MTR—A tool for displaying the global structure of software systems

by GUY de BALBINE

Caine, Farber & Gordon, Inc.
Pasadena, California

INTRODUCTION

The purpose of this paper is to propose a new computer based method for displaying the global structure of software systems. We are particularly motivated by finding a manageable representation of the directed graph (digraph) of calls within a design or an executable program, each directed edge representing one or more calls from a procedure to another. We expect such a representation to provide a better understanding of the relationships between various components of a software system and to lead to the design of systems that are better organized.

The directed graph of calls within a program is an example of what we call invocation digraphs. Invocation digraphs are a particular class of directed graphs with at most one directed edge from one vertex to another or to itself. These invocation digraphs constantly arise during the design, implementation and maintenance of software systems.

Usually, invocation digraphs are drawn by hand if they are drawn at all. “Structure charts” provide the most familiar example of a graphic representation which is visually informative. However, such a technique is of limited value in practice because of the considerable effort required to draw even a graph of modest size—and large graphs are the most interesting because they are the least understood. One of the major difficulties is deciding where to position adequate size boxes on the page so that the resulting structure becomes visually informative instead of hopelessly confusing. Often, a judicious positioning of the boxes becomes apparent only after the first charting is essentially complete. Even if a good layout is obtained, the charts remain cumbersome and expensive to maintain and modify.

One obvious solution is to try mechanizing the whole drafting process. This certainly alleviates the manpower problem but leaves other problems still unsolved—how to dimension and position boxes, draw edges that do not cross over at random, handle the off-page connector problem and cope with the lack of adequate graphic devices at many installations.

As an alternative, we propose an extension of the simple yet powerful indented representation of trees, capitalizing on the observation that most invocation digraphs of interest are loop free or almost loop free. We call this one-dimensional, indented representation the Modular Tree Representation (MTR) of the invocation digraph.

In an MTR, the position of each node (procedure, subroutine, ...) is chosen systematically to reflect its relationship with other nodes in the structure. Although the connections are not explicitly materialized by lines, the specific location of each node in the MTR implies to the reader that certain connections must exist. Thus, the MTR emphasizes the global structural properties of the digraph while relegating specific path connections to a secondary role. We believe this approach to be practical because invocation digraphs are most often used to grasp a morphological “understanding” of the structure of a software system rather than to trace through many individual paths.

DEFINITION OF INVOCATION GRAPHS

We start with a set of components numbered 1 through n. Components is a loosely defined term which will mean subroutines, procedures, design segments, ..., depending upon the context.

Let us construct the invocation graph G as follows:

- create for each component an associated vertex, e.g., \( v_i \) corresponds to the ith component;
- for all pairs of vertices \( (v_i, v_j) \) draw a directed edge from \( v_i \) to \( v_j \) if there is at least one invocation of the jth component from the ith one.

In some simple cases, the invocation graph may turn out to be a tree but, in general, we expect a directed graph, some of its vertices having indegrees greater than 1. Invocation graphs are a subset of directed graphs since there can be at most one edge between any pair of vertices, distinct or not. Notice that invocation digraphs have loops if co-routines or recursively callable components are present.

We shall use the \( n \times n \) adjacency matrix A whose element \( A(i,j) \) has the value

\[
\begin{align*}
1 & \text{ if there is a directed edge from } v_i \text{ to } v_j \\
0 & \text{ otherwise}
\end{align*}
\]
The reachability matrix $R$ is an $n \times n$ matrix such that its element $R_{ij}$ has the value

1 if there exists a directed path, i.e., a sequence of directed edges from $v_i$ to $v_j$, 
0 otherwise

Invocation digraphs usually have only one begin node $v_1$, i.e., $v_1$ is the only vertex in $G$ with indegree 0. But they may have several or even none at all if every component is at least invoked once.

If several vertices with indegree 0 exist and/or the graph is not at least weakly connected with begin node $v_1$, i.e., the corresponding undirected graph is not connected, we create another vertex $v_0$ and draw a directed edge from $v_0$ to each node with indegree 0. We then determine if all vertices are reachable. If some vertices are still not reachable from $v_0$, we select any such vertex, connect $v_0$ to it with a directed edge and compute the reachability again. This process repeats until all vertices are reachable from $v_0$. Although this algorithm clearly forces the graph to become weakly connected (which is our requirement), it is inadequate in practice. In the appendix, we shall present some refinement to this algorithm guaranteeing that the number of vertices artificially connected to $v_0$ is minimal. At any rate, we can now renumber the vertices so that $v_0$ becomes $v_1$ and so forth. Thus, we have obtained a weakly connected invocation graph with begin node $v_1$. Without loss of generality, we can, therefore, turn our attention to weakly connected invocation digraphs with begin node $v_1$ only.

MTR PROPERTIES

The MTR is an extension of the common indented representation of trees. By design, the MTR becomes the preorder representation if applied to a tree. In that case, the tree is traversed from left to right, visiting descendants first. Each vertex traversed is listed on a new line, horizontally indented to show its distance from the root. An example is shown in Figure 1.

The purpose of the square bracket to the left of $v_1$ is to make the scope of $v_1$ visible, i.e., to identify the set of vertices that can only be reached by traversing $v_1$. Since $v_1$ is the root of the tree, every vertex is enclosed in the outermost and only bracket. The role of brackets will become obvious when we examine more complex MTR examples later.

In practice, invocation digraphs are rarely as simple as trees. As soon as a procedure is called from two distinct procedures, the invocation graph no longer is a tree. Due to the non-recursive nature of most software systems being built, we expect invocation graphs to be acyclic or almost acyclic.

This observation is crucial for the MTR. Notice that it is simply a paraphrase of the fact that most software systems are hierarchical in nature. They have a top and a bottom—high level components and low level ones. Thus, we can expect a representation that displays these hierarchical properties to be useful both at the global and local levels.

To introduce the MTR of a non-trivial digraph, let us informally examine Figure 2. Looking at part 1 of Figure 2 first, we have separated the subgraph into four disjoint trees by breaking the incoming edges to vertices with indegrees greater than 1.

The four subtrees rooted at vertices 1, 6, 9 and 11 each have an associated bracket in the MTR. These brackets allow the following “understanding” of the graph:

- 1 calls 2, 3, and 4 which are called by no one else.
- 2 calls 5 and 6, but 6 is called elsewhere
- . . .
- the trees rooted at 6 and 9 are referred to by the tree rooted at 1 (because they are in the broken part of the bracket associated with 1)
- the tree rooted at 9 is referred to by the tree rooted at 1 but also by the tree rooted at 6 (since it is in the broken part of the bracket associated with 1 but also indented to the right of 6)
- the tree rooted at 13 is only referred to by the tree rooted at 9 (since it is in the broken part of the bracket associated with 9)

In particular, we can infer from the MTR that the tree rooted at 13 is not known outside the “scope” of 9. Similarly, the trees rooted at 6 and 9 are not known outside the “scope” of 1. This property of not being known outside the “scope” of a particular vertex is fundamental for using the MTR. Each square bracket shields the vertices that are defined within it, except the root, from outside references, i.e., there are no incoming edges except possibly to the root. On the other hand, references from within the square bracket to outside vertices are freely permitted.

In order to arrive at this MTR, we have made use of the dominance concept. It can be summarized as follows: in a weakly connected graph with a begin node of $v_1$, we say
that \( v_i \) dominates \( v_j \) if \( v_i \neq v_j \) and every path from the begin node \( v_i \) to \( v_j \) contains \( v_i \).

We say that \( v_i \) directly dominates \( v_j \) if:

(i) \( v_i \) dominates \( v_j \);
(ii) if \( v_k \) dominates \( v_j \) and \( v_k \neq v_i \), then \( v_k \) dominates \( v_i \).

One of the important properties of the dominance relationship is that every node except the begin node (which has no dominators), has a unique direct dominator.

In the example shown in Figure 2, the subtrees rooted at 6 and 9 are located within the scope of 1 because vertices 6 and 9 have 1 as their common direct dominator. Similarly, the subtree rooted at 13 is located within the scope of 9 because 9 is the direct dominator of 11 whereas 1 is only a dominator of 11.

By placing the definition of each subtree within the scope of the direct dominator of its root, we guarantee that each subtree is enclosed within the most restrictive scope possible. This rule provides much of the power obtainable from the MTR.

In Figure 2, we saw examples of square brackets drawn with either a solid or broken line and defining two regions:

—a solid region containing the subtree attached to the root \( v_i \) and obtained by only traversing vertices with indegree less than two (i.e., with at most one predecessor). Notice that the vertices enclosed in angle brackets \( \langle \) seem to violate that rule but these are only visited, not traversed. We call the solid region associated with \( v_i \) the primary scope of \( v_i \).

—a broken region containing the subgraphs whose roots all have \( v_i \) as their direct dominator and indegree at least equal to 2. We call this broken region the secondary scope of \( v_i \). In general, there may be not just one but several secondary scopes of a given vertex. Two subgraphs are in distinct secondary scopes of \( v_i \) if they are disconnected within the scope of \( v_i \).

Within one of the secondary scopes of a given vertex, we expect to have \( p \) bracketed subgraphs, \( p \geq 0 \). Let us designate by \( s_1, s_2, \ldots, s_p \) the roots of these \( p \) subgraphs and let \( L(s_i) \) be the indentation level of \( s_i \). By construction, we require that:

\[
L(s_i) > L(s_j) \quad \text{if} \quad R(s_i, s_j) = 1 \wedge R(s_j, s_i) = 0
\]

\[
L(s_i) = L(s_j) \quad \text{if} \quad R(s_i, s_j) = 1 \wedge R(s_j, s_i) = 1
\]

To make the solution practical, we also require that \( \max_i (L(s_i)) \) be minimum.

This indentation strategy has the advantage that for an acyclic subgraph, the root of each subgraph is further indented than the root of any subtree that references it. Notice that if two roots \( s_i \) and \( s_j \) are at the same indentation level, then

\[
(R(s_i, s_j) = 0 \wedge R(s_j, s_i) = 0) \vee (R(s_i, s_j) = 1 \wedge R(s_j, s_i) = 1)
\]

i.e., either they call each other recursively, possibly via some indirect paths, or neither calls the other. As we mentioned earlier, most invocation digraphs of interest are acyclic or almost acyclic so that we expect most subgraphs that end up at the same indentation level not to reference each other.

Whether or not there are any recursive invocations can be quickly determined by scanning the definition/reference lists in the right hand margin of an MTR. Whenever a vertex with indegree greater than 1 is referenced (i.e., vertices in \( \langle \) ), we append the MTR line number at which its definition is located. Conversely, whenever a vertex with indegree greater than 1 is defined, we append the list of MTR lines at which it is referenced. If a vertex with indegree greater than 1 happens to be on a cycle,

From the collection of the Computer History Museum (www.computerhistory.org)
a '*' precedes any MTR line number at which it is either defined or referenced.

Figure 3 shows an example of MTR containing recursive invocations. The reference lists contain '*' to indicate recursive paths and ']' to separate references located within the primary scope from those in the secondary scope. For instance:

```
REF AT (19) 72,228
  primary  secondary
  scope    scope
  references references
```

Finally, each reference line can be supplemented by a left arrow showing which secondary scope contains the corresponding definition. An example is shown on Figure 4.

**CONCLUSION**

We have introduced the idea of Modular Tree Representation (MTR) of invocation digraphs. The algorithms necessary to build MTR's in practice are presented in Appendix A. The MTR provides a canonical representation for invocation digraphs.
Figure 3—Partial MTR of the FORTRAN 77 syntax
Figure 4—MTR with arrows

From the collection of the Computer History Museum (www.computerhistory.org)
In order to experiment with the MTR concept on real projects we have built an omnibus MTR processor. With appropriate input/output interfaces, it can be used to process invocation graphs generated from PDL designs, HOL source code, load modules. . . . We have already used the processor in the following contexts:

— as an extension of the PDL processor to help understand the morphology of a software design. The invocation graph is, in that case, the graph of segment references in a PDL design.

— as an aid in packaging software, particularly when grouping low level design segments into modules. The MTR provides an intrinsic decomposition into modules from which the actual module structure can be selected.

— as an analyzer of existing packages to help understand the hierarchy of calls to procedures, references to external structures. . . . An executable load module can be processed automatically to yield the MTR of all external references.

— as an aid in designing overlay structures by applying the MTR to the nonoverlay load module.

Based on our experiments so far, we can make the following observations:

• MTR's should be used during software development, particularly during the design phase. We have found that even a cursory examination of the computer generated MTR of a design could trigger design changes aimed at making the resulting MTR more "structured" i.e., more simply nested. For instance, realizing that a "GET NEXT CHARACTER" design segment was not nested within "GET TOKEN" simply because of an early initializing reference, suggested making "GET NEXT CHARACTER" a self-initializing module.

• MTR's can be effectively used during maintenance to help assess the impact of a change on an existing package. In that case, the MTR processor is applied to the executable load module directly.

• Practical MTR's can only be computer generated. This is due to the size of practical applications (the load module of the MTR processor itself has 90 vertices and almost 600 edges) and also to the high sensitivity of the MTR to small changes in the input graph.

• MTR's should only be computer maintained. This can be easily achieved because MTR's can be produced cheaply (for instance, the MTR of the MTR processor only takes 8.85 sec of computer time on an IBM 370/158 and a few thousand lines of print). Therefore, a new MTR can be produced every time a design change takes place or a new load module is generated.

There are several questions about the MTR which have not been answered yet. The first important question is to determine user acceptance of such a tool. Admittedly, an MTR is not as intuitively obvious as a graphical structure chart. It takes a little practice to learn how to read an MTR, i.e., to visualize the graph through the MTR and be able to use the MTR to reason about the graph. The second question is to investigate modifications and/or extensions which might make the MTR even more useful in practical situations. These may involve modifications of the display layout or extensions of the MTR capabilities. For instance, one might try visualizing the implicit scope of data items by indicating the direct dominator which controls all references to a given data item. Then, there is the fascinating possibility of applying the MTR to nonconventional areas. For instance, Figure 3 showed the partial MTR of the FORTRAN 77 syntax.

Clearly, further experimentation is needed to answer those questions and assess the full potential of the MTR technique.

BIBLIOGRAPHY


APPENDIX

A1 Construction of the MTR representation

This section deals with the algorithms required for generating MTR's. The construction of the MTR of an invocation graph involves primarily:

• finding the direct dominator of every vertex in G;

• performing an extended topological sorting of a digraph, acyclic or not;

• selecting entry points to force the invocation digraph to become weakly connected.

These algorithms are either classical or extension of classical ones.
It is important to notice that there exist several alternative formulations for most of the algorithms described here, depending upon the choice of data structure representation. For the MTR, we have chosen to use bit vectors and bit matrices rather than linked lists for representing the graphs and their ancillary data structures. Consequently, these algorithms are oriented towards parallel bit vector operations. For instance, for finding the descendants of a particular graph node, we rely on a bit vector “fusion” algorithm instead of the more familiar, stack oriented, depth first search algorithm.

A2 Computation of Direct Dominators

The algorithm for the computation of direct dominators can be found in Reference 1, for instance. Using PDL, the algorithm can be informally written as follows:

- COMPUTE DIRECT DOMINATORS (DIRECT_DOM())
  LET THE VERTICES BE NUMBERED 1, 2, ..., N WITH 1 THE BEGIN NODE
  INITIALIZE DIRECT_DOM(I) = 1, I = 1, 2, ..., N
  DO FOR EACH VERTEX I EXCEPT 1
    DELETE TEMPORARILY VERTEX (I)
    BUILD LIST OF DESCENDANTS(D(I)) REACHABLE FROM VERTEX (I)
    DO FOR EVERY VERTEX J IN D(I)
      IF DIRECT_DOM(I) = DIRECT_DOM(J)
        SET DIRECT_DOM(J) = I
      ENDIF
    ENDDO FOR
    RESTORE VERTEX (I)
  ENDDO FOR

A3 Extended Topological Sorting

We now examine the algorithms required to determine the indentation level of a group of p shared subtrees having the same direct dominator. A directed graph G' with vertices \( \{v'_1, v'_2, ..., v'_p\} \) is built as an invocation digraph where the p components would be the p shared subtrees. In other words, there is an edge between \( v'_i \) and \( v'_j \) if and only if the root of the jth subtree is referenced from within the ith subtree. Notice that G' is not in general weakly connected, but this is not required by the algorithm that follows.

The problem is to partition the set of vertices \( \{v'_1, v'_2, ..., v'_p\} \) into the smallest number of disjoint subsets \( S_1, S_2, ..., S_k \) such that

(a) if there is a path from \( v'_1 \) to \( v'_j \) but not from \( v'_j \) to \( v'_1 \) then

\[ v'_i \in S_\nu, v'_j \in S_\mu, 1 \leq \nu < \mu \leq k \]

(b) if there is a path both from \( v'_1 \) to \( v'_j \) and \( v'_j \) to \( v'_1 \) then

\[ v'_i, v'_j \in S_\nu, 1 \leq \nu \leq k \]

If \( v'_i \in S_\nu \), the rank of the ith vertex is said to be \( \nu \). In the acyclic case, this problem is a simple topological sorting and we know that condition (a) above is satisfied after the sort is performed. But if G' contains cycles, the classical topological sorting algorithm cannot be applied. We shall, therefore, extend it to handle cyclic graphs as well.

In the simple topological sort of an acyclic graph, vertices and their outgoing edges are deleted immediately after they have been processed, and one of the remaining vertices with zero indegree is selected as the next vertex to be processed. It is equivalent but computationally simpler to replace the deletion operation by the assignment to each vertex of a rank, initially 0. Thus, the search for a vertex with 0 indegree becomes the search for a vertex whose predecessors all have a non-zero rank.

- TOPOLOGICAL SORTING (ACYCLIC GRAPH)
  LET THE VERTICES BE NUMBERED 1, 2, ..., N
  INITIALLY SET RANK(I) = \( \infty \), I = 1, 2, ..., N
  DO WHILE THERE ARE VERTICES WHOSE RANK() = \( \infty \) STILL
    DO FOR EACH VERTEX I
      IF RANK(I) = 0 THEN
        COMPUTE PRED(), THE SET OF IMMEDIATE PREDECESSORS OF I, EXCLUDING I
        SET MXRNK = \( \max(\text{RANK(PRED(I)}, \text{RANK(PRED(2)}), ... \) )
        IF RANK \( \neq \infty \) THEN
          RANK(I) = MX RNK + 1
        ENDIF
      ENDIF
    ENDIF
  ENDDO FOR
  ENDDO WHILE

From the collection of the Computer History Museum (www.computerhistory.org)
To handle graphs with cycles, we modify the search strategy so that if the search for a vertex whose predecessors all have a non-zero rank fails, we then search for a maximal circuit whose predecessors, excluding nodes on the circuit, all have a non-zero rank. A circuit is maximal if it is not possible to find another circuit that includes at least one of its vertices as well as some other vertex that it does not already contain. The algorithm can be carried out in two steps:

—replace each maximal circuit by a single substitute vertex having the same immediate predecessor and immediate successor vertices, thus obtaining an acyclic graph but with self loops;
—apply classical topological sorting to the derived acyclic graph, reflecting the rank assignment to every vertex of a maximal circuit when its substitute vertex is assigned a rank.

This approach provides both a proof and a computational method.

In order to find the maximal circuits we can use the following method:

We start with the simple adjacency matrix $A = (a_{ij})$ defined

```
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
|   A | S | E | D | E | A | T | M | O | N | H | C | F | C | V | T | H |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
|   2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
```

Figure 5—Partial MTR of an IBM 370 load module

From the collection of the Computer History Museum (www.computerhistory.org)
by

\[ a_{ij} = \begin{cases} 
1 & \text{if there exists one or more directed edges from vertex } v'_i \text{ to vertex } v'_j, \ i \neq j \\
0 & \text{otherwise}
\end{cases} \]

and proceed iteratively to obtain the reachability matrix \( R \).

Let \( a_{i1}, a_{i2}, \ldots, a_{im} \) be the only \( m \) non-zero values of the first row of \( A \). We set

\[ a_{ij} = a_{i1} \land a_{i2} \land \ldots \land a_{im}, \quad j = 1, 2, \ldots, n \]

If the new first row of \( A \) differs from the old one, we perform the operation again, using the \( m' - m \) new non-zero entries in the first row. If the new first row of \( A \) is identical to the previous one, the first row of \( R \) has been obtained. We now proceed to obtain the second row, then the third. \ldots This process is often known as fusion.

When all \( n \) rows have been computed, the non-zero diagonal entries of \( R \) indicate those vertices that belong to some directed circuits. If we now compute

\[ R = R \land R^T \]

where the logical product is taken element by element, the resulting \( R \) matrix has the following structure:

- if \( r_{ii} = 1 \) vertex \( i \) is on a directed circuit
- if \( r_{ij} = 1 \) then \( r_{ji} = 1 \) by construction, \( v'_i \) and \( v'_j \) are on the same maximal circuit.

### A4 Selecting entry points

The method for finding descendants that we have just described can now be used to solve a problem deferred earlier, namely, the appropriate selection of vertices to make the graph weakly connected. Such a selection is necessary when the initial graph still has more than one component after all vertices with indegree 0 have been linked to the artificially created begin node \( v_0 \). Let us assume that there are \( p \) such remaining unreachable vertices \( V = \{v_1, v_2, \ldots, v_p\} \). The problem is to find the minimal number \( k \) of vertices from \( V \) sufficient to cover all vertices in \( V \); i.e., all vertices in \( V \) are reachable from 1 or more of the \( k \) selected “entry” vertices. Notice that the solution is, in general, not unique since any vertex on a circuit may be replaced by any other on the same circuit.

The following algorithm finds the entry vertices:

```
LET E( ), R( ), C( ) BE P ELEMENT VECTORS WITH VALUE 0, 1
INITIALLY SET E( )=0, C( )=0
DO WHILE THERE EXISTS I SUCH THAT C(I)=0
  FIND THE DESCENDANTS (R( )) OF VERTEX (I)
  I.E. R(J)=1 IF VERTEX J IS REACHABLE FROM I
  SET E(I)=(E( ) \land R( ))
  SET E(I)=1
  SET C( )=C( ) \lor E( ) \lor R( )
ENDDO WHILE

. . . THE ENTRY VERTICES ARE I SUCH THAT E(I)=1, I=1, \ldots, p
```

The selection of entry vertices is illustrated on Figure 5.

The arrows ‘\( \rightarrow \)’ in the left margin indicate addresses of the IBM OS/VS FORTRAN run time library which are not directly referenced within the load module.