Abstract
This paper discusses a clustering and layout approach for representing certain types of directed graphs in an object notation. The approach is particularly interesting when applied to program call graphs. An object-oriented abstraction of a call graph simplifies the diagram while highlighting the shared attributes among the programs.

General problem description
Diagrams representing software systems are often quite large. The size and complexity of the diagrams pose problems for the tool developer who is trying to produce the diagram as well as the end-user who tries to interpret it. This paper will discuss and propose solutions for some of the issues affecting tool developers and end users.

Directed graphs are used to represent several different kinds of information, such as control flow within a module and calling relationships between modules. Different kinds of layout and abstraction schemes may be needed to assist the user in perceiving the underlying structure of the data being represented. Abstractions which work on control flow graphs are often not useful for graphs representing calling relationships:

- Experimentation has shown that interval reduction is effective for control flow graphs, but it has proved to be not helpful for graphs showing calling relationships. Calling diagrams frequently contain disconnected subgraphs, multiple entry points and irreducible cycles which make them ill-suited for interval reductions.

An object representation seems to be better suited for abstracting calling relationships, particularly in systems programs such as operating systems. These systems are characterized by severely unstructured, irreducible calling patterns. Our interval reduction provides a bottoms-up abstraction which mirrors the top-down refinement process programmers often used to develop program logic. In contrast, programmers often abstract calling relationships by functionality. This is evident when several subroutines are grouped into a module, and several modules are represented by a sub-system call. In other words, logic abstractions generalize the conditions which lead to an action while functional abstraction generalizes the definitions of the action to be performed. Abstraction of a logic flow results in a coarser perception of the decision detail. Abstraction of calling relationships should result in a coarser view of the functions being performed.

- Layout schemes should be designed to highlight the important information being presented in the graph. The top-to-bottom, fan-out style of a control flow diagram emphasizes the sequential nature of the logic. One of the key purposes of a call graph is to show the relationships between various functional entities. Therefore, the layout scheme should highlight the calling patterns between groups of related programs.

This suggests that layout and abstractions schemes need to be under user control in some fashion, although defaults should be used for most situations.
Figure 1 illustrates some key points of our object oriented layout. Figure 1a shows a calling diagram using a traditional directed graph layout. In Figure 1B, each box represents an object and the connecting arcs show calling relationships. Boxes contained within another box are sub-objects. Sub-objects inherit all the attributes (in this case calling relationships) of the containing object(s) and may have additional individual attributes. A bold arc indicates the concentration of several arcs. The bold arcs originating at D and at the container for BC indicate calls to A, B, and C.

We found it necessary to constantly make some tradeoffs due to both practical (usability) and technology constraints in both the layout and display mechanisms. Following is a brief description of some of the constraints:

- Usability and space were constantly in conflict. The font sizes were carefully adjusted to provide readable text while minimizing the size of the nodes. The solutions were different for interactive displays and hardcopy output. Hardcopy problems were further aggravated because many user sites have different printer configurations. Our layout algorithm provided for adjustment of node spacing (horizontal and vertical), minimum distances between nodes and edges, and the maximum angle for an edge.

- We were constrained to a fixed physical layout size corresponding to a commercial cad/cam plotter that printed thirty-five inches wide and up to fifteen feet long. People frequently hung the diagrams on walls and the practical size limit was the height of a typical office wall, or about eight feet. We experimented with rotation and alignment of nodes. The most effective orientation was a vertically aligned diagram up to seven feet long. This diagram was typically placed on a wall and used as a system overview or reference diagram.

Interactive users had to manage with the restricted viewing area in our windowed environment (OS/2 Presentation Manager™). Features such as zoom and overview windows were essential to enable users to understand the size of the entire diagram and to understand context of a particular portion of the diagram.

- It was important that we show all data in the diagram. This meant that if any data was abstracted or compressed from an overview diagram, then a reference was required to identify where there was hidden data.

- The diagram was to be printed on the fewest number of pages practical. One early option was to partition the graph into regions and print the detail of each region on a separate page. However, we were not able to find a a partitioning algorithm which produced a minimum number of uniformly sized partitions that provided any structural insights.

- We wanted to minimize visual complexity. Visual complexity has many factors, but can be thought of as the "busi-ness" of the graph. Although a few line crossings seem to be acceptable, excessive line crossing makes it difficult to understand a graph.
• Several technical constraints made it desirable to reduce the size of the layout problem by breaking the graph into smaller pieces and processing the pieces one at a time. Since layout can be exponentially proportional to the number of nodes and arcs being processed, breaking the graph in smaller pieces and laying out several smaller graphs may be substantially faster than laying out one large graph. Some researchers have reported as much as a 5X improvement in layout time by partitioning the original graph.

These points, and others, were important design considerations which affected the performance and usability of our graphical interface. The next section will provide a general description of the approach suggested for solving the problems and the following section will give some descriptions of the algorithms and heuristics.

**Approach**

This approach uses a clustering algorithm to identify a set of nodes and/or edges that share a set of common attributes.

Clustering is a form of abstraction that is especially useful for structure specification. It involves grouping entities, which have some important common properties, into a cluster, associating those properties with the cluster itself, and then regarding the cluster as an atomic entity whenever possible. Interactions between members of clusters are specified in terms of the clusters themselves.

Clustering enables users to associate characteristics or attributes with subsets of the entire graph being examined. Packaging the clusters into subgraphs and separating the subgraphs geographically provides additional help in summarizing complex structures and understand modularity. Additionally, some connections between clusters can be collected into edge concentrations, providing even more visual clarity for end users.

Our approach consists of several phases:

1. Build Subgraph Objects and Hierarchy Trees.
   Nodes are grouped into objects and sub-objects using a clustering algorithm. A hierarchy tree records the containing relationships between objects. Traversing the hierarchy tree enables us to determine the size of each object. Object size includes space allocation for the text associated with each object, space for the bold container boundary, plus a buffer space surrounding the object.

   A metagraph is built showing the relative positions of the outermost objects, or meta-objects, and the connections between them. The arrangement is adjusted to accommodate the actual height and width of each meta-object.

3. Subgraph layout
   The contents of each meta-object are laid out as subgraphs, including arcs between objects contained in the same meta-object.

4. Routing of subgraph connections.
   Arcs with endpoints in two different subgraphs are added to the final layout.

**Build the subgraph objects**

The algorithm described below works by grouping programs (source programs) that call the same program (target programs). The clustering scheme is modeled after the concept of an object-class hierarchy. Related programs are grouped into objects and sub-objects. Sub-objects have all the attributes of the containing object as well as any additional attributes which are specific to the sub-object.

We use lists to describe each node, and its successors and predecessors, that is, nodes which are called by or which call the node. A second type of list identifies the source (SRC) and target (TGT) intersections. Each node is compared to the set of SRC and TGT intersections. A list of objects is updated to reflect the intersection of a node with the sets of SRC and TGT nodes. A hierarchy tree describes relationships between sub-objects and their containing objects.

**Data Structures**

- **pred(y)** For each node y, pred(y)=x is a list of predecessors such that (x,y) is an arc in the graph.
- **succ(x)** For each node x, succ(x)=y is a list of successors such that (x,y) is an arc in the graph.
- **SRC(a,b,...,n)** For any set of nodes (a,b,...,n), SRC identifies the intersection of their predecessors. SRC(a,b,...,n) = pred(a) \(\cap\) pred(b) \(\cap\) pred(c) \(\ldots\) \(\cap\) pred(n).
- **TGT(a,b,...,n)** For any set of nodes (a,b,...,n), TGT identifies the intersection of their successors. TGT(a,b,...,n) = succ(a) \(\cap\) succ(b) \(\cap\) succ(c) \(\ldots\) \(\cap\) succ(n).
- **Contained_in(x)** For any object, SRC(x) or TGT(x), Contained_in(x) identifies the container object, if any.

**Algorithm**

1. Initialize SRC and TGT to the empty lists.
2. Sort and number the nodes by decreasing number of successors. This will reduce the amount of splitting and joining to be done in later steps.
3. Compare each node(i) with all TGT(j)
   a. If succ(i) = TGT(j) then
1) add node(i) to the set of source nodes in TGT(j).
2) If succ(i) is a wholly contained subset of TGT(j) then
   a) create a new set TGT(i) = succ(i).
   b) Contained_in(TGT(i)) = TGT(j) to show TGT(i) contains TGT(j).
   c) if TGT(j) is a wholly contained subset of TGT(i) then
      1) create a new set TGT(i) = succ(i).
      2) Contained_in(TGT(i)) = TGT(j) to show TGT(i) contains TGT(j).
   d) if neither succ(i) nor TGT(j) is wholly contained by the other, but there is an intersection, then
      1) Split succ(i) and TGT(j) into succ(i') and TGT(j'), and succ(i') \cap TGT(j)
      2) create a new set TGT(i')
      3) Contained_in(succ(i') \cap TGT(j)) = TGT(j) because it is a subset of TGT(j)
      4) Contained_in(TGT(j')) = TGT(j) because it is a subset of TGT(j)
   e) if there is no intersection between succ(i) and any TGT(j) then
      1) create a new TGT(i) = succ(i) \cap TGT(j)
      2) create a new set TGT(i')

The process is repeated with predecessors, pred(i) and SRC(j), to establish source groupings.

A final pass compares the intersection sets SRC and TGT to build a single set of objects. Sometimes there will be an intersection of nodes between some SRC(i) and TGT(j). A new object is created to resolve the conflict. The following data sets derived from the diagram in Figure 1 on page ii illustrate this situation:

```
succ(A) = 0, pred(A) = BCD
succ(B) = ABC, pred(B) = BCD
succ(C) = ABC, pred(C) = BCD
succ(D) = ABC, pred(D) = 0
TGT(BCD) = ABC, SRC(ABC) = BCD
```

Objects: D, Contained_in = 0, arc_summary = ABC
Objects: ABC, Contained_in = 0, arc_summary = 0
Objects: A, Contained_in = ABC, arc_summary = 0
Objects: BC, Contained_in = ABC, arc_summary = ABC

An objective of the layout is to emphasize the connections between groups of nodes, so arcs which connect multiple entity objects will be highlighted by a bold line. Arcs between single entity objects will be connected with a thinner, less noticeable line type.

Placement of the objects may be constrained by the size of the objects. Connected objects should be placed first, then all disconnected objects may be added in available spaces.

In a control flow diagram, the nodes with no predecessors represent entry points of the program and are often placed at the top of the diagram. In many cases, there is no useful concept of a "first" module in a call graph. Instead objects will be placed as close as possible to related objects; in other words, a goal of the layout is to minimize the length of the sum of all arcs, with bold lines representing multiple arcs having a greater weight than arcs between single entity objects. A technique to help do this is that arcs connecting multiple-entity objects have a shorter minimum length than arcs connecting single-entity objects.

Traverse the hierarchy to determine the "size" of each object. Size is:

- the sum of all sub-objects contained in each meta-object,
- plus room for the container boxes,
- plus a buffer space to surround each object.

Layout of metagraph

We want to highlight the relationships between the most prominent objects, so the layout algorithm will first place the outermost container objects, or meta-objects. Sub-objects will be added in a later step. Furthermore, spatial emphasis will help show degree of relationship. Objects which have caller-callee relationships will be closer than objects which are separated by an intervening called object and objects which are not related by any calls are likely to be separated even farther. Each meta-object will be enclosed in a bold box to emphasize the object boundary. Disconnected objects will have a larger buffer space surrounding them than connected objects.

In order to achieve proper placement of the meta-objects, it is necessary to consider the connections between meta-objects, including connections between sub-objects in different meta-objects. This information is summarized by traversing the hierarchy tree and collecting information about all connections. Information from the example is shown below:

```
Objecti = D, Contained_in = 0, arc_summary = ABC
Objectz = ABC, Contained_in = 0, arc_summary = 0
Objecta = A, Contained_in = ABC, arc_summary = 0
Objectb = BC, Contained_in = ABC, arc_summary = ABC
```

The intersections of the successors is TGT(BCD) and the intersection of the predecessors is SRC(ABC). Since TGT(BCD) and SRC(ABC) have some elements in common, it necessary to create a new object containing that intersection. The result is that object A and object BC are contained in object ABC.

161
Subgraph layout

The subgraph details for each meta-object are drawn in this phase. Subgraph layout is similar to the metagraph layout, with some additional constraints. We chose to use a single column format for the subgraph since width was more of a problem than overall length. The first step arranges all the connected subgraphs with a goal of minimizing the sum of arc lengths within each subgraph. Connections from external objects will cause the sub-objects to be placed in the top, center, or bottom of the meta-object. Following an initial, trial placement the layout is adjusted to eliminate line crossings and to shorten total arc length. Bold container boxes and the text for each sub-object are placed in the containing objects.

Routing of subgraph connections

Arcs between sub-objects in different meta-objects are added. In some cases, it may be necessary to rearrange sub-objects within a meta-object in order to minimize the arc lengths. In the event that rearranging sub-objects disturbs intra-object connections, minimizing arc length within a subgraph takes precedence over arcs between meta-objects.

Examples

Figure 2. Calling diagram of compiler code fragment
Figure 3 shows the same calling data displayed in an object-oriented layout.
Figure 4. Calling diagram of financial transaction
Figure 5. Object layout of transaction calling diagram

Figure 4 on page vii shows a call graph from a commercial financial system displayed using a traditional style layout. Figure 5 shows the same calling data displayed in an object-oriented layout.

Acknowledgements


Bibliography