Technical Report No. 5/94 - Department of Computer Science - University of Bremen - 1994

The Graph Visualization System *daVinci* - A User Interface for Applications

MICHAEL FRÖHLICH AND MATTIAS WERNER

The visual representation of graphs becomes more and more important in today's computer applications, but good graph visualization is difficult to implement and powerful graph drawing tools that could be integrated in application programs are hard to find. This paper presents the interactive graph visualization system *daVinci* that tries to overcome this deficit. *daVinci* provides a hierarchical graph layout of high quality together with comfortable facilities like interactive abstractions and fine tuning layout operations. The system is designed for communication with an arbitrary application program which is using *daVinci* as a graphical user interface for graphs.

1. Introduction

Directed graphs are frequently applied in computer applications as a general data structure for representing objects and relationships between them. Graphs are used to implement hierarchies, dependencies, networks, configurations, dataflows, and so on.

In software systems, it is a good policy to show the user a visual image of internal graph structures for information. Handling the application by interacting with the graph visualization (as with a graph editor) is even better to give the user a more precise feedback about the program. A graphical notation with nodes and edges is often considered to be intuitive and easy to grasp (refer to [1] for a further discussion). But the more complex and large a graph becomes, the more quality aspects of a good graph layout (such as minimizing edge crossings, edge bends or area consumption) must be taken into account (see [2]). Otherwise a viewer may loose orientation and will be confused sooner or later.

Although graph visualization is accepted as indispensable for many domains, the use of this technique is not very common in today's computer applications. Frequently, a user has to deal with uncomfortable textual interfaces or poor ad-hoc drawings of graphs. The reason for this deficit is the great effort to implement a satisfying graph layout. In software development projects there is often a lack of personnel resources or money to realize these "auxiliary features". Furthermore, flexible and powerful graph visualization tools, which are reusable and easy to integrate with applications, are hard to find. These facts were the motivation for developing the interactive graph visualization system *daVinci*. The primary aim was to create a universal system that could be used by arbitrary applications as a user interface to display directed graphs in many different visualization styles.

There are a lot of powerful graph visualization tools such as CG [3], the prototype of Henry [4], GRAB [5], GraphEd [6], GMB [7], dot [8], etc. A detailed survey of each approach would exceed the size of this article. Unfortunately, most of the existing systems for graph visualization are designed as isolated viewing components and do not offer good interaction capabilities for use as a full user interface for application programs.

2. Concepts of *daVinci*

One fundamental concept of our system is the separation from an application that is using *daVinci* as a user interface for graphs. The connected program is exclusively responsible for controlling the graph structure, and the only task of the visualization system is to display this graph on the screen. So *daVinci* is different from conventional graph editors, because the system is not able to alter the graph structure directly by itself. Each modification must be done by an application. For this purpose, *daVinci* provides an application interface for communicating with the connected program. If an application uses this communication mechanism, the user is able to control the system by interacting with the visualization. Details about the interface and examples for existing applications are given below.

Another goal of *daVinci* is the ability to influence the visualization of a graph interactively. Interactions are operations like:

- Using abstractions for temporarily suppressing uninteresting parts of a graph visualization.
- Reducing the scale for getting an overview.
- Improving the graph layout manually (fine tuning).

Experiences have shown us that users often like to influence or modify an automatically generated layout in details, because standard representations are rarely perfect. Some graph drawing systems such as GMB [5], dot [8] or EDGE [9] provide constraints for controlling the graph layout in a declarative way. For example, to define the order of several nodes in the visualization, one has to formulate the desired effect in the constraint language and to add these constraints to the input graph. This method is indirect, difficult and troublesome in certain cases, because constraints could have side effects on other parts of the layout. In *daVinci* it is possible to influence the generated graph layout directly “at runtime”. These interactions have an immediate feedback on the visualization, so one will get results much faster.

The base of *daVinci* is the automatic layout of a graph. For the layout algorithm the method of Sugiyama et al. [10] was adopted and improved in details. It was particularly important to treat the integration of the automatic layout and the interaction capabilities. During the layout process, the system must take previously performed interactions into account. For example, modifications to the visualization of a graph have to be considered if the user is changing the order of nodes interactively in order to eliminate some remaining edge crossings. The resulting graph constellation is always used as a base for a new call of the layout algorithm.

3. Graph Layout

daVinci graphs are provided in an ASCII term representation that allows to use attributes for influencing the visualization of nodes and edges. Cyclic graphs are supported as well as graphs with multiple edges, i. e. two nodes in a graph may have more than one edge between them. The algorithm of Sugiyama requires an acyclic graph, so *daVinci* uses a prepass to transform cyclic into acyclic graphs by inverting one of the edges in each cycle.

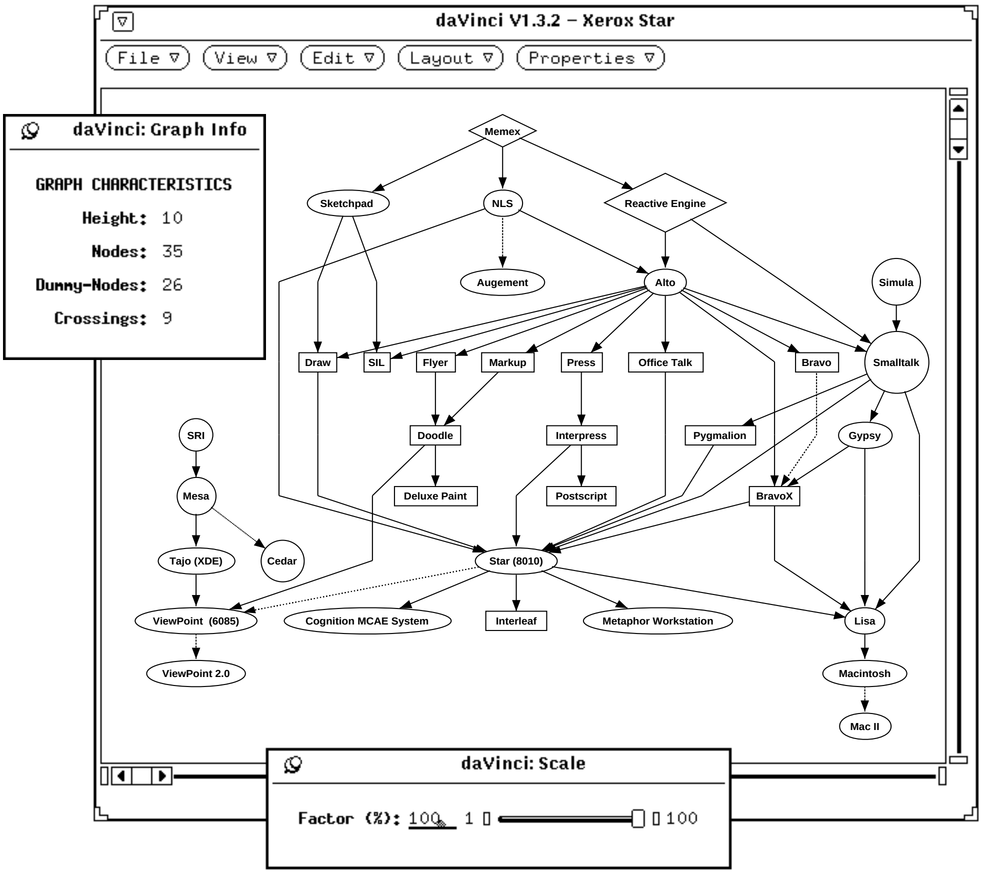


Figure 1. Visualization of a graph with *daVinci*. This graph is taken from [11] and was used as an example in [9]. The graph shows the dependencies of systems related to the Xerox Star project. Different shapes are used in this graph visualization to distinguish hardware (ellipses), programming languages (circles), software (boxes) and virtual systems which were never realized (rhombs). The graph layout was optimized with fine tuning operations.

By using node- and edge attributes in the term representation of a graph, it is currently possible to specify text, font, color and shape of nodes and color and pattern of edges. Figure 1 shows the visualization of a graph in *daVinci*, where attributes are used to set different node shapes. The visualization facilities of the system will be enlarged in the future to permit more specific graph representations. For example, it will be possible to draw parts of a graph as nested boxes, as formatted text or in separate windows.

daVinci's graph layout algorithm for minimizing edge crossings is based on the heuristic method which was introduced by Sugiyama et al. [10]. To discuss our improvements, we present a brief survey of the original approach. For more details, the reader should consult the quoted article of Sugiyama.

The algorithm is generating a hierarchical top-down representation of a directed acyclic graph. By convention, the origin node of a directed edge is called parent and the target node is called child. In a first step, all of the nodes are separated into horizontal levels to get a “proper hierarchy”. Long span edges which are connecting nodes on non-adjacent levels are filled up with invisible dummy nodes at each intermediate level. In the resulting hierarchy, each edge now leads downwards and connects two nodes or dummy nodes on adjacent levels. In the next step each node gets a variable indicator value for edge crossings, the “barycenter”. Figuratively speaking, the barycenter of a node is the “center of gravity” (i. e. the current average position) beneath the parent nodes and above the child nodes. So each node has two different values: The upper barycenter with the average position of the parent nodes and the lower barycenter with the average position of the child nodes. It was observed that sorting the nodes on a level according to their barycenter causes a reduction of crossings for the edges from the level of parent nodes (for the upper barycenter) and for the edges to the level of child nodes (for the lower barycenter). The so called phase 1 of the Sugiyama algorithm processes each level of a graph from top to bottom and afterwards from bottom to top. While going down, an upward sorting according to the lower barycenter is performed for the nodes of each level. On the return to the top of the graph, the nodes of each level are sorted according to their upper barycenter in the same way. This complete pass is repeated in phase 1 for a given number of iterations to get a more or less stable order of nodes. Another phase 2 is applied between different calls of phase 1 to invert the order of node sequences with equal barycenters on each level. This procedure should break the stability of constant node orders which cannot be modified with phase 1 („This operation has been empirically found to be effective", [10]). Both phases are alternating for a given number of iterations which can be set in the user interface of *daVinci* to control the considerable cost of the layout algorithm.

We observed that inverting the order of adjacent nodes with the same barycenter in phase 2 does not produce good results at all times. In many cases, phase 1 cannot modify the order of nodes after processing phase 2, because the barycentric sorting has stabilized. So the following phase 2 inverts the same node sequences as in the former call of phase 2 and the old node order is restored. At this point, the layout algorithm runs into a cycle and can not produce additional, probably better node orders with fewer crossings. This situation was also described by Rowe et al. in [5].

To avoid these problems, we have improved phase 2 of the Sugiyama algorithm by using a random distribution for the node sequences with equal barycenters instead of inverting the order. Each call of phase 2, based on the same initial graph, generates a different node order. So cycles cannot occur and *daVinci*'s layout algorithm often produces much better graph visualizations with fewer edge crossings than the original method. Unfortunately, resulting from the usage of a random distribution, the graph layout may be worse in rare cases than the layouts calculated with the traditional algorithm. Furthermore, the layout in *daVinci* is non-deterministic and one node order can hardly be reproduced for the same graph in another session unless the initilization value of the random function is kept.

The advantage of our approach can be demonstrated with a comparison of different systems for graph visualization. In [10] Sugiyama et al. have used a hierarchical graph representation of Forresters dynamic world model [13] as an example. With the original algorithm, they were able to reduce the number of edge crossings to 72. Rowe et al. have applied the same graph in [5] to show the capabilities of the GRAB system. With an improved layout method, they obtained a graph drawing with 49 edge crossings. But while transferring the graph example of Sugiyama, the authors made two mistakes by adding an additional edge from node 16 to 4 and by omitting the edge from node 15 to 16. Newbery-Paulisch have also shown the world model graph in [9], drawn by the EDGE system with 40 crossings.dot [8], the successor system of DAG [12], is able to generate an automatically calculated layout for this graph with 29 edge crossing. This was the result of our own experiments with dot. Figure 2 illustrates the world model graph drawn by *daVinci*. With the improved layout algorithm it was able to calculate a graph visualization with only 23 edge crossings. We have to remark that these good results cannot be achieved in *daVinci* with each execution of the layout algorithm, caused by the non-determinism of this procedure. The average results for this graph are between 23 and 32 edge crossings

Our studies of the non-deterministic layout method have shown that the variation in number of crossings for all iterations is much larger than the variation of the traditional algorithm of

Sugiyama. The best layout with the minimal number of edge crossings is usually obtained in between and not after finishing all iterations. So using the node order of the last iteration (as it is generally done) would yield poor results. In *daVinci*, the state of each iteration is kept in a history to recover the best calculated graph layout at the end..

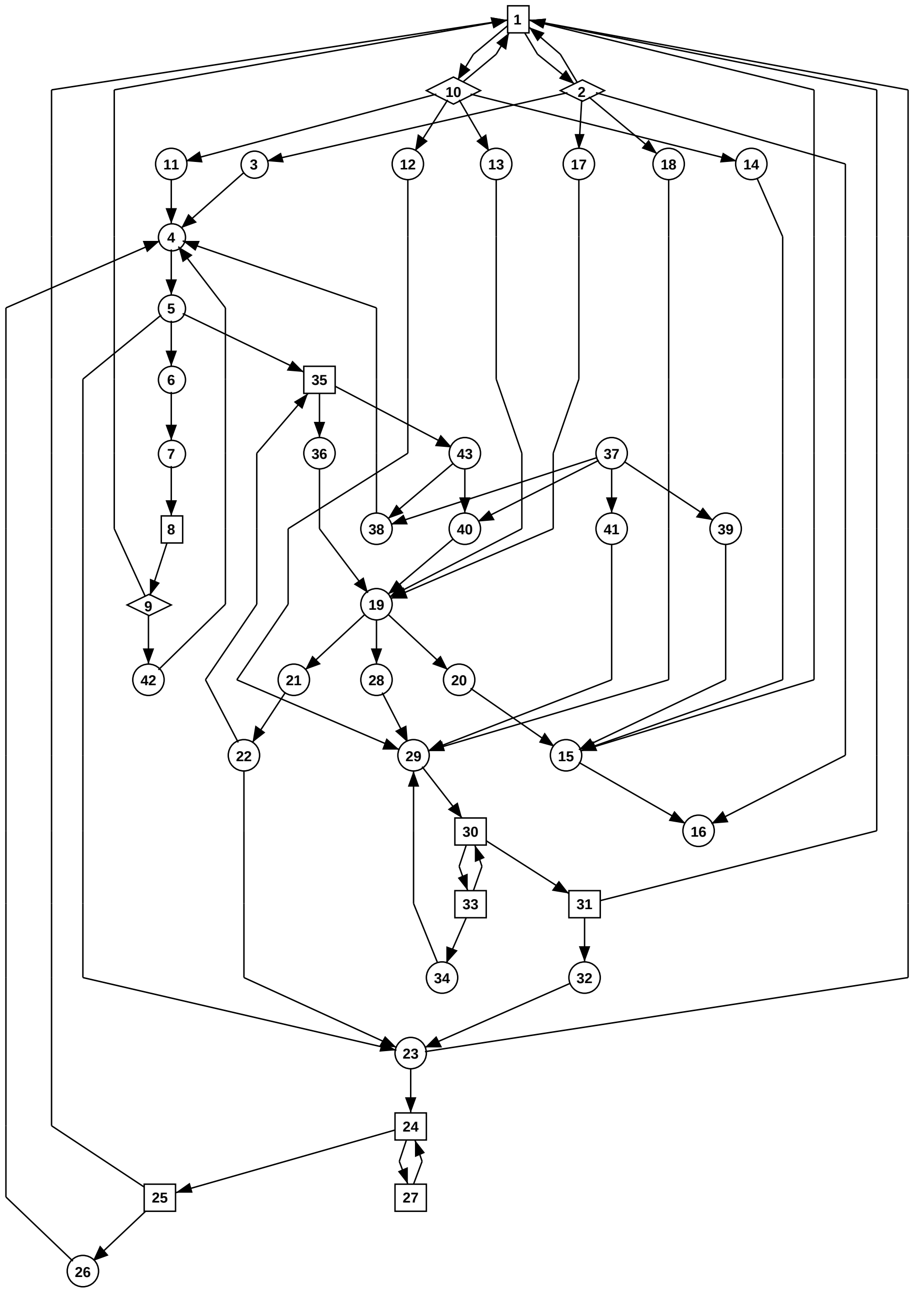
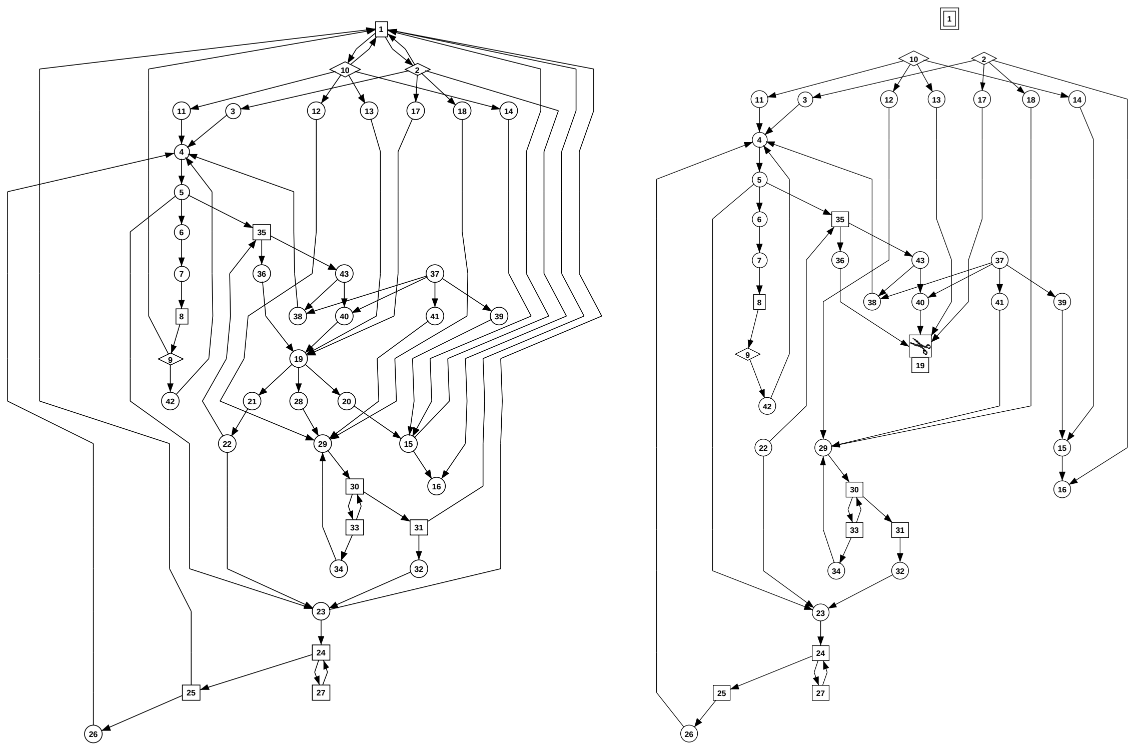


Figure 2. Graph of the dynamic world model drawn by *daVinci*. This graph is a standard example for graph visualization systems (see [5], [9], [10]) and was originally taken from [13]. The optimized automatic layout algorithm of *daVinci* has reduced the number of edge crossings down to 23.

Another improvement of the layout algorithm deals with the optimization of the long span edges that connect two nodes on non-adjacent levels. In the algorithm of Sugiyama it is not guaranteed that these edges are as straight as possible. Long span edges with many bends are difficult to follow for a beholder. After the layout procedure a new heuristic edge straightening



(a) (b)

Figure 3. Both illustrations show the graph of figure 2. (a) Graph layout in *daVinci* without performing the edge straightening algorithm. (b) This visualization demonstrates the use of interactive abstractions in *daVinci*. The subgraph at node 19 is faded out with the subgraph hiding function. An icon with scissors is used to show the applied abstraction. Furthermore, the edges of node 1 are suppressed with the edge hiding function. This kind of abstraction is shown in the visualization by surrounding the node with a box.

algorithm is performed in *daVinci* to reduce the number of bends. For each long span edge, two additional routes are calculated. The first is placed downwards from the parent node to the child node and the second is placed upwards from the child to the parent. Both routes are directed as straight as possible, but without moving any other node. The best result among the original route and the two new generated routes is selected by an evaluation function. This procedure is performed twice with each long span edge. At first, all edges in the graph are processed from left to right with the described method. Afterwards, the edges are optimized again by processing them from right to left. This second pass utilizes the space that was produced by the first pass. The results of this algorithm are very good in most cases. The world model graph in figure 2 was visualized with the described edge straightening algorithm. Figure 3 (a) demonstrates how the graph would be drawn in *daVinci* without this layout optimization.

4. Interactive Refinements

The considerable complexity of graph layout problems implies the need for heuristics, but with the use of heuristics it is not guaranteed that the algorithm returns the most optimal result. To deal with this deficit, *daVinci* offers interactive operations for refining the layout in a direct way.

In contrast to other systems with a static visualization, *daVinci* offers the possibility to improve the calculated graph layout by moving nodes with the mouse. A user can adjust the position of nodes and is even able to change their order. This fine tuning technique can be used for eliminating unnecessary edge crossings by hand that cannot be avoided by the heuristics at the moment. Based on the corrected graph visualization, it is possible to start the layout algorithm again for further improvements. Fine tuning can also be used for restoring semantic information of a graph in the visual representation: for example, related nodes can be brought together or particular edges can be made vertical. Interactive fine tuning is particularly useful if a graph visualization is intended for presentation or publication. In the future, it should also be possible to use constraints (see [7], [8] and [9]) in *daVinci* for defining the requirements on the layout, but without abandoning fine tuning.

After the user has improved the layout, the system state can be saved to keep the applied interactions for another session. Furthermore, a graph visualization can also be saved in Encapsulated PostScript Format for printing or exporting to other programs.

Interactive abstractions can be used to suppress uninteresting parts in a graph to simplify the visualization. Two interactive abstractions are implemented at the moment: subgraph hiding and edge hiding. With the first kind of abstraction, the subgraph of the currently selected node can be suppressed in the visualization. All of the childs are omitted except for nodes which have parents outside this subgraph. The space saved by this abstraction can be utilized by the graph layout algorithm on user demand. The other interactive abstraction is edge hiding, used to leave out all edges of a selected node. This operation is particularly useful if a node has many incoming and outgoing edges that produce a lot of edge crossings. After using an edge abstraction for these nodes, the traceability of the remaining edges is substantially better. Figure 3 (b) demonstrates the use of interactive abstractions for the world model graph.

With another interaction it is possible to get an overview by reducing the scale of a graph visualization continuously. In a scaled representation, all interactions such as fine tuning and abstractions are still available. Scaling is particularly useful for large graphs to see the complete structure. Naturally, details like the text of nodes are omitted in a reduced scale, so it is desirable to have both a normal view and a scaled view. This feature will be realized in the future by supporting multiple coupled views in several windows (refer to [4] and [14]).

The described interactive operations are only affecting the layout and not the structure of a graph. However, it was the intention to provide a user interface for application programs. A user should control the application by interacting with the visual representation of a graph. Due to the availability of a clearly arranged visualization, he is permanently informed about the internal state of the application. With this facility it is possible to improve the transparency of an application drastically (refer to [15]). Interactions should trigger graph manipulating operations in the application and should have an immediate feedback to the visualization. For this purpose, *daVinci* provides an application interface for the communication with a connected application program.

5. Application Interface

An arbitrary program can be connected to the application interface in order to use *daVinci* as a user interface for graphs. The communication between both systems is event driven, i. e. the application sends a graph and is informed about events like the selection of a node by the user. After receiving an event notification, the application must interpret this information in its own context. Depending on the event, the application can modify its own data structure and send an updated graph back to the visualization system to display the changes. So the application interface is bidirectional. At the moment, the communication with an application is realized with UNIX pipes. On the first pipe, an application sends commands to control the visualization system, on the second pipe it receives answers from *daVinci* (i. e. about events triggered by the user). The independence of both systems can be regarded as an advance of this loosely coupled style, so *daVinci* can be reused as a universal display component. The separation between a graph visualization system and graph administrating application was applied with the system GraphView before [16], but without providing the manifold capabilities of *daVinci*'s application interface.

A typical communication process looks as follows: after connecting with *daVinci*, an application transmits a menu tree with operations that could be applied to the displayed graph. These menus are automatically integrated into the *daVinci* user interface at runtime. The application is responsible for modifying the graph structure, so only the names of operations in the menus and not the functions that should be performed are given to the visualization system. *daVinci* sends an event notification with this name if a menu entry is selected by the user. Afterwards, the application has to process the corresponding operation on its own data structure. The application is able to deactivate each menu entry for selection to avoid the choice of non-applicable operations in the current state.

After defining the menus, an application transmits a graph to the visualization system. The graph is immediately displayed in a window by *daVinci*. Afterwards, the user is able to use the above described interactions to manipulate the layout. Meanwhile, the application is informed about each selection of nodes and edges. The application interprets these events and decides which operations are applicable for the current selection. Then the corresponding menu entries are enabled by the application interface. If the user selects one of the activated menu entries, the application will be informed about this event and performs the corresponding operation on its own data structure. Finally, a new graph with the modifications is sent to *daVinci*. For example, a graph editor application could specify a menu "delete node". This menu entry is enabled by the application if any node is selected. When the user selects this entry in the menu, the graph editor deletes the chosen object in its data structure and sends a new graph without the deleted node to *daVinci*. In the future, it will be possible for an application to send only incremental updates of a graph after modification to guarantee stability of the layout.

Further commands are provided by the application interface to display status information inside the visualization window or to execute dialogs with the user which can be specified at runtime (so *daVinci* can be viewed as an interpreter for user interfaces). With dialogs, an application is able to execute inquiries and is informed about the input of the user afterwards. So the graph editor application in the example above could ask the user for confirmation if the node should really be deleted. Currently the use of dialogs is limited in *daVinci*. There are only dialogs for entering strings or choosing among two alternatives. In the future the application interface should provide much more complex dialogs as known from modern user interface toolkits (e. g. buttons, exclusive and non-exclusive choices, sliders, checkboxes, etc.).

Other commands of the application interface permit an application to perform each *daVinci* operation that is available for the user. By using these commands, an application is able to control the visualization system in a precise manner. For example, an application is able to suppress subgraphs by using abstractions, it can reduce the scale of the visualization and is able to save the current graph in a PostScipt file. A more detailed description of the application interface and the further functionality of *daVinci* can be found in the user manual [17].

6. Implementation and Efficency

daVinci is implemented in the functional programming language ASpecT developed at the University of Bremen. One of the characteristics of functional programming languages is the absence of variables in the usual sense. Furthermore the language is strongly typed. For these reasons, a developer is obliged to use a clean programming style and side effects are prevented. In addition, ASpecT provides powerful abstraction and modularization possibilities and offers a huge library of reusable functions and abstract data types like lists, arrays, sets, trees and so on.

The ASpecT compiler translates the functional program to C code and makes it possible to write fast and efficient, yet portable software systems in this way. According to our experiences, the runtime overhead of ASpecT is rather small in compare to hand-coded C programs. By using C as an intermediate language, it is possible to write certain ASpecT functions directly in C. This feature was utilized to implement the user interface of *daVinci* with the XView toolkit. Choosing the functional language ASpecT for implementing *daVinci* was an important decision for our project. In spite of small personal resources, we were able to realize impressive results very quickly. Using a traditional programming language like Modula 2 or C would have taken a multiple amount of time.

Since *daVinci* is an interactive system, the execution time of operations must be small enough to avoid undesirable delays. In particular, critical operations must be taken into account which turn a user impatient in case of delay. Examples for critical operations in *daVinci* are scrolling, refreshing the screen and interactions such as fine tuning or abstractions. One of the primary aims was the efficient implementation of these critical operations.

The drawing routines of *daVinci* are designed for reusing already drawn parts of a graph visualization without performing unnecessary output operations. For this purpose, the drawing area is divided into a grid of smaller segments. In addition to the drawing area of the window, there are two more pixmaps for each grid segment: one for the nodes and one for the edges. Two pixmaps are needed for each segment, because edges may overlap nodes in rare cases. The nodes and edges of a segment are not drawn before the segment becomes visible in the window for the first time. After a segment is drawn, the image is transferred from the pixmaps into the window with copy operations. All of the drawn segments are kept for reuse until the graph visualization changes on a large scale. Due to this cache technology it is possible in *daVinci* to scroll back into a previously visible area without any noticeable delay. Furthermore, it is not necessary to draw the whole graph at initialization time, which would result in unacceptable delays for large graphs. In *daVinci* the scrolling is implemented by shifting the window content for the corresponding amount and copying the freshly appearing parts from the segments. Ideally only fast copy operations have to be executed if the segments are up-to-date.

Particular modifications in the segments such as selecting nodes, applying abstractions or using fine tuning do not require a complete redrawing of the segment pixmaps. Instead there are object-related delete-, draw- and copy-functions for nodes and edges to recover a segment with minimal cost. For example, if one is using the fine tuning by moving a node with the mouse, then this node is shifted on the node pixmap with a copy operation. Furthermore, only the edges of the moved node are deleted on the edge pixmap and are drawn at the new position afterwards. So only minimal amount of required operations is performed to update the visualization of a graph.

7. Existing Applications and Availability

As a result of the surprising worldwide approval, *daVinci* is now a wide-spread system for the visualization of graphs. Responses of users have shown us the versatile nature of the program. For example, *daVinci* is used for drawing module dependency graphs, dataflow diagrams, function call graphs, class hierarchies or syntax trees. The system is also applied for the representation of neural nets, integrated circuits, molecule structures, computer networks or for showing dynamic processes in a parallel program. Some existing application programs, which are using *daVinci* as a user interface for graphs, should be described in more detail:

- The authors have implemented a graph editor that is part of the *daVinci* distribution. The editor is a separate program, communicating with the visualization system by using the application interface. With this application it is very easy to create graphs interactively.
- The University of Karlsruhe in Germany has developed the interactive program verification system KIV [18]. This system uses *daVinci* as a user interface for displaying “project graphs” to offer the user a survey of the program that should be proved.
- The Parallel System Research Group of the University of Exeter, UK, applies *daVinci* to animate timed petri-nets. The application is a simulator for generating petri-nets sent as a *daVinci* graph to the application interface.
- Sun Microsystems Laboratories Inc. in Chelmsford, USA, uses *daVinci* in a research project to visualize the distribution of World Wide Web documents on the Internet. The aim of the project is to improve the usability and accessibility of information on the Internet. The application program of Sun uses the well known Mosaic system for displaying WWW documents and *daVinci* for showing the graph of documents visited.

The interactive graph visualization system *daVinci* has been developed by the authors since December 1992. The program is freely available for non-profit use. The currently supported computer platforms are Sun SPARC workstations and Intel PC’s with Linux. Ports to other UNIX platforms will be available in the future. In a period of nine months, *daVinci* was transmitted by our ftp server (ftp.Uni-Bremen.DE in /pub/graphics/daVinci) more than 1000 times to educational and commercial sites all over the world. The system is also available on several ftp servers in North America, Japan and Australia.

8. Future Work

Besides the improvement of the existing system components (particularly graph layout, interactions and application interface), there are three fields of further research efforts: supporting new visualization possibilities for nodes and edges, providing visualization definitions for a rule based specification of the graph representation and offering stability for dynamic graph structures.

Even now it is possible in *daVinci* to influence the visualization of a graph by using attributes in the external term representation. This can be done by assigning graphical objects to nodes and edges. At the moment, the graphical objects for nodes cover boxes, circles, ellipses and rhombs. Additional objects should be available in the future to offer novel types of visual representations. Enclosing objects of arbitrary size will be provided to draw the (acyclic) subgraph of a node inside this graphical object. In certain cases, this can be done for the nodes of a subgraph too, so one will get a hierarchy of nested boxes. Furthermore, graphical objects will be provided to visualize subgraphs as formatted text or to draw them in a separate window. These new features may be used to get more compact visualizations, particularly important for large graphs.

To draw a graph different from the default visualization, a graphical object has to be assigned individually to each node and edge at the moment. In the future, it will be possible to specify the visual representation for a category of graphs by rules in a visualization definition.

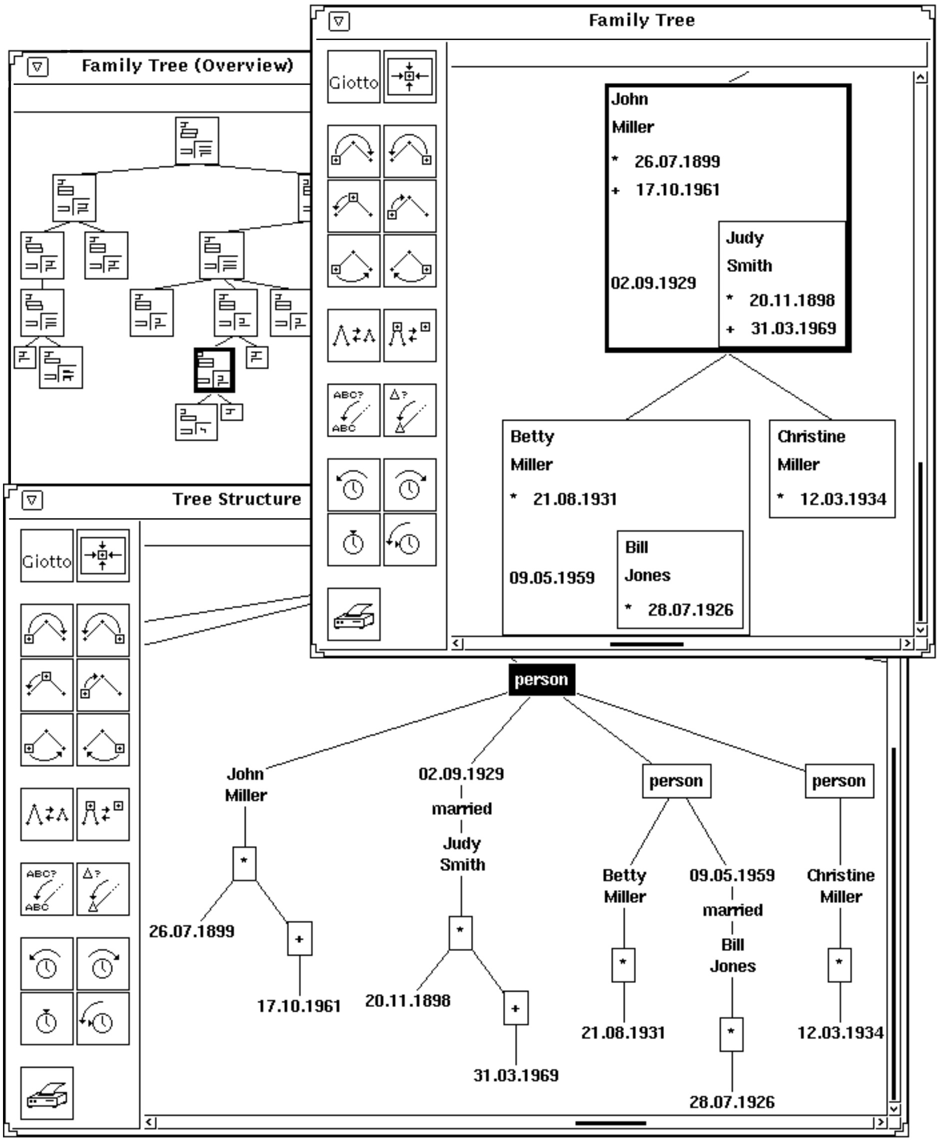


Figure 4. Visualization of a family tree in three different views with GIOTTO, the successor system of *daVinci*. The use of enclosing objects is demonstrated here.

Each node and edge of a graph will get a type for this purpose. The user will be able to write a rule to define the visualization of each node or edge that matches the type. Constraints will also be provided with a visualization definition to control the layout of a graph.

Some of these features have already been implemented for the non-interactive tree visualization system GIOTTO, the prototype of *daVinci*. GIOTTO is an acronym for "Graphical Instead Of Textual or Tabular Output" and was the master's thesis of the authors [14]. Several months after finishing the work with GIOTTO, we unfortunately found out that Tamassia et al. had realized a system with the same name [2]. Figure 4 demonstrates the visualization of a family tree with our GIOTTO in three different views. Each view is generated from the same tree structure. In the lower window "Tree Structure", one can find the tree in a conventional (space consuming) representation. The window "Family Tree" shows the tree in another, more compact view, where subtrees with person data are drawn as formatted text and married persons are placed inside the node of the partner by using enclosing graphical objects. In the window "Family Tree (Overview)", one can see the family tree in a reduced scale. The simultaneous availability of different views can be achieved in GIOTTO by using a separate visualization definition for each view. For example, the representation of persons with enclosing objects and formatted text is defined in one rule. If any structure with a family tree is loaded, then this rule will be applied to all person subtrees automatically to get the desired results. This comfortable method for specifying the visualization of a structure will be available in *daVinci* shortly.

Another area of research is the support of stability for the graph layout. This is particularly important for applications using *daVinci* as a user interface. At the moment, an application must send a whole new graph to the application interface if the displayed structure has changed. Even small modifications to a graph may result in extremely different layouts, so a user could be confused. To solve this problem, an application shall be able to send only the graph increments in this situation. These informations can be used to update the graph visualization with a minimal number of layout modifications.

9. Conclusion

We have presented a survey of the interactive graph visualization system *daVinci*, which has several advanced characteristics. For the hierarchical representation of directed graphs, an improved layout method based on the algorithm of Sugiyama is used. The visualization of a graph can be influenced with attributes. After a graph is displayed, the user is able to modify the layout in an interactive way. Nodes and edges can be moved with the mouse for fine tuning, parts of a graph can be suppressed temporarily with interactive abstraction operations, and the scale of a visualization can be reduced continuously to get an overview.

One of the fundamental ideas of the system is the connection with an external program by using the application interface. The application program is administrating the visualized graph structure. With this concept, *daVinci* can be applied as a user interface for arbitrary applications that want to provide a graphical view to their data structures without loosing control of the displayed graph and without dealing with visualization issues. Thus a user is able to operate with the application by interacting with the graph visualization in *daVinci*.

Visual representations of graphs are needed in computer programs more than ever to show structures in an intuitive way. But the considerable costs of high quality graph layout often prevent the optimal realization of this demand. To simplify the development of applications, a powerful, easy to integrate and reusable graph visualization system such as *daVinci* is needed. The amazing worldwide interest in *daVinci*, documented by numerous existing applications, has shown us the soundness of our approach.

References

1. R. Arnheim (1972) Visual Thinking. University of California Press, Berkeley.
2. R. Tamassia, G. di Battista, C. Batini (1988) Automatic Graph Drawing and Readability of Diagrams. *IEEE Transactions on Systems, Man, and Cybernetics*, Vol. 18, No. 1, pp. 61-79.
3. C. L. Combs (1992) An Automated Drawing System for Directed Graphs. Master's Thesis. Department of Computer Science and Engineering, Auburn University, Alabama.
4. T. R. Henry (1992) Interactive Graph Layout: The Exploration of Large Graphs. Ph. D. Thesis. Technical Report 92-03. University of Arizona.
5. L. A. Rowe, M. Davis, E. Messinger, C. Meyer, C. Spirakis, A. Tuan (1987) A Browser for Directed Graphs. *Software - Practice and Experience*, Vol. 17, No. 1, pp. 61-76.
6. Himsolt, M. (1993) A View to Graph Drawing Algorithms through GraphEd. In: *Proceedings of the International Workshop on Graph Drawing '93*. Sèvres, France, pp. 117-118.
7. D. Jablonowski, V. A. Guarna jr. (1989) GMB: A Tool for Manipulating and Animating Graph Data Structures. *Software - Practice and Experience*, Vol. 19, No. 3, pp. 283-301.
8. E. Koutsofios, S. C. North (1993) Drawing Graphs with dot - dot User's Manual. Technical Report. AT&T Bell Laboratories, Murray Hill, New Jersey.
9. F. Newbery-Paulisch (1993) The Design of an Extendible Graph Editor. *Lecture Notes in Computer Science* 704. Springer-Verlag, Heidelberg.
10. K. Sugiyama, S. Tagawa, M. Toda (1981) Methods for Visual Understanding of Hierarchical System Structures. *IEEE Transactions on Systems, Man, and Cybernetics*, Vol. 11, No. 2, pp. 109-125.
11. J. Johnson, T. L. Roberts, W. Verplank, D. C. Smith, C. H. Irby, M. Beard, K. Mackey (1989) The Xerox Star: A Retrospective. *IEEE Computer*, Vol 22, No. 9, pp. 11-29.
12. E. R. Gansner, S. C. North, K. P. Vo (1988) DAG - A Program that Draws Directed Graphs. *Software - Practice and Experience*, Vol. 18, No. 11, pp. 1047-1062
13. J. W. Forrester (1971) World Dynamics. Wright-Allen, Cambridge, Massachusetts.
14. M. Fröhlich, M. Werner (1991) GIOTTO - Graphical Representation of Structured Objects (in german language). Master's Thesis. Department of Computer Science, University of Bremen, Germany.
15. G. Raeder (1985) A Survey of Current Graphical Programming Techniques. *IEEE Computer*, Vol. 18, No. 8, pp. 11-25.
16. B. Birgisson, G. Shannon (1989) GraphView: An Extensible Interactive Platform for Manipulating and Displaying Graphs. Technical Report No. 295. Computer Science Department, Indiana University, Bloomington, Indiana.
17. M. Fröhlich, M. Werner (1994) *daVinci* V1.3 User Manual. Institute for Formal Methods in Software-Engineering, University of Bremen, Germany (available with each *daVinci* distribution).
18. W. Reif (1994) Formal Software Development in the KIV System. To appear in: M. Broy, S. Jähnichen: KORSO, Correct Software by Formal Methods. *Lecture Notes in Computer Science*. Springer-Verlag, Heidelberg.