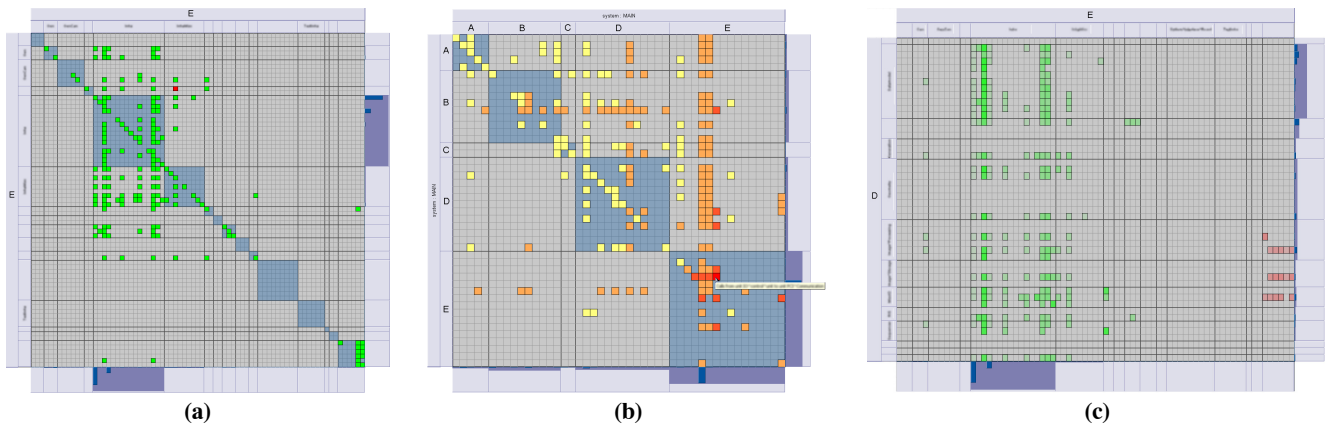


Figure 5: Displaying additional data: call permissions (a), call neighborhood (red calls are closer to call under pointer) and call density (c)

### 4.3 Multidimensional Information

Besides their suitability for multilevel visualizations, another advantage of matrix-type visualizations is that they offer sufficient space to display additional information. In a standard node-link view, edges are usually displayed as thin lines of varying length and orientation. This makes picking, coloring and labelling more difficult. A matrix visualization has the advantage of a uniform edge representation and four matrix edges on which we can display node attributes. In general, two of the four edges are used to display node names, and the two edges remaining can be used to visualize attributes such as subsystem size, number of changes or a number of system metrics such as (inverse) coupling, complexity etc. Figure 5a shows an example in which we used hierarchical histograms on the right and bottom matrix edges to visualize the size distribution within the largest layer of our system. One unit takes up over 95% of this subsystem (see also Fig 2a) with the major bulk of this unit being formed by only two modules.

Edge attributes can be visualized by means of color or transparency, other cues such as shading or texture could also be considered. We visualized a number of attributes in figure 5. Figure 5a uses color to indicate whether a call was allowed according to the architect specifications. This information is of prime interest to the architect. Figure 5b uses a color scale to indicate the local neighborhood of a call. Calls that have a shorter path-distance to the call under consideration are indicated in red. Finally, in figure 5c transparency is used to indicate the call density of a matrix cell. The call density of a cell is higher when a larger percentage of the subcells in that cell contain calls. Other attributes that are more interesting from a software engineering perspective, such as the creation time of a call in the development process or the person responsible for that specific call, could also be shown, but unfortunately these were unavailable in the system under consideration.

## 5 Results

Although the use of a matrix representation for the display of graphs might seem awkward at first, one can easily identify general system characteristics with some experience. For example, library-type high level components that receive a lot of incoming calls, but only make a few outgoing calls themselves can be identified by scanning for vertical columns of calls. The same goes for controller components, which make a lot of outgoing calls but receive fairly little

incoming calls. This type of behavior can be identified at different levels of abstraction: Layer *E* (rightmost major column in figure 5b) is a typical library layer, and within this layer there are two units acting as an interface (large orange vertical columns in figure 5b). If we zoom in on calls to this specific layer we can identify a number of unit subcomponents acting as interfaces within these units. Figure 5c shows such a zoomed section.

One of the first practical results of the system was the observation that the (list oriented) information the architects previously used as a system summary was incomplete. A large number of low-level inter-class calls were not reflected in calls at higher abstraction levels due to errors in the data extraction process. Since in this visualization every call has its own place, which means that the lack of a call somewhere can also be spotted, a number of very sparsely populated submatrices immediately drew the attention. With respect to the questions the architects wanted to be answered, the following observations were made:

**Spotting unwanted calls** When using the system, architects performed a quick scan of the system at the top abstraction level, and zoomed in on calls that they could not immediately understand (generally lone calls in an otherwise empty submatrix, such as the ones in the bottom left corner of figure 5b) or calls that were marked as 'not allowed'. After one or two semantic zooms they could identify the function of a call by the component naming conventions they used and decided whether a call was wanted or unwanted. Calls that are initially considered as 'not allowed' by the system, can interactively be set to 'allowed'. This leads to a steady reduction in the amount of unwanted connections between subsystems, as these are either removed from the code (in case of an implementation fault) or removed from the set of unwanted connections (in case the architects did not foresee this call).

**Determining component dependencies** Spotting component interdependencies is easy if we take only direct dependencies into account. Architects can simply check a single row or column in the matrix to find all components calling or called by the current component. These components deserve special attention when changes are made to the component under consideration. Detecting indirect dependencies in a matrix visualization is much harder however, since there is no preservation of the closeness criterium: calls that are structurally close, such as calls  $C(a,b)$  and  $C(b,c)$  for example, can visually be located far apart. This can be partially countered by indicating the neighborhood of a call upon selection or by displaying the  $n$ -step closure of relation  $C(a,b)$ . Both are not very satisfying solutions however. In fact, invalidation of the 'semantic

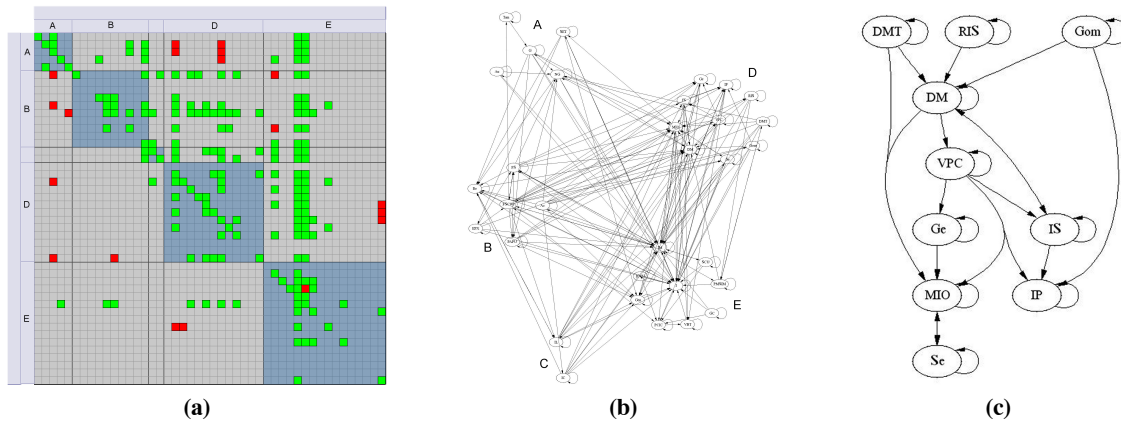


Figure 6: Matrix representation versus node-link diagrams: callmatrix of entire system (a), force directed layout of entire system (b) and layered layout of subsystem in figure 3 (c)

closeness equals visual closeness' principle is probably the main argument against matrix type representations. However, also traditional graph layout methods do not always succeed in maintaining this principle, given the large number of long edges often present in layouts of complex graphs.

**Stability** Matrix representations present a stable, regular and clean layout of complex interrelations. The addition of a small number of new calls or new components does not significantly change the resulting visualization. Architects can make snapshots of their system at regular intervals and easily track the evolution of their system. Stability also aids in the planning of a project: dependencies that are to be implemented in the future could be indicated in a different color. The same goes for sections of the system that have already been tested. The main advantage of a stable layout however, is the fact that architects are presented with a consistent visual image of their design. Mentally fragmented information is aggregated in a single image, aiding in the creation of a mental map of the software system as a whole.

Finally, we compare a matrix representation with traditional node link diagrams generated with GraphViz [Gansner and North 1999] in figure 6. Figure 6b shows a force directed layout of the subcomponents of the five top level layers. The layout was slightly modified to reflect the cluster structure. The large number of long edges between subsystems obscure a lot of interesting information that can be extracted using the matrix type visualization in figure 6a. For example the fact that layer *E* is called much more than it calls other layers is completely obscured in figure 6b. The same goes for the fact that *D* mainly calls *E* while calls to other layers are very rare. The large number of long edges also make it hard to determine where an edge starts or ends. Although visualization techniques can be applied to partially solve these problems, one might also wonder to what extent node-link diagrams are part of the problem instead of the solution to visualizing large graphs. Extracting local structure from a matrix visualization requires much more effort however. Figure 6c shows a node link representation of the subsystem shown in figure 3. In general, for small graphs (say less than 100 nodes), we think node link diagrams are superior to matrix representations.

## 6 Conclusions

Since displaying large graphs as visual networks often brings more problems than it solves, we have advocated the use of matrix oriented graph representations. Call matrices have been used to dis-

play dynamic properties of object oriented programs on a smaller scale [de Pauw et al. 1998], we found that they exhibit a number of properties that makes them attractive to use as management tools in large scale software projects as well. The fact that call matrices use a **uniform representation** for calls between two clusters of nodes and calls within a single node cluster avoids problems with multiscale viewing. Their **recursive structure** makes it simple to employ smooth multiscale zooming. Finally, their **visual stability** provides a consistent birds-eye view of the entire software system. This is especially useful in large projects in which already a great deal about the (static) call structure of a system is implicitly known, in which case the matrix representation can also serve as a useful externalization of possibly fragmented knowledge of the architects. We improved on existing methods by providing a smooth semantic zoom between abstraction levels, which makes it easier for users to maintain context.

In principle, the method outlined here can also be used in other application areas, such as gene microarrays, network or phone call data, as long as the input dataset conforms to the requirements mentioned in section 2. These require both an hierarchical decomposition of the dataset (which is not always available especially when not much is known about the underlying structure) and a layering of calls. Further work should focus on relieving these rather stringent requirements. Especially the requirement that calls can only be made between two components in the same layer rules out application of this method to a large number of reverse engineered software datasets, which are often less structured. On the visualization side one can think of interactively expanding or contracting hierarchies, visualizing multiple attributes and making a visual distinction into explicitly defined relations between components and inherited relations. Recent work by [Ziegler et al. 2002] also offers some good ideas on interactivity.

An important topic we have not touched on is the ordering of rows and column within a matrix cell. Ideally, one would like to see the pattern of relations between subcomponents reflected in the row and column ordering. As an example, given subcomponents  $x, y$  and  $z$  with  $C(x, y)$  and  $C(y, z)$ , the row column ordering  $(x, y, z)$  is clearly preferable over  $(z, x, y)$ . Since finding a such 'good' row and column ordering for an arbitrary subgraph is a special case of finding a good clustering [Batagelj and Ferligoj 2000] it is beyond the scope of this paper. We managed to avoid the problem here by manually imposing a suitable ordering where needed. In general, we feel that the potential of using call matrix visualizations as mental aids in large software engineering projects has been overlooked so far and we hope that this paper inspires more people to look beyond node link diagrams for the visualization of very large graphs.

**Acknowledgements** Special thanks go to André Postma, Ben Pronk and Ivo Canjels at Philips Medical Systems for their valuable input and discussions. This work was supported by the Netherlands Organisation for Scientific Research (NWO) under grant 612.000.101.

## References

- ABELLO, J., AND KORN, J. 2002. Mgv: a system for visualizing massive multidigraphs. In *IEEE Transactions on Visualization and Computer Graphics*, vol. 8, no. 1, 21–38.
- ABELLO, J., AND KRISHNAN, S. 1999. Graph surfaces. In *Proceedings of the 4th International Congress on Industrial and Applied Mathematics (ICIAM'99)*, 234–244.
- BATAGELJ, V., AND FERLIGOJ, A. 2000. Clustering relational data. In *Data Analysis*, Springer, Berlin, Gaul, Opitz, and Schader, Eds., 3–15.
- BECKER, R., EICK, S., AND WILKS, A. 1995. Visualizing network data. In *IEEE Transactions on Visualization and Computer Graphics*, vol. 1, no. 1, 16–28.
- DE PAUW, W., KIMELMAN, D., AND VLISSIDES, J. 1998. Visualizing object-oriented software execution. In *Software Visualization*, MIT Press, Stasko, Domingue, Brown, and Price, Eds., 329–346.
- GANSNER, E., AND NORTH, S. 1999. An open graph visualization system and its applications to software engineering. In *Software - Practice and Experience 2000*, vol. 30(11), 1203–1233.
- JOHNSON, B., AND SHNEIDERMAN, B. 1991. Tree-maps: A space-filling approach to the visualization of hierarchical information. In *Proceedings of the IEEE Visualization '91 Conference*, 284–291.
- POSTMA, A. 2003. A method for module architecture verification and its application on a large component-based system. In *Information and Software Technology*, vol. 45, no. 4, 171–194.
- STASKO, J. 1998. Smooth continuous animation for portraying algorithms and processes. In *Software Visualization*, MIT Press, Stasko, Domingue, Brown, and Price, Eds., 103–118.
- STOLTE, C., TANG, D., AND HANRAHAN, P. 2002. Multiscale visualization using data cubes. In *Proceedings of IEEE Symposium on Information Visualization 2002*, 7–14.
- STOREY, M.-A., BEST, C., AND MICHEAUD, J. 2001. Shrimp views: An interactive environment for exploring java programs. In *Proceedings of International Workshop on Program Comprehension (IWPC'01)*, 111–112.
- VAN HAM, F., 2003. Project website at <http://www.win.tue.nl/~fvham/matrix>.
- VAN WIJK, J., AND NUIJ, W. 2003. Smooth and efficient zooming and panning. In *IEEE Symposium on Information Visualization 2003*.
- ZIEGLER, J., KUNZ, C., AND BOTSCH, V. 2002. Matrix browser - visualizing and exploring large networked information spaces. In *Extended Abstracts of the International Conference on Computer Human Interaction SIGCHI 2002*, 602–603.