SETS AND GRAPHS IN GRAAL*

Victor R. Basili
Department of
Computer Science
University of Maryland

KEY WORDS: GRAAL, graphs, sets, model, design

This paper is an attempt at presenting a high level model of the set and graph aspects of the graph algorithmic language GRAAL [5]. The problem area for which the language GRAAL was designed was the solution of graph problems of the type primarily arising in applications. It was designed with two objectives in mind. The first was to develop a language which permitted the writing of graph algorithms in a highly readable form with as natural a set of primitives as possible for describing the algorithm. The second was to allow for a wide variety of graphs of different types and complexity with as little degradation as possible in the efficient implementation and execution of an algorithm designed for a specific type of problem.

LANGUAGE OVERVIEW

The first pass in the design consisted of a general specification of the language primitives. It was decided that strictly set theoretic development of graph theory would allow for considerable flexibility in the selection of storage representations for different graph structures (see [5] for a motivation of this decision). Therefore two primitives that were included in the language were sets and graphs.

Sets, however, were placed in GRAAL mainly for the purpose of defining graphs and their specific design was motivated by this. Their introduction into the language generated the need for several set operations. These include the standard set union ($\cup$), intersection ($\wedge$), difference ($-$), and symmetric sum (Δ). There is a subset operator which constructs the subset of all elements of a set that satisfy a specified boolean relation. There are some common relational operators such as $=$, $\neq$, and $\subseteq$ which return the value true if two sets are equal, not equal or one is a subset of the other, respectively, and the value false otherwise.

There are a variety of graph structures available in GRAAL distinguished by the family of graph operators provided for constructing and traversing each specific structure. The present graph definitions include a directed pseudograph, an undirected pseudograph, a directed graph, in node form and an undirected graph in node form. We use the generic term "graph" for all of these.

*This research was supported in part by the Office of Naval Research, Mathematics Branch, under Grant N00014-67-A-0239-0021 (NR-044-431).

The graph construction operators consist of an assign operator which would attach a node, arc, node and arc or two nodes and their connecting arc to a graph, and a detach operator which would remove a set of nodes and/or arcs from a graph. Both of these operators return graphs as a result.

There are two graph operators, nodes and arcs, which return the set of all nodes and the set of all arcs in a graph, respectively.

The graph connectivity operators take as an argument a set expression which designates either a set of nodes or of arcs of the specified graph. The various possible operators presently included in the language are the incidence operator inc, the positive and negative incidence operators pinc and ninc, the star operator star, the positive and negative star operators pstar and nstar, the bound operator bd, the positive and negative boundary operators pbd and nbd, the coboundary operator cob, the positive and negative coboundary operators pcb and ncb, the adjacency operator adj, and the positive and negative adjacency operators padj and adj. Each one of these operators is associated with specific types of graphs.

The data structures decided upon for the language were arrays and lists. It was felt that these two structures could handle the static and dynamic storage needs of the type of problems attacked by the language.

The control structure decided upon was the standard algebraic language control structure since the purpose of the language was for writing readable and easily expressed algorithms. In fact, it was felt that the language should be imbedded in a standard algebraic language format which includes the normal integer and real variables and integer and real arithmetic.

This defines the basic outline of language features. Almost all, except for the set and graph constructs, are common to a large number of standard programming languages, such as ALGOL or FORTRAN. In fact, GRAAL was originally designed as an extension to ALGOL [5] and a first implementation of the language was done as an extension to an existing FORTRAN compiler on the UNIVAC 1108 [1]. What is needed is a closer look at the definition of sets and graphs.

The first questions are what is a set in the context defined here, and what does it consist of? That is, where do sets fit in the definition of an algebraic programming language and what are reasonable ways of defining them for their use in the definition of graphs?

There are several other programming languages that include sets among their constructs. SETL [6] and SETTEL [9] are both languages which were designed for writing general set-theoretic algorithms. In SETL a set is basically considered as an unstructured data structure whose elements may be a variety of objects such as integers or even other sets. In SETTEL a set is defined over some user-specified universe which might be the integers or integers and reals, etc. MADCAP [7], a language which was primarily defined for combinatorial computing, treats sets as subsets of the natural numbers. There are also several other languages that have set constructs of some form or other.

For GRAAL it was decided at the beginning that a set would be a data type rather than a structure. The basic philosophy that guided this decision was that the data types of a language represented the basic set of primitives of the language. That is, they represent those data elements and their respective operators which are used at the basic level of the algorithms written in that language. Thus it was agreed that sets and the set operations were part of the basic primitive notions of GRAAL.

This is not a new approach, although it appears to be with respect to sets. It attempts to separate the concepts of data types and data structures into disjoint categories in a particular language, depending upon the applications addressed by that language. Data types are the primitive data elements of the application area and represent the lowest level upon which the algorithm is based. For example, in the standard algebraic languages, FORTRAN and ALGOL, integer and real are the basic data types since they are the basic units of data used by algorithms in the languages. These data types may be thought of as unstructured data types because their structure (which is usually defined by the machine word-size) is fixed. There are basic data types which do have a more variable, nonmachine-defined structure. Strings in SNOBOL, for example, are a structured data type. In fact, there is a very close analogy between strings in SNOBOL and sets in GRAAL. They both have a definite structure whose internal definition is hidden from the user and immaterial to him. They consist of subelements, which are characters for strings and elements for sets. They have operators acting upon them which allow the user to access any subgroup of the elements as well as operations between them, such as concatenation for strings or union for sets. They may be stored in structures, such as arrays, whose structural design is more visible to the user because he uses the array strictly as a storage mechanism for his primitive data types.

Granted that a set is a data type rather than a data structure, what is a reasonable definition of its members? They are elements, usually representing nodes or arcs, which are members of some universe of elements, just as the subelements of a string are characters which are members of some universe of characters. The difference is that the maximum size of the universe of characters is usually thought of as fixed while the universe of elements must be highly dynamic. In order to define a new element, it must be dynamically created from this universe of elements.

Each element may have associated with it any number of typed properties. This permits the elements of a set or the nodes and arcs of a graph to have any number of integer, real, or string values associated with them.

We may now ask a similar question about a graph. What is a graph and what is it composed of? The answer to this question is similar to the answer to the previous one. For the same reasons a graph is really a structured data type, which is composed of undeclared data elements which play the roles of nodes or arcs and are interrelated to define the particular graph structure.

MODEL OF SETS AND GRAPHS

Now that some of the basic informal GRAAL design has been given, we are ready to present a high-level informal model of these concepts. The operational semantic modeling language used is an informal version of the hierarchical language HGL [2].

Essentially HGL is a definitional facility for specifying models which may be used to describe the semantics of programming languages. The basic HGL primitives consist of (1) a set of nodes each with an associated structured value, atomic value, and a set of attributes, and (2) a set of primitive transition functions that can change the structure, atom, or an attribute associated with a node and can define and delete nodes. More specifically, each node n in HGL may be thought of as a high-level memory cell representing a unit of information about the language. The atomic value associated with the node v(n) usually represents some nonstructured program or data element. The structured value associated with a node h(n) usually represents some language component which must be structured and whose structure is of interest. The attributes associated with a node, a(n,attribute), usually represent characteristics or properties of the unit of information contained in a node. These attributes can be thought of as a symbol table which might contain the compile time properties known about that unit of information.

The association between a node and a structure value, atomic value, and attributes can be defined as mappings, but they may be thought of as pointers. Thus each node would have a pointer to an atomic value, a structure value, and a table of attributes. Let $\mathcal{N}$ be a set of nodes, $\mathcal{D}$ be a set of atoms, and A be a set of attributes. Let G be a set of graphs defined over elements of $\mathcal{N}$. An <u>attributed hierarchical graph (h-graph)</u> over $(\mathcal{N}, \mathcal{D}, A, G)$ is defined as a 4-tuple (N, v, h, a) where $N \subset \mathcal{N}$, $v: N \to \mathcal{D}$, $h: N \to G$, and $a: N \times I_k^+ \to A$ where I_k^+ denotes the first k-positive integers, are the value, structure, and attribute mappings, respectively.

The three primitives that change the atom, structure or an attribute associated with a node are setv(n,d) which assigns the atomic value d to node n , seth(n,g) which assigns the graph structure g to node n , seta(n,att,p) which assigns value p to attribute att of node n , respectively. The two primitves that add and remove nodes are define $(n|v(n)=d,h(n)=g,a(n,att_1)=a_1,\ldots,a(n,att_r=a_r))$ which adds a new node to the nodeset of the h-graph along with its associated v, h, and a mappings and delete(n) which removes the node n from the nodeset of the h-graph.

We now model the set and graph features of GRAAL by defining their interpretation on a high-level machine whose primitive operations are the HGL operations and whose data structure and control structure are given below.

The data structure used is a set. The set is defined over nodes from N. Construction primitives for a set will consist of (i) enumerating the elements of the set, e.g., $\{n_1,n_2\}$, (ii) defining sets in terms of other sets using the standard set-theoretic operators such as $\cup$, $\cap$, $-$, etc., (iii) defining a set by a predicate, i.e., $\{x|BE(x)=\text{true}\}$ where BE is some boolean expression on the set.

An accessing primitive which yields an element of a set will be defined by select (s) which returns a random element from the set s.

The control mechanism used for expressing algorithms consists of (i) set expressions using the set construction and accessing primitives along with the five HGL primitive functions, (ii) a linear sequence of statements of type (i) which are performed in sequential order. Each statement or subsequence of statements may be preceded by a conditional implying the statement or subsequence is performed only if the conditional is true.

We use these data and control structures to define a model (high-level machine architecture) for the set and graph features of GRAAL. The memory is essentially an associative memory which can be accessed by set operations returning the node or nodes which satisfy some specified requirements. It will also be assumed that each node in memory has a special name associated with it, called the id attribute of that node. This id-attribute will be used to access a node when the node id is known and uniquely specified; otherwise, some set operation will be used. For example, h(n), where n is a node, may sometimes be written as h(id(n)).

Let the universe of elements be defined by a node, with id-attribute U, which contains all the elements of the universe. Each element can then be represented as a node of U having an id-attribute which distinguishes it from all other element nodes. Assume the id of each element node takes the form u_i where i represents a unique integer ordering of the elements. In what follows an extended BNF notation is used to define the syntactic structures of the model. Its use is explained as the examples proceed. Basically, its uses brackets [x] to represent a node whose v or h mapping is defined by x. The distinction between the v and h mappings is made by using an underscore to designate the v mappings. We will use the term contents of a node when it is immaterial as to whether we are referring to the v or h mappings of that node. Thus

```
<universe> ::= [{<element>}]; id = U
```

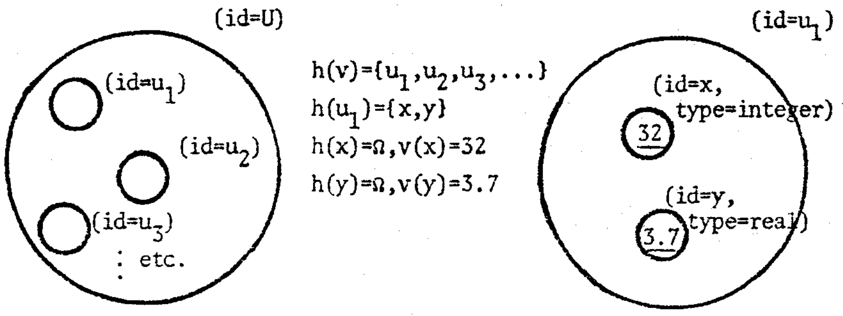
defines universe to be a node with one attribute id, equal to U, whose h mapping is a set of elements, i.e., h(U) = a set of elements.

Properties may then be associated with an element by setting the h mapping of that element node to a set of nodes called property nodes. The contents of a property node is the property value and the attributes represent the name and type of that property. The property may be an atomic value, if the type is integer or real, for example, or it may be a structure if the type is set or graph. Thus we may define

```
<element> ::= [{<property>}];
    id = <unique name u_i>
<property> ::= [<atomorstructure>];
    id = <identifier>, type = <type>
<atomorstructure> ::= <value>|<structure>
<structure> ::= {[<atomorstructure>]}
<value> ::= <identifier>|<number>| etc.
```

A set may then be modeled by a node whose content is a set of element nodes $\{u_i,\ldots,u_k\}$. It should be noted that the nodes in the set are not copies of the element nodes but the actual element nodes. (In terms of an implementation, a set can be though of as containing pointers to the actual elements in the universe.)

```
<set> ::= [{<element>}];
    id = <identifier>, type = set
```



The universe of elements and an element with two properties x and y

Figure 1

For convenience let us assume there is a node with id-attribute SETS, such that h(SETS) = the nodes defining all the sets in the language, i.e.,

<sets> ::= [{<set>}]; id = SETS

The GRAAL set operators can now be defined using the above definitions of the set and universe data structures and the set construction and accessing operators. The GRAAL operators are underlined to distinguish them from the set-theoretic ones.

The GRAAL set operators are create which creates a single element set, called an atomic set, from a newly-specified element in the universe; subset which creates a set of elements from elements in the universe that satisfy some boolean condition; elt which creates an atomic set by randomly selecting an element from a designated set; $\underline{\cup}$ the set union operator; $\underline{\cap}$ the set intersection operator; $\underline{\sim}$ the set difference operator; $\underline{\Delta}$ the symmetric sum operator; $\underline{=}$ the equivalence predicate; $\underline{\neq}$ the nonequivalence predicate; $\underline{\subseteq}$ the subset predicate. These operators may be defined in terms of the model as follows:

$\underline{\text{create}} = \{\text{select}(\{n \in h(U) \mid v(n)=h(n)=a(n)=\Omega \wedge n/h(S),\ \forall\, S \in \text{SETS}\})\}$ where Ω represents an undefined value

$\underline{\text{subset}}(x,\text{BE}(x)) = \{n \mid \text{BE}(n)=\underline{\text{true}}\}$, where BE is any valid boolean expression in the language

$$\underline{\text{elt}}(S) = \begin{cases} \{\text{select}(S)\} & \text{if } S = \emptyset \\ \emptyset & \text{if } S = \emptyset \end{cases}$$

$$\underline{\cup}(S,T) = h(S) \cup h(T)$$

$$\underline{\cap}(S,T) = h(S) \cap h(T)$$

$$\underline{\sim}(S,T) = h(S) \sim h(T)$$

$$\underline{\Delta}(S,T) = (h(S)-h(T)) \cup (h(T)-h(S))$$

$$\underline{=}(S,T) = \begin{cases} \underline{\text{true}} \text{ if } h(S) = h(T) \\ \underline{\text{false}} \text{ otherwise} \end{cases}$$

$$\underline{\neq}(S,T) = \begin{cases} \underline{\text{true}} \text{ if } h(S) \neq h(T) \\ \underline{\text{false}} \text{ otherwise} \end{cases}$$

$$\underline{\subseteq}(S,T) = \begin{cases} \underline{\text{true}} \text{ if } h(S) \subseteq h(T) \\ \underline{\text{false}} \text{ otherwise} \end{cases}$$

Accessing a property of an element node of a set is done by the two-argument function access whose first argument is the id of a property node and whose second argument is an atomic set containing the element whose property is desired. The function returns the property node itself.

$$\underline{\text{access}}(\text{prop},\text{set}) = \begin{cases} \text{select}(\{x \mid x \in h(\text{select}(S)) \wedge \text{id}(x)=\text{prop}\}) \\ \quad \text{if } |\text{set}|=1 \wedge \text{select}(\text{set}) \in h(U) \\ \underline{\text{undefined}} \text{ otherwise} \end{cases}$$

We will now discuss four of the various graph types found in GRAAL. For a more extensive discussion, see [3].

```
<graph> ::= <directed graph in node form>|
              <undirected graph in node form>
            <directed pseudograph>|
              <undirected pseudograph>
```

It was stated earlier that in GRAAL a graph is defined by its operators. A graph may then be modeled by the set of nodes which compose it if it is a graph in node form or the sets of nodes and arcs which compose it if it is a pseudograph. The actual graph structure of a graph g may be modeled by letting each of its component elements contain as property nodes the sets of nodes or arcs defined by the relevant graph operators on that component, relative to the graph g. For example, a directed graph in node form would be modeled by a node whose type attribute is directed graph and whose id attribute is the name of the graph, say g. This node would contain a single node representing the nodes of g and whose id would be nodes·g (a convenient way of chaining unique names). Then each of the elements in the node called nodes·g would contain two property nodes of type set, one containing the set of nodes which form the positive adjacency of that node in g, whose id attribute is pa·g, and one containing the set of nodes which form the negative adjacency of that node in g, whose id attribute in na·g. Other graph types are defined analogously. This approach permits the definition of graphs without imposing any specific data structure on them.

In order to make what follows more concise and at the same time to separate the data structure dependent primitives from the general algorithm, consider the following auxiliary functions: insert, add, and sub. Assume p is a property name, a is an atomic set, s is a set, and n is a node.

```
insert(p,a) = seth(select(a),
      h(select(a)∪{define(m|id(m)=p∧type(m)=set)})

/* insert adds to the element in a property node
   with id = p and type = set */

add(n,s) = seth(n,h(n)∪s)

/* add unions the set of nodes S to the contents
   of n */

sub(n,s) = seth(n,h(n)-s)

/* sub removes the set of nodes S from the con-
   tents of n */
```

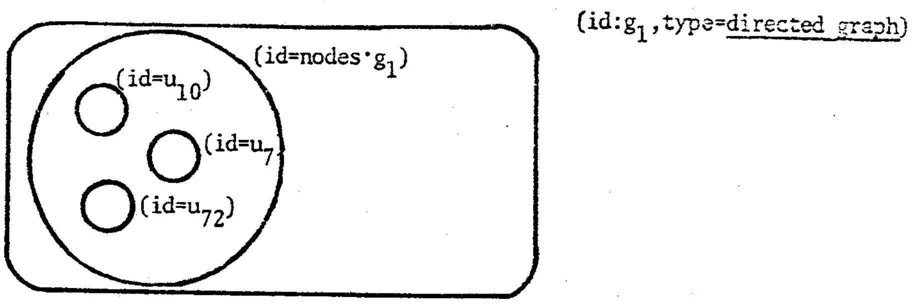
The directed graph, undirected graph, undirected pseudograph, and directed pseudograph and their relevant operations will now be defined. In what follows g will represent a graph of the appropriate type, a_i atomic sets, and s_i any sets.

directed graph

```
<directed graph in node form> ::= [<nodes>];
   id = <identifier>, type = directed graph

<nodes> ::= [{<element>}];
   id = nodes·<identifier>
```

For each directed graph g, the graph structure



$h(g_1) = \{\text{nodes}\cdot g_1\}$

$h(\text{nodes}\cdot g_1) = \{u_7, u_{10}, u_{72}\}$

$h(u_7) = \{\text{prop1}, \ldots, \text{propn}, \text{pa}\cdot g_1, \text{na}\cdot g_1, \ldots\}$
$h(\underline{\text{access}}(\text{pa}\cdot g_1, \{u_7\})) = \{u_{10}\}$
$h(\underline{\text{access}}(\text{na}\cdot g_1, \{u_7\})) = \{u_{72}\}$

$h(u_{72}) = \{\ldots \text{prop}_j, \ldots, a\cdot g_1, \text{na}\cdot g_1, \ldots\}$
$h(\underline{\text{access}}(\text{pa}\cdot g_1, \{u_{72}\})) = \{u_{10}, u_7\}$
$h(\underline{\text{access}}(\text{na}\cdot g_1, \{u_{72}\})) = \emptyset$

$h(u_{10}) = \{\ldots \text{prop}_k, \ldots, \text{pa}\cdot g_1, \text{na}\cdot g_1, \ldots\}$
$h(\underline{\text{access}}(\text{pa}\cdot g_1, \{u_{10}\})) = \emptyset$
$h(\underline{\text{access}}(\text{na}\cdot g_1, \{u_{10}\})) = \{u_7, u_{72}\}$

Representation of the Directed Graph g_1:

$$\begin{tikzcd}
 & u_{10} & \\
u_{72} \arrow[ur] \arrow[rr] & & u_7 \arrow[ul]
\end{tikzcd}$$

Figure 2

will be defined as follows: For all elements x $x \in h(\text{nodes}\cdot g)$, there will be two property nodes n_1 and $n_2 \in h(x)$ such that $id(n_1) = \text{pa}\cdot g$, $type(n_1) = \underline{\text{set}}$, $id(n_2) = \text{na}\cdot g$, and $type(n_2) = \text{set}$. These two nodes represent the sets of positive and negative adjacency nodes for that element x in g. These nodes are defined by the appropriate graph construction operators.

$\underline{\text{nodes}}(g) = \{n \mid n \in h(\text{nodes } g)\}$

```
assign(g,a_1,a_2) =
   $a_1 \not\subseteq$ nodes(g) → add(nodes·g,$a_1$)
                          insert(pa·g,$a_1$)
                          insert(na·g,$a_1$)
   $a_2 \not\subseteq$ nodes(g) → add(nodes·g,$a_2$)
                          insert(pa·g,$a_2$)
                          insert(na·g,$a_2$)
   add(access(pa·g,$n_1$),$n_2$)
   add(access(na·g,$n_2$),$n_1$)
```

```
assign(g,a_1) =
   $a_1 \not\subseteq$ nodes(g) → add(nodes·g,$a_1$)
                          insert(pa·g,$a_1$)
                          insert(na·g,$a_1$)
```

```
detach(g,s) =
   sub(nodes·g,s)
   sub(access(pa·g,{x}),s), $\forall x \in$ nodes(g)
   sub(access(na·g,{x}),s), $\forall x \in$ nodes(g)
   sub(y,{pa·g}), $\forall y \in s$
   sub(y,{na·g}), $\forall y \in s$
```

```
detach(g,s_1,s_2) =
   sub(access(pa·g,{x}),$s_1$), $\forall x \in s_2$
   sub(access(na·g,{x}),$s_2$), $\forall x \in s_1$
```

$\underline{\text{padj}}(s,g) = \bigcup_{x \in s} h(\underline{\text{access}}(\text{pa}\cdot g, \{x\}))$
$\underline{\text{nadj}}(s,g) = \bigcup_{x \in s} h(\underline{\text{access}}(\text{na}\cdot g, \{x\}))$
$\underline{\text{adj}}(s,g) = \underline{\text{padj}}(s,g) \cup \underline{\text{nadj}}(s,g)$

The undirected graph, the undirected pseudograph, and the directed pseudograph are defined in analogous ways.

undirected graph

<undirected graph in node form> ::= [<nodes>]; id = <identifier>, type = undirected graph

```
assign(g,a_1a_2) =
   $a_1 \not\subseteq$ nodes(g) → add(nodes·g,$a_1$)
```

```
                       insert(a·g,$a_1$)
  $a_2 \not\subseteq$ nodes(g) → add(nodes·g,$a_1$)
                       insert(a·g,$a_2$)
  add(access(a·g,$a_1$),$a_2$)
  add(access(a·g,$a_2$),$a_1$)
assign(g,$a_1$)
  $a_1 \not\subseteq$ nodes(g) → add(nodes·g,$a_1$)
                       insert(a·g,$a_1$)
detach(g,s) =
  sub(nodes·g,s)
  sub(access(a·g,{x}),s), $\forall$ x $\in$ nodes(g)
  sub(y,{a·g}), $\forall$ y $\in$ s
detach(g,$s_1$,$s_2$) =
  sub(access(a·g,{x}),$s_1$), $\forall$ x $\in$ $s_2$
  sub(access(a·g,{x}),$s_2$), $\forall$ x $\in$ $s_1$
adj(s,g) = $\bigcup_{x \in s}$ h(access(a·g,{x})
```

undirected pseudograph

```
  <undirected pseudograph> ::= [<nodes>,<arcs>];
    id = <identifier>, type = undirected pseudo-
    graph

  arcs ::= [{<element>}]; id = arcs·<identifier>
```

Then the operators are

```
      arcs(g) = {n|n$\in$h(arcs·g)}
assign(g,$a_1$,$a_2$,$a_3$) =
  $a_1 \not\subseteq$ nodes(g)    add(nodes·g,$a_1$)
                     insert(s·g,$a_1$)
  $a_2 \not\subseteq$ nodes(g)    add(nodes·g,$a_2$)
                     insert(s·g,$a_2$)
  $a_3 \not\subseteq$ arcs(g)     add(arcs·g,$a_3$)
                     insert(i·g,$a_3$)
  add(access(s·g,$a_1$),$a_3$)
  add(access(s·g,$a_2$),$a_3$)
  add(access(i·g,$a_3$),$a_1$)
  add(access(i·g,$a_3$),$a_2$)
detach(g,S) =
  let  $S_a$ = {m$\in$S|m$\in$arcs(g)}
  then sub(arcs·g,$S_a$)
       sub(access(s·g,{x}),$S_a$), $\forall$ x $\in$ nodes(g)
       sub(y,{i·g}), $\forall$ y $\in$ $S_a$
  let  $S_n$ = {m$\in$S|m$\in$nodes(g)}
  and $S_{na}$ = {m$\in$arcs(g)|inc({m},g)$\cap S_n \neq \emptyset$}
  then sub(nodes·g,$S_n$)
       sub(arcs·g,$S_{na}$)
       sub(y,{s·g}), $\forall$ y $\in$ $S_n$
       sub(y,{i·g}), $\forall$ y $\in$ $S_{na}$
       sub(access(s·g,{x}),$S_{na}$), $\forall$ x $\in$ nodes(g)
star(s,g) = $\bigcup_{x \in S}$ h(access(s·g,{x})
inc(s,g) = $\bigcup_{x \in S}$ h(access(i·g,{x})
bd(s,g) = $\Delta_{x \in S}$ {inc({x},g)|size(inc({x},g))=2}
cob(s,g) = $\Delta_{x \in S}$ {star({x},g)|size(inc({m},g))=2}
```

directed pseudograph

```
  <directed pseudograph> ::= [<nodes>,<arcs>];
    id = identifier type = directed pseudograph

assign(g,$a_1$,$a_2$,$a_3$) =
  $a_1 \not\subseteq$ nodes(g)  → add(nodes·g,$a_1$)
                      insert(ps·g,$a_1$)
                      insert(ns·g,$a_1$)
  $a_2 \not\subseteq$ nodes(g)  → add(nodes·g,$a_2$)
                      insert(ps·g,$a_2$)
                      insert(ns·g,$a_2$)
  $a_3 \not\subseteq$ arcs(g)   → add(arcs·g,$a_3$)
                      insert(pi·g,$a_3$)
                      insert(ni·g,$a_3$)
  add(access(ps·g,$a_1$),$a_3$)
  add(access(ns·g,$a_2$),$a_3$)
  add(access(pi·g,$a_3$),$a_1$)
  add(access(ni·g,$a_3$),$a_2$)
detach(g,S) =
  let  $S_a$ = {m$\in$S|m$\in$arcs(g)}
  then sub(arcs·g,$S_a$)
       sub(access(ps·g,{x}),$S_a$), $\forall$ x $\in$ nodes(g)
       sub(access(ns·g,{x}),$S_a$), $\forall$ x $\in$ nodes(g)
       sub(y,{ni·g}), $\forall$ y $\in$ $S_a$
       sub(y,{pi·g}), $\forall$ y $\in$ $S_a$
  let $S_n$ = {m$\in$S|m$\in$nodes(g)}
  and $S_{na}$ = {m$\in$arcs(g)|pinc({x},g) $\cap S_n \neq \emptyset$
                   $\vee$ ninc({x},g) $\cap S_n = \emptyset$
  then sub(nodes·g,$S_n$)
       sub(arcs·g,$S_{na}$)
       sub(y,{ps·g}), $\forall$ y $\in$ $S_n$
       sub(y,{ns·g}), $\forall$ y $\in$ $S_n$
       sub(y,{pi·g}), $\forall$ y $\in$ $S_{na}$
       sub(y,{ni·g}), $\forall$ y $\in$ $S_{na}$
       sub(access(ps·g,{x},$S_{na}$),$\forall$x $\in$ nodes(g))
       sub(access(ns·g,{x},$S_{na}$),$\forall$x $\in$ nodes(g))
```

$$\underline{\mathrm{pstar}}(S,g) = \bigcup_{x\in S} h(\underline{\mathrm{access}}(ps\cdot g,\{x\}))$$
$$\underline{\mathrm{nstar}}(S,g) = \bigcup_{x\in S} h(\underline{\mathrm{access}}(ns\cdot g,\{x\}))$$
$$\underline{\mathrm{star}}(S,g) = \underline{\mathrm{pstar}}(S,g) \cup \underline{\mathrm{nstar}}(S,g)$$

$$\underline{\mathrm{pinc}}(S,g) = \bigcup_{x\in S} h(\underline{\mathrm{access}}(pi\cdot g,\{x\}))$$
$$\underline{\mathrm{ninc}}(S,g) = \bigcup_{x\in S} h(\underline{\mathrm{access}}(ni\cdot g,\{x\}))$$
$$\underline{\mathrm{inc}}(S,g) = \underline{\mathrm{pinc}}(S,g) \cup \underline{\mathrm{ninc}}(S,g)$$

$$\underline{\mathrm{pbd}}(S,g) = \Delta_{x\in S}\, \underline{\mathrm{pinc}}(\{x\},g)$$
$$\underline{\mathrm{nbd}}(S,g) = \Delta_{x\in S}\, \underline{\mathrm{ninc}}(\{x\},g)$$
$$\underline{\mathrm{bd}}(S,g) = \underline{\mathrm{pbd}}(S,g)\ \Delta\ \underline{\mathrm{nbd}}(S,g)$$

$$\underline{\mathrm{pcob}}(S,g) = \Delta_{x\in S}\, \underline{\mathrm{pstar}}(\{x\},g)$$
$$\underline{\mathrm{ncob}}(S,g) = \Delta_{x\in S}\, \underline{\mathrm{nstar}}(\{x\},g)$$
$$\underline{\mathrm{cob}}(S,g) = \underline{\mathrm{pcob}}(S,g)\ \Delta\ \underline{\mathrm{ncob}}(S,g)$$

EXAMPLES AND IMPLEMENTATION

In order to demonstrate the use of these graph and set features in the solution of a particular problem, consider these features imbedded in a standard algebraic programming language. Two procedures are given; one creates a subgraph of a graph, the other creates a spanning tree for a graph.

```
procedure subgraph (graph G, set N, graph subG)
  /* This procedure sets up the subgraph of    */
  /* G  which has a given set N of nodes of    */
  /* G  as node set.                           */
set S,x,y,a
while N ≠ empty
  do x := elt  (N)
    S := subset(a, a ∈ star(x,G) ∧ inc(a,G) ⊆ N)
    N := N ∿ x
    if  S  = empty
      then assign (subG,x)
      else for all   a ∈ S
        do y   := inc(a,G) ∿ x
          if  y  = empty
            then  y  :=  x
            end /* if */
          assign (subG, x - y  to  a)
        end /* for all */
      end /* if */
end /* while */
```

```
procedure spantree (graph G, set r, graph TREE)
  /* This procedure generates a directed span-  */
  /* ning tree with root  r  for the connected  */
  /* component of  G  containing the node  r .  */
set S,T,w,x,y,a
  S := r
  T := cob(r,G)
  while T ≠ empty
    do for all a ∈ T
      do w := bd(a,G)
        y := w - S
        if y ≠ empty
          then S := S ∨ y
               x := w ∿ y
               assign(TREE, x → y to a)
          end /* if */
      end /* for all */
    T := cob(S,G)
end /* while */
```

For the sake of completeness, a brief discussion of a possible internal data structuring of the universe, sets and graphs will now be given. A natural data structure for the universe of elements would be dynamic array-like structure which would permit an easy and quick ordering and accessing scheme for the elements. This can be done by using the index of the element in the array as its unique (integer) name. This would lead to a relatively efficient implementation of the universe and of sets.

Sets could be implemented using a list type data structure because of their dynamically varying size. These lists can be ordered, however, according to the unique integer name associated with each element in the universe. This intrinsic ordering permits a great improvement in the efficiency of the set operations [7].

Properties, like sets, should also be defined by a list structure since there are a variable number of properties associated with each element. This tends to be efficient with respect to space but inefficient with respect to accessing speed.

Graphs are essentially defined through the properties of their nodes and arcs. However the individual elements of a particular graph can be chained together across their respective property lists. For a complete description of a particular implementation, see [4].

REFERENCES

[1] Basili, V.R., Mesztenyi, C.K., and Rheinboldt, W.C., FGRAAL--FORTRAN extended graph algorithmic language, University of Maryland, Computer Science Center, Technical Report TR-179, 1972.

[2] Basili, V.R., and Turner, A.J., A hierarchical machine model for semantics of programming languages, Proceedings, Symposium on High-Level Language Computer Architecture, ACM, November 1973.

[3] Basili, V.R., A structured approach to the design of GRAAL, University of Maryland, Computer Science Center, Technical Report TR-299, 1974.

[4] Mesztenyi, C.K., Brietenlohner, H., and Yeh, J.C., FGRAAL technical documentation, University of Maryland, Computer Science Center, Technical Report TR-200, 1972.

[5] Rheinboldt, W.C., Basili, V.R., and Mesztenyi, C.K., On a programming language for graph algorithms, BIT 12, 1972, 220-241.

[6] Schwartz, J., On programming, I and II: An interim report on the SETL project, New York University, Courant Institute of Mathematical Science, 1973.

[7] Shapiro, S., The list set generator, CACM 13, 1970, 741-744.

[8] Wells, M.B., Elements of Combinatorial Computing, Pergamon Press, Oxford, 1970.

[9] Wipple, J., Set theoretic extensions of algorithmic languages, Ph.D. Dissertation, Stevens Institute of Technology, 1972.