ON RESOURCE-BASED CONFIGURATION—RENDERING COMPONENT-PROPERTY GRAPHS

Oliver Niggemann * **Benno Stein** * **Michael Suermann** *

* *Department of Mathematics and Computer Science / Knowledge-based Systems*
University of Paderborn, D–33095 Paderborn, Germany
email: {murray | stein | michel}@uni-paderborn.de

Abstract: In a broader sense, this paper is on knowledge acquisition support for configuration problems. Actually it concentrates on resource-based configuration—precisely: on the visualization of component-property graphs. A component property graph is a directed graph in first place, which defines nodes (components, functionalities) and directed edges (relations between components and functionalities).
Each resource-based configuration problem defines such a component-property graph. From a knowledge engineering viewpoint, the visualization of this graph is of great value: It can illustrate functional cause-effect chains within complex technical systems, exhibit crucial points in the modeling, or organize the knowledge base of a large system into useful parts.
This paper provides an answer to the following question: Given a knowledge base containing a resource-based component model—in which way and to which quality can the underlying component-property graph be drawn automatically?

Keywords: configuration, graph visualization, clustering, heuristic search

1. INTRODUCTION

Configuration is the process of composing a technical system from a predefined set of objects. This process relies on a particular component model, which is a useful abstraction of the domain and the technical system to be composed (Tong, 1987), (Stein, 1995).

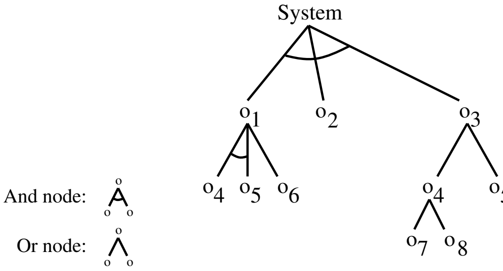

Figure 1: *And-Or tree of some technical system.*

In this context the visualization of a configuration knowledge base of a given configuration problem actually means to visualize the underlying component model. Such a visualization should not happen by a universal approach but exploit meta knowledge concerning the type of component model under investigation. E. g., hierarchical component models, which form the basis of skeletal configuration problems, are often visualized in the form of And-Or trees. Figure 1 gives an example.

Also resource-based component models can be visualized as graphs, so-called component-property graphs; see Figure 2 for an example. Component-property graphs will be discussed in greater detail below.

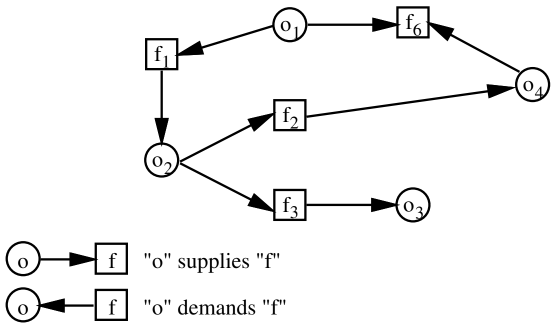

Figure 2: *Component-property graph of a technical system.*

Matter of the paper in hand is the visualization of resource-based component models, so to speak, of the related component-property graphs. Why is such a visualization useful? The main reasons lie in acquisition and maintenance purposes:

Resource-based component models are defined locally, component by component, and normally they are formulated textually. A consequence is that the global view, which a knowledge engineer has developed in his mind, will go lost sooner or later in the course of the acquisition process. Moreover, knowledge bases are maintained by several persons, and, modifications and revisions of a knowledge base may occur a long time after its initial building.

In this connection a visualization of the component-property graph is of great value. It can illustrate functional cause-effect chains within complex technical systems, exhibit crucial points in the modeling, or organize the knowledge base of a large system into useful parts. This wish immediately raises the question: Given a knowledge base containing a resource-based component model—in which way and to which quality can the underlying component-property graph be drawn automatically?

This paper gives answers to these questions; it is organized as follows. Section 2 provides a short introduction to the resource-based configuration approach. Section 3 presents some generic issues respecting graph visualization, while section 4 shows how component-property graphs, implicitly defined in a knowledge base, can be rendered properly. Section 5 compares our visualization results to those of existing tools.

2. RESOURCE-BASED CONFIGURATION

This section provides a short introduction to resource-based configuration.

2.1 *The Resource-Based Component Model*

The resource-based component model establishes an abstraction level that reduces a domain to a set of components described by simple functionality-value-pairs. More precisely, all technical properties that are relevant for the configuration process form a set of resources (functionalities), which are supplied or demanded by the components.

E. g. when configuring a small technical device such as a computer, one property of power supply units could be their power output, and one property of plug-in cards could be the cards' power consumption. Both properties are reflected by the resource "power": A power supply unit *supplies* some power value, while a plug-in card *demands* some power value. Figure 3 depicts some resource-based descriptions of computer components.

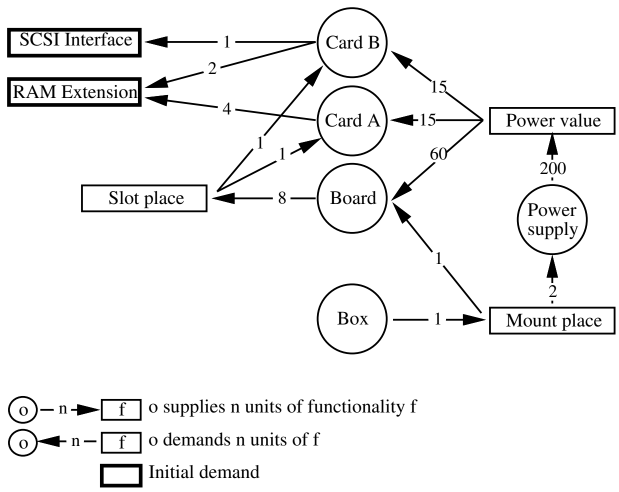

Figure 3: *Resource-based component descriptions*

Note that a component-property graph as shown in Figure 3 represents a simplified functional model of the domain. Actually, resource-based configuration means the instantiation and simulation of such a functional model.

The resource-based component model is suitable for a configuration problem if the following conditions are fulfilled: (*i*) structural information plays only a secondary role, (*ii*) the components can be characterized by resources that are supplied or demanded, and (*iii*) the components' properties are combined in order to provide the system's entire functionality.

A precise specification of the resource-based configuration problem and its solution can be found in (Stein, 1995).

2.2 *The Configuration Process*

Main purpose of this subsection is to round off the introduction of resource-based configuration; it is of secondary importance with respect to visualization aspects.

If there exists a configuration C that solves a resource-based configuration problem, C can be determined by means of the balance algorithm. This algorithm operationalizes a generate-and-test strategy and has been operationalized in the configuration systems Cosmos, Ccsc, Akon, and Mokon (Heinrich and Jüngst, 1991), (Laußermair and Starkmann, 1992), (Weiner, 1991), (Stein and Weiner, 1991). The generate part, controlled by propose-and-revise heuristics or simply by backtracking, is responsible for selecting both an unsatisfied functionality f and a set of objects that supply f. The test part simulates a virtual balance. A functionality (resource) is called unsatisfied, if its supplied amount x is smaller than its demanded amount y.

Basically, configuration works as follows. First, the virtual balance is initialized with all demanded functionalities, and C is set to the empty set. Second, with respect to some unsatisfied f, an object set is formed; the related supplies and demands are added to the corresponding functionalities of the balance, and C is updated by the object set. Third, it is checked whether all functionalities are satisfied. If so, C establishes a solution of the configuration problem. Otherwise, the configuration process is continued with the second step.

3. VISUALIZING GRAPHS

3.1 *Generic Concepts*

When visualizing graphs the question about a definiton of a good or lucid layout plays a key role. Important criteria for such a layout are:

- No two vertices should intersect, and
- neither should an edge and a vertice.
- Having placed all vertices, edges should be drawn by straight lines.
- The angle between two intersecting edges should be obtuse.
- Parallel edges should not be too close.
- Clusters in the graph should be identifiable as such.

Note, however, that both the criteria and their importance depend on the actual design problem. Various approaches for visualizing graphs exist; some visualize graphs by constructing just one layout, others construct several possible layouts and choose the best. The algorithms also vary in their usage of domain knowledge: some are general graph visualization tools, but many, like ours, exploit features of the domain.

3.2 *More on Component-Property Graphs*

In component-property graphs, components point only to functionalities and vice versa. As a consequence, component-property graphs are bipartite. Figure 4 shows a different layout of the graph of Figure 2 that emphasizes this characteristic.

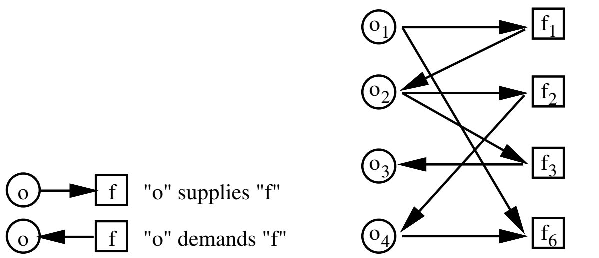

Figure 4: *The component-property graph of Figure 2.*

This bipartite-characteristic can be exploited when constructing a layout for component-property graphs: If the nodes of a graph are arranged in several layers, no edges must be drawn *within* a layer, if all nodes in the layer are of the same type (cf. *The Leveling Step* in the next section).

Recall that component-property graphs represent a functional model of a technical system. Technical systems are composed from subsystems or modules, which finally are built up with components. Although in a configuration knowledge-base these submodules are not labelled as such, they can often be identified by a smart clustering algorithm (cf. *The Clustering Step* in the next section).

Such a clustering algorithm exploits the fact that the components within a module are closely connected, while the modules themselves are connected via rather narrow interfaces. The automatic identification of modules is a particularity when visualizing component-property graphs; note that standard approaches for graph visualizations may consider user-defined clusters but do not try to identify clusters on their own.

It is quite evident that a clustering that resembles the technical setup of a system is a benefit from the knowledge-engineering standpoint.

4. VISUALIZING COMPONENT-PROPERTY GRAPHS

We now present a visualization algorithm exploiting the already mentioned features of the graph and considering our criteria of a lucid layout.

4.1 *Strategy*

Three major steps to a good layout can be identified. In a first step the clusters must be found. The next task consists of finding a good arrangement of the clusters on the page, finally we seek positions for the nodes within a cluster.

4.2 *The Clustering Step*

How can a cluster be identified? As mentioned above clusters relate to modules or subsystems of a technical system, therefore we expect less edges between clusters than within a cluster: Modules are combined by only a few edges, while the components within a cluster are highly connected. A clustering divides the vertices V of a Graph $G = (V, E)$ into disjunct sets $C_1, \ldots, C_n, n \in N$.

Definition 4.1 (Clustering). A clustering $\mathcal{C}$ of a graph $G = (V, E)$ is defined as follows:
$\mathcal{C} = \{C | C \subseteq V\}$, and $\forall C_i, C_j \in \mathcal{C} : C_i \cap C_j = \emptyset$.

The cut of two disjunct sets of vertices is defined by the number of edges between these sets:

Definition 4.2 (Cut). The cut $cut : V^2 \to N$ is defined as follows:
$\forall C_1, C_2 \subseteq V, C_1 \cap C_2 = \emptyset$, and $cut(C_1, C_2) = |\{(v_1, v_2) | (v_1, v_2) \in E, v_1 \in C_1, v_2 \in C_2\}|$.

How can the quality of a clustering be measured? We want a small interface between clusters, i.e. the cut between clusters should be as small as possible. On the other hand, defining just one cluster $C_1 = V$ can not be called a great solution. Hence we demand the number of clusters to be maximized. However, we also want the clusters to be as dense as possible.

Definition 4.3 (Density of a cluster). The density $d : \mathcal{P}(V) \to \mathbf{R}$ of a cluster C_i is defined as follows: $d(C_i) = \frac{|\{(v_1,v_2)|(v_1,v_2)\in E, v_1,v_2 \in C_i\}|}{|C_i|}$.

Definition 4.4 (Density of a clustering). The density d_2 of a Clustering $\mathcal{C}$ is defined as follows:
$d_C(\mathcal{C}) = \frac{\sum_{C_i \in \mathcal{C}} |\{(v_1,v_2)|(v_1,v_2)\in E, v_1,v_2 \in C_i\}|}{\sum_{C_i \in \mathcal{C}} |C_i|}$.

Definition 4.5 (Cut of a clustering). The Cut cut_C of a Clustering $\mathcal{C}$ is defined as follows:
$cut_C(\mathcal{C}) = \sum_{C_i, C_j \in \mathcal{C}, i<j} cut(C_i, C_j)$.

Definition 4.6 (Connectivity of a clustering). The Connectivity con of a Clustering $\mathcal{C}$ is defined as $con(\mathcal{C}) = \frac{cut_C(\mathcal{C})}{|\mathcal{C}|}$

Obviously we can rewrite d_C as $d_C(\mathcal{C}) = \frac{|E| - cut_C(\mathcal{C})}{|V|}$.

Definition 4.7 (Quality of a clustering). The quality q of a clustering $\mathcal{C}$ is defined as $q(\mathcal{C}) = \frac{d_C(\mathcal{C})}{con(\mathcal{C})}$.

How can a cluster be found? Here we present a solution based on a greedy strategy. Starting with an arbitrary vertex within a cluster, the algorithm enters the cluster as deep as possible, i.e. it increases the cut of the respective cluster. In a second phase the algorithm walks through the cluster while the cut remains constant. In a last phase the cut is allowed to decrease, i.e. the algorithm tries to find the border of the cluster. Figure 5 illustrates the situation.

A special feature of component-property graphs supports the termination of the clustering: clusters are normally connected to a functionality or component with a small degree (see vertex v_2 in Figure 5). The algorithm does not cross this special vertex, therefore the clustering stops at the interface of components.

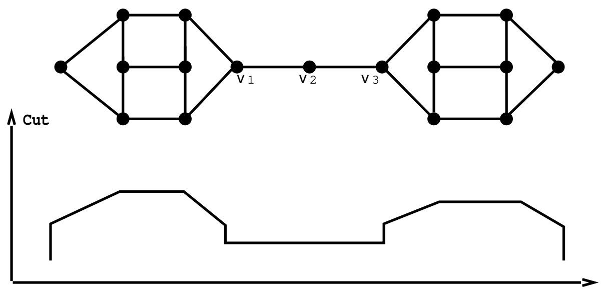

Figure 5: *Cut and cluster connectivity.*

A division of the graph by a vertical borderline would result in a cut as shown in the diagram below the graph. An algorithm intended to find clusters could exploit the “hill-appearance” of the clusters in the cut-diagram.

Note that not all vertices are suitable as start vertices for the clustering algorithm. The vertex should be within a cluster and not be part of an interface between components (in the example above v_2 is not suitable). So vertices with in-degree = out-degree = 1 are not used as starting points.

For some knowledge-bases a preprocessing step is useful: In some graphs there exist a few functionalities or components with an exceptional high degree, e.g. a functionality “power consumption”, which is demanded by almost all components. This functionality could feign a non-existing module and therefore make a cluster detection difficult. To overcome this problem vertices with an exceptional high degree should be deleted.

4.3 *The Cluster Arrangement Step*

Having identified the clusters, the width and height necessary for their graphical representation is estimated. Resting on empirical evaluations we developed heuristics for this task. We call this graphical representation of a cluster a box; two boxes are connected iff there exists at least one edge between vertices from the corresponding clusters.

Definition 4.8 (Box representation of a Clustering). Let $\mathcal{C}$ be a clustering of the graph $G = (V, E)$. The box representation $\mathcal{B}(C) = (V_B, E_B)$ of $\mathcal{C}$ consists of boxes $V_E = \{b_1, \ldots, b_n\}$ (each relating to a cluster) and of connections $E_B \subseteq V_B^2$, where $(b_i, b_j) \in E_B \Leftrightarrow \exists v_a \in b_i, v_b \in b_j$ and $((v_i, v_j) \in E \vee (v_j, v_i) \in E)$.

To arrange the boxes on the pane a local optimization strategy is applied: Starting from reasonable initial positions we allow random position changes, and if these moves result in a better layout, we restart the process with the new arrangement. Of course, in order to optimize the layout the notion "lucid layout" must be defined in mathematical terms. We do not want boxes to intersect; we also would like to avoid a box being between two connected boxes. Finally, connected boxes should be as close as possible. The following optimization function f formalizes the desired conditions:

Definition 4.9 (Quality of a box arrangement). Let $\mathcal{A}$ be an arrangement of boxes on the pane, then the quality f of the arrangement is defined as follows:

$$f(\mathcal{A}) = \begin{cases} \infty, & \text{if two boxes intersect---otherwise:} \\ \displaystyle\sum_{(b_i,b_j)\in E_B} dist(b_i, b_j) + \lambda \cdot \sum_{(b_i,b_j)\in E_B} num(b_i, b_j) & \end{cases}$$

where $dist(b_i, b_j)$ refers to the distance between two boxes, λ is a constant (we use $\lambda = \frac{\text{screen size}}{5}$) and $num(b_i, b_j)$ relates to the number of boxes between b_i and b_j. In Figure 6 and 7 the area between two boxes is shown hatched.

Initially we place the boxes on a circular arc (cf. Figure 6), in the course of the optimization process we decrease the step size of the movements.

Figure 7 shows the result of the optimization process.

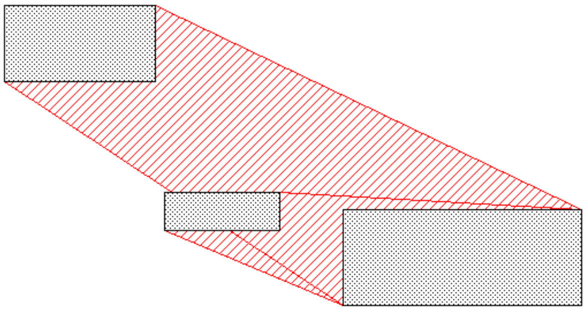

Figure 6: *Initial arrangement of the clusters.*

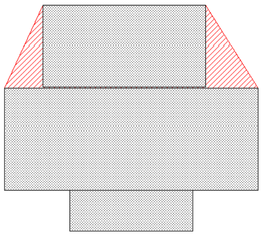

Figure 7: *After the cluster arrangement step.*

4.4 *The Leveling Step*

The last step of our visualization process determines the positions for the vertices within a box. We apply a modified leveling approach (Eades and Wormald, 1994; Sander, 1996). At first, the vertices of a cluster are divided into layers; here it is desired having an edge only between neighbored layers. Then the vertices of a layer are ordered, minimizing the number of edge intersections. Both the division into layers and the ordering within a layer take the global arrangement of boxes into consideration: A vertex v_1 in box b_1 connected to another vertex v_2 in a different box b_2 should have a position within b_1 close to b_2. Ghost vertices must be added for edges crossing layers. Finally the ordering of vertices of a layer is mapped onto Euclidean positions.

How can the division into layers be found? For acyclic graphs a topological sorting is sufficient, but for our mainly cyclic graphs the NP-complete feedback edge set problem needs to be solved. We overcome this problem by using a heuristic. For our domain it makes sense to allow only one type of vertex (functionality or component) being in a layer. Thus we find the components or functionalities with the smallest in-degree and choose one type to be placed as first layer. As a decision criterion the accessibleness of all other vertices starting from the vertices of the first layer is examined: The more vertices can be reached, the better. Unaccessible vertices are simply put into the appropriate of the first two layers. All other layers are derived canonically from the first layer. Vertices with connections to vertices in other boxes above (below) the actual box are also moved into the appropriate of the first (last) two layers.

How can the vertices within a layer be ordered? Here, the well-known barycenter heuristic is employed. In a first run, starting with the second layer, each vertex v obtains a number $bc(v) = \frac{1}{|N_p|} \cdot \sum_{u \in N_p} bc(u)$, where N_p are the predecessors of v belonging to the same cluster. The bc values for the first layer are initialized in a reasonable way. In a second run, going from the second last layer up to the first layer, each vertex obtains a new bc value: $bc(v) = \frac{1}{|N_s|} \cdot \sum_{u \in N_s} bc(u)$, where N_s denotes the successors of v belonging to the same cluster. This standard procedure is slightly modified; a vertex v_1 in box b_1 connected to another vertex v_2 in a different box b_2 left (right) of b_1 is attracted to the left (right) side of its layer. This is achieved by giving v_2 a virtual bc value of 0 or max_x, where max_x is the maximum number of vertices in a layer.

For each edge crossing a layer an additional ghost vertex is added that prevents edges from crossing vertices. These ghost vertices are treated like normal vertices.

The vertices in a layer are mapped onto real positions on the pane as follows. We place all vertices on a given grid, the y-position on the grid follows from the division of the vertices into layers, we also use the bc value as a initial value for the x-position and optimize the x-positions later. Our optimization criteria is the overall edge length in a cluster. The process starts at the first layer. In a first run, we go through the vertices of a layer from the left to the right. If moving a vertex v horizontally improves the overall edge length, we restart this process with the new x-position for v. After going through the layer from the left to the right, we rerun the procedure going from the right to the left. This way, free space created by the first run can be used to improve the positioning in the second run. Figure 8 shows the result.

4.5 *Realization*

Our approach has been implemented under Windows NT using Allegro Common Lisp. Note that recognizing the clusters and their interfaces is quite easy. Nevertheless, for a "creative" arrangement of the cluster much more time needs to be spent. This can be considered as acceptable since the process of visualization does happen only once when importing a textual knowledge base.

Since the runtime behavior is dominated by the cluster arrangement step, it can be scaled quite easily: The better the cluster layout shall be, the more time must be spent.

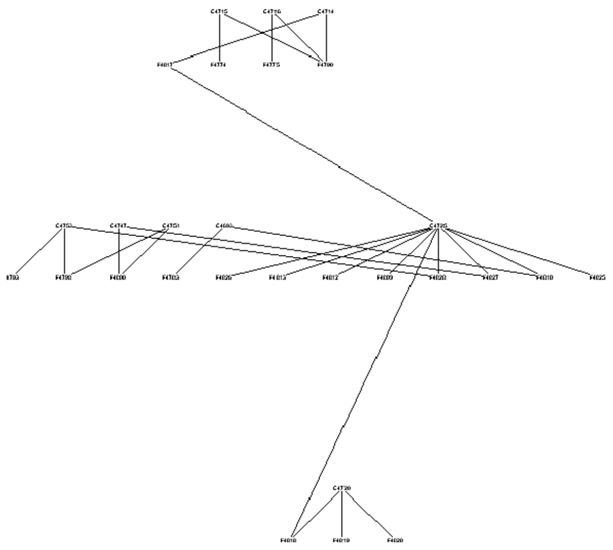

Figure 8: *After the cluster arrangement step.*

The knowledge-bases that have been used to evaluate our approach stem from real-world examples. These examples where analyzed respecting the distribution of the node types, the node degrees, the components' supply-demand overhangs, and the functionalities' supply-demand ratios. Using this statistical information new knowledge bases have been generated from the existing ones by merging, deletion, and upsizing rules.

5. EXISTING VISUALIZATION TOOLS

The next two subsections show layout results that have been produced with VCG and Da Vinci for the component-property graph in Figure 9. VCG and Da Vinci are universal graph visualization tools, and therefore they may exploit less domain knowledge about resource-based configuration graphs. Nevertheless, a comparison will give some insight into graph visualization. The last subsection, 5.3, is on AKON, a resource-based configuration tool that provides a graphical acquisition mode for component-property graphs.

5.1 *VCG*

The VCG tool was developed at the University of Saarbrücken for the purpose of visualizing compiler graphs. It is based on a leveling approach and can take

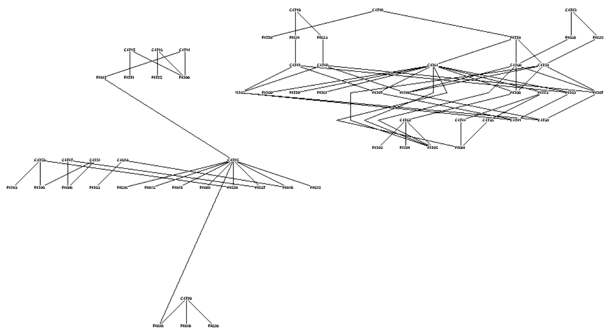

Figure 9: *Graph generated with our approach.*

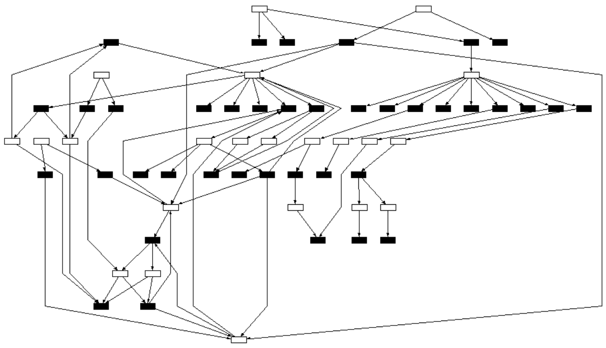

Figure 11: *Graph generated by Da Vinci.*

cluster information into consideration; these cluster information must be part of the input.

Opposed to our cluster arrangement step, the cluster visualization of VCG is intertwined with the leveling algorithm. This results in a fast runtime behavior, but confines the freedom for arranging the clusters.

Figure 10 shows the visualization of our example. For further information, please see (Sander, 1994) and (Sander, 1996).

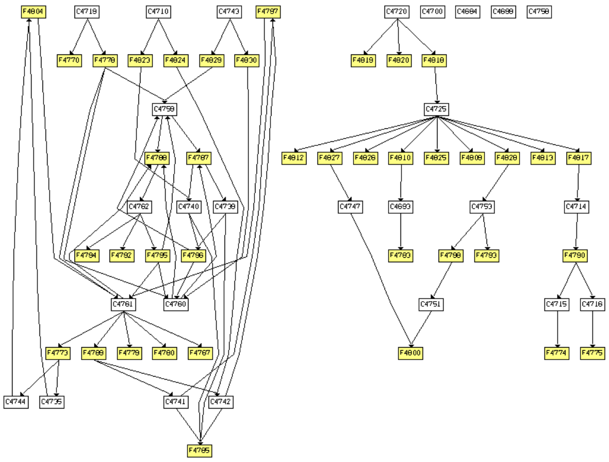

Figure 10: *Component-property graph generated by VCG.*

5.2 *Da Vinci*

Another popular tool for graph visualization is Da Vinci. Written at the university of Bremen, it is another realization of the leveling approach. Figure 11 shows the graph of Figure 10 and Figure 9 respectively, visualized by Da Vinci. Detailed information on Da Vinci can be found in (M. Fröhlich, M. Werner, 1994).

5.3 Akon

Akon is not graph visualization tool but a fully-fledged configuration system that operationalizes the resource-based configuration approach (Kleine Büning *et al.*, 1994).

Within Akon, configuration knowledge cannot be entered in textual form but is completely graphically specified, by means of drag-and-drop operations. Different types of lines indicate supply and demand relations respectively. A supply (demand) connection between a functionality icon, $\boxed{F}$, and a component icon, $\boxed{C}$, is established by dragging $\boxed{C}$ over $\boxed{F}$ ($\boxed{F}$ over $\boxed{C}$). In this way, complex knowledge bases, say component-property graphs, can be formulated efficiently while avoiding any syntactical mistake. Figure 12 depicts a section of a telecommunication knowledge base formulated using Akon.

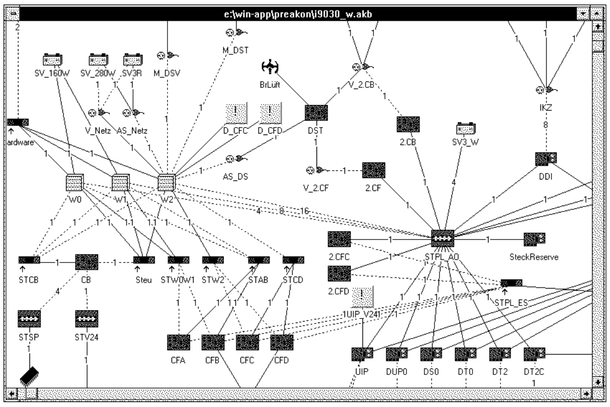

Figure 12: *A telecommunication knowledge base in* Akon.

Being in Akon's acquisition mode, the layout of the component-property graph can be organized reflecting both configurational aspects and the knowledge engineer's flavor. As a matter of concept, the component-property graph is not generated algorithmically but by the human sense for design and usefulness.

To us, the knowledge bases in Akon define a measurement respecting the quality of our automatically generated layouts. However, if all knowledge bases for resource-based configuration problems were acquired by direct building the component-property graph, there would be no need for their graphical post-processing.

6. CONCLUSION

Components with supplied and demanded functionalities form the backbone of each resource-based configuration problem. This backbone can be illustrated by means of a so-called component-property graph, where the set of nodes is formed by the components and functionalities, and the edges stand for supply and demand relations.

For different acquisition and maintenance tasks relating configuration knowledge-bases, a smart visualization of component-property graphs is very useful. The paper showed in which way and to which quality such a visualization can be performed automatically. The developed approach consists of three basic steps—cluster detection, cluster arrangement, and inside-cluster leveling.

Within the configuration domain, the clusters play a special role: they represent subsystems or modules of a larger technical system. Their detection and visualization thus gives insight into the technical structure of the system defined in the knowledge-base.

Our approach has been realized and compared to the graph visualization tools VCG and da Vinci. These visualization tools generate clear layouts but require the definition of clusters by the user, and the cluster arrangement is restricted to hierarchical layouts only.

REFERENCES

Eades, Peter and Nicholas C. Wormald (1994). Edge Crossing in Drawing of Bipartite Graphs. In: *Mathematica 1994*. Springer Verlag.

M. Fröhlich, M. Werner (1994). The Graph Visualization System daVinci - A User Interface for Applications. Technical Report No. 5/94, Dept. of Computer Science, University of Bremen.

Heinrich, M. and E. W. Jüngst (1991). A Resource-based Paradigm for the Configuring of Technical Systems for Modular Components. In: *Proc. CAIA '91*. pp. 257–264.

Kleine Büning, Hans, Daniel Curatolo and Benno Stein (1994). Knowledge-Based Support within Configuration and Design Tasks. In: *Proc. ESDA '94, London*. pp. 435–441.

Laußermair, T. and K. Starkmann (1992). Konfigurierung basierend auf einem Bilanzverfahren. In: *6. Workshop "Planen und Konfigurieren", München*. FORWISS, FR-1992-001.

Sander, Georg (1994). Graph Layout through the VCG Tool. Technical Report A/03/94.

Sander, Georg (1996). Layout of Compound Directed Graphs. Technical Report A/03/96.

Stein, Benno (1995). Functional Models in Configuration Systems. Dissertation. University of Paderborn, Department of Mathematics and Computer Science.

Stein, Benno and Jürgen Weiner (1991). Model-based Configuration. In: *OEGAI '91, Workshop for Model-based Reasoning*.

Tong, Christopher (1987). Towards an Engineering Science of Knowledge-based Design. *Artificial Intelligence in Engineering* **2**(3), 133–166.

Weiner, Jürgen (1991). Aspekte der Konfigurierung technischer Anlagen. Dissertation. Gerhard-Mercator-Universität - GH Duisburg, FB 11 Mathematik / Informatik.