A Model for Representing Programs Using Hierarchical Graphs

STEPHEN S. YAU, FELLOW, IEEE, AND PAUL C. GRABOW, STUDENT MEMBER, IEEE

Abstract—**In this paper a hierarchical graph model for programs based on the concepts of recursive graphs (RG's) and Codd relations is presented. The purpose of the model is to clearly represent the structure of a program implemented in a structured language, such as Pascal, Algol, or PL/1, so that the program can be analyzed and modifications to the program can be clearly specified. The model uses an RG representation for the control flow and the data flow with an equivalent relational representation. It also has a relational representation for the objects defined within the program. The various aspects of the model are illustrated using Pascal constructs and a model for an example Pascal program is given.**

Index Terms—**Codd relations, control flow, data flow, graph grammar, hierarchical graphs, program analysis, program model, program modifications, program objects, recursive graph.**

I. Introduction

A DIRECTED graph has been used to describe the control flow in a program [1]-[4]. It has also been used to describe the data flow in conjunction with the control flow [5]-[6]. The control-flow graph (or program graph) represents the ways in which control can be transferred between program units during execution. A data-flow graph shows how data objects can move between program units during execution. Hierarchical graphs, such as H-graphs [7]-[9], recursive graphs [10]-[12], and the Graph Model of Behavior used in SARA [13], [14], have been used in conjunction with design and/or analysis of programs which are hierarchically structured. Graph models have been used in data-flow analysis of programs [15]-[25]. Various specification and/or design methodologies [26]-[31] have used program models with graphical representations.

In this paper, we will present a hierarchical graph model for programs based on the concepts of *recursive graphs* (RG's) and *Codd relations* [32], [33]. The purpose of this model is to clearly represent the structure of a program implemented in a structured language, such as Pascal, Algol, or PL/1, so that the program can be analyzed and modifications to the program can be clearly specified. The structure of a program includes the control flow, the data flow, and the objects defined within the program. In Pascal, these objects include constants, labels, types, variables, procedures, and functions. The model is a hierarchy of labeled, directed graphs which represent the control flow and the data flow of the program being modeled. These graphs are represented as a collection of Codd relations. Additional relations are used to describe the structure of program objects. The operations on the model are defined in terms of the graph structures. Applications of the model include the low-level specification of modifications to a program and to facilitate the analysis of the program, especially during the maintenance phase of the software life cycle.

In order to clearly represent the structure of a program so that analyses can be performed and modifications specified, the model should be defined in terms of the grammar of the implementation language of the program being modeled and formal operations should be used to specify modifications to the program. The model should be hierarchical so that it can easily represent programs written in structured languages. It must be able to represent the control flow, the data flow, and the structure of program objects in a uniform way. Furthermore, it should be capable of interfacing with a text editor and should allow graphic representation. The model should be implementable using existing software tools and database systems.

The RG representation for the model was chosen because it is able to relate complex information about the program structure in a clear way. The hierarchical nature of an RG allows us to analyze any particular level of a hierarchical structure such as the statement structure of a program written in a structured language. The RG can be stored and managed by a database system where the relational model describes the logical structure of the database. The physical structure of the database, however, can be something other than relational. The relational model allows us to implement operations on the model in terms of the relational algebra or the relational calculus. Furthermore, relations can be used to represent the control flow, the data flow, and the structure of program objects in a uniform way.

To illustrate these ideas, we will describe a portion of the model for an example Pascal [34] program. We will also describe some basic operations on the model in terms of the RG representation.

In order to simplify the discussion and the representation of the model in this paper, we make the following assumptions: the body of a REPEAT is a COMPOUND statement; function calls have no side effects; <string>, <expression>, <unsigned integer>, and <unsigned real> are treated as terminal symbols with values; and we treat READ and WRITE procedure calls as statement types with definite input or definite output.

Manuscript received January 26, 1981; revised June 29, 1981. This work was supported by the Office of Naval Research under Contract N00014-80-C-0167.

The authors are with the Department of Electrical Engineering and Computer Science, Northwestern University, Evanston, IL 60201.

II. Preliminaries

In this section, we will briefly describe the concept of an RG, a graph-grammar [9] representation of control statements, and the way in which we construct relations from the grammar for the language used to implement the program. An RG is used as the graphical representation of the model. The graph grammar for a structured language provides the link between the formal definition of the language and the RG model of a program written in that structured language. The grammar is used to identify the relationships which can exist between syntactic constructs in the language. This information is then used to define a database representation for these constructs.

A Recursive Graph (RG)

The concept of an RG was first introduced by Kunii, Harada, and Saito [10]. The specific notation to be used in our application here can be derived from the definition in Buchmann and Kunii [12], in which an RG is considered as a pair

$$R = (N,A)$$

where N and A are two sets of parameterized nodes and arcs, respectively.

A node n_i in N is defined as

$$n_i(\text{nid},\text{ns},\text{type},\text{nptr})$$

where "nid" is a unique numeric identifier for a node, "ns" is a semantic descriptor for the node, "type" gives the type of node, and "nptr" is a pointer to the graph at an adjacent level of detail.

Similarly, an arc a_i in A is defined as

$$a_i(\text{aid},\text{as},\text{bn},\text{en},\text{aptr})$$

where "aid" and "as" are the arc identifier and the semantic descriptor, respectively, "bn" and "en" represent the beginning node and end node of an arc, respectively, and "aptr" is a pointer to the arc at an adjacent level of detail.

A Graph Grammar and a Graph Language

A *graph language* [5], [9] is a language defined by a set of production rules which are expressed as directed graphs. This set of production rules constitutes the *graph grammar* [5], [9]. A graph grammar is a generalization of an ordinary context-free grammar which defines a graph language. A graph grammar generates a language of *terminal graphs* in much the same way as a context-free grammar generates a set of terminal strings.

The Construction of Relations from a Grammar

In this section we will describe how relations are constructed to represent program objects. Program objects are syntactic constructs defined in the grammar of the language used to implement the program being modeled. A procedure for constructing a set of relations from the Backus Naur form (BNF) of a grammar is presented. How the procedure should be used and some of the associated problems will be discussed. To illustrate the various aspects of the method we refer to the production rules and the associated relations of the Appendix.

The grammar identifies the syntactic constructs and the relationships which can legally exist among them. A production rule in the grammar is used as input to a procedure which generates a set of relations for the given production rule. The BNF of a grammar is used here since it is a formal meta-language which allows us to precisely describe the process used to define the relations. This is necessary if the process is to be in any way automated. Other forms of the grammar can be used, however, if they offer capabilities equivalent to BNF. Our examples here use the BNF grammar for Pascal from [35].

For convenience, we identify two types of production rules: *class* and *structure*. A *class rule* contains at least two nonempty "choices" on the right-hand side. Each choice can be considered as a class. All other rules are considered *structure rules.*

The following procedure can be used to construct relations from a BNF production rule. In order to illustrate some specific cases, we reference certain relations in the Appendix which demonstrate the procedure for the Pascal language.

1) If there is no relation with the name of the production rule, then assign the name of the production rule to the relation (e.g., IDENTIFIER) and include an attribute composed of the relation name concatenated with "-ID" (e.g., IDENTIFIER-ID) as the first attribute in the primary key, else stop.

2) If the production rule is a structure rule, then do the following.

a) For each nonterminal on the right-hand side which occurs more than once in a sequence (e.g., <identifier>, {<identifier>}), form a relation name using the nonterminal in the list concatenated with "-LST" (e.g., IDENTIFIER-LST); if there is no relation with this name, then form a new relation with this name and do the following.

i) Use the relation name concatenated with "-LST-ID" to form the name of the first attribute in the primary key (see relation (3) in the Appendix).

ii) Add the attribute "SEQ" to the relation as the second attribute of the primary key. This determines the order in which the elements occur in the sequence.

iii) Add an attribute to the list relation using the nonterminal name concatenated with "-ID" for the attribute name.

b) For each nonterminal on the right-hand side of the rule, add an attribute to the relation with the name of the attribute constructed from the name of the nonterminal concatenated with "-ID" (e.g., TYPE-ID). (For example, see relation (23) in the Appendix.)

3) If the rule is a class rule, do the following.

a) If each class is a single nonterminal which occurs only once in the production rule, then add the attributes CLASS and CLASS-ID to the relation; the domain for CLASS will be the set of nonterminals from the right-hand side of the production rule.

b) If a nonterminal occurs in more than one class on the right-hand side of the production rule, give each class a unique name, create a production rule for each class with its unique name, and form a relation for each new production rule.

Using this procedure, a class rule can be represented as a relation of the form

<u>name | **name-id** class class-id</u>

where the name of the relation corresponds to the name of the rule, the attribute "class" has the set of "choices" from the right-hand side of the rule as its domain, and attribute "class-id" identifies a tuple in the relation which corresponds to the relation of a given class name. When a class rule does not have a simple name for a choice, then the choice is given any unique name. For example, the production rule

<assignment statement> ::= <variable> := <expression>|
<function identifier> := <expression>

may have the following relational representation:

assign | **assign-id** class class-id

where the attribute "class" has the domain {A-VAR, A-FUNC}. Using the above approach, we create two more relations, "A-VAR" and "A-FUNC," to represent the two classes in the domain for attribute "class."

The procedure creates more relations than necessary and/or practical. However, our approach in defining the model is to use the procedure to generate a set of relations, and then to combine relations wherever possible. When relations are combined, some information may be lost. However, the information lost may not be critical considering the way in which the model is used. For example, it may not be necessary to make the distinction between <type> and <simple type> in the Pascal grammar. Therefore, the two production rules may be combined under <type>. Once a set of relations are defined, the compiler for the language in which the program is written can be modified to produce the relations during compilation.

The Appendix gives the relations and their associated production rules for the Pascal language which were reduced from the set of relations created using the procedure. Only experimentation with the model can determine if this set is sufficient for our purposes. Just as it is difficult to determine if a set of production rules represents a minimally complete set for a programming language, it is difficult to decide if a set of relations used to represent a program is minimally complete for the model. One of the reasons for choosing a relational representation for the model is to facilitate experiments with various sets of relations. The relational database model allows us to redefine relations easily by adding attributes, combining relations, or creating more than one relation from a given one.

Many of the relations in the Appendix were formed from more than one production rule. Some were defined by using our procedure after the production rules were combined. Others were formed from joins and the elimination of the attribute on which the join was performed. For example, relations (1)-(3) were formed by using the procedure on the production rule

<program> ::= **program** <identifier> (<identifier>
{, <identifier>});

This rule is a combination of the three rules which precede relation (1). Note that the information classifying the identifiers in the list as file identifiers was lost. However, we have included an extra attribute in the relation for <identifier> to distinguish the class of an identifier.

Relation (11) can be formed by joining the relations that would be created by applying the procedure to the two production rules individually. The following three relations are created by applying the procedure to each rule separately:

type-def-part | **type-def-part-id** type-def-1st-id

type-def-1st | **type-def-1st-id seq** type-def-id

type-def | **type-def-id** ident-id type-id

If the last two relations are joined over "type-def-id," and this attribute is then deleted, we have

temp1 | **type-def-1st-id seq** ident-id type-id

If this relation is joined with "type-def-part" over "type-def-1st-id" and this attribute is then deleted, we obtain

temp2 | **type-def-part-id seq** ident-id type-id

which is equivalent to relation (11) in the Appendix.

In the Appendix, <unsigned integer>, <unsigned real>, and <identifier> are not decomposed. For example, in relation INT (relation (8) in the Appendix), the values for attribute LITERAL are simply unsigned integers. In relation EXP (relation (38) in the Appendix), expressions are represented as strings. The representation chosen for expressions, integers, reals, and strings will depend on the use of the model.

III. The Model

Syntactic constructs defined by the BNF of the implementation language are used to define the model. Fig. 1 shows an RG representation for a module. It is noted that a module can be an entire program, a procedure, or a function. Node S0 represents the statement block, nodes Di and Do represent the in-input and output sets (formal parameters and global variables). The nodes labeled P and Q serve as the input and output ports, respectively. The rectangles on the left represent the objects defined with the module. Here, "MOD-PRT" is used to denote the function and procedure declaration part.

The Control-Flow Representation

For the control-flow representation, nodes are defined in terms of statement types in the implementation language. The statement block of a module written in a structured language is one COMPOUND statement which can be expanded into a sequence of statements. These statements can, in turn, be expanded until there are no more statements to expand. The statement types we consider are COMPOUND, IF-THEN-ELSE, WHILE, REPEAT, CASE, and FOR. We define an RG representation, called a *statement control-flow graph* (SCG), for each of these statement types based on the corresponding production rule in the graph grammar [9] of the implementation language. Fig. 2 shows the SCG's for six statement types.

Within the SCG's, we identify the following types of nodes: an *expression* (E) node, a *statement* (S) node, a *port* (P or Q) node, a *simple-block* (B) node, an *index* (I) node, and an *exit*

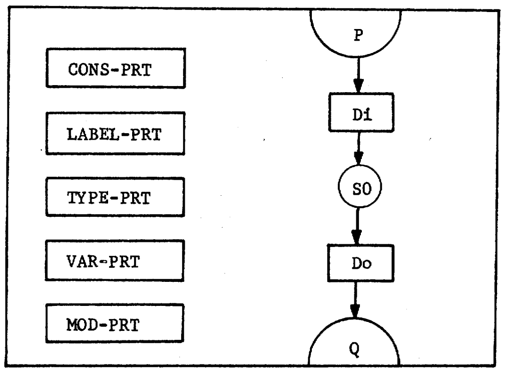

Fig. 1. RG representation for the uppermost level of a module.

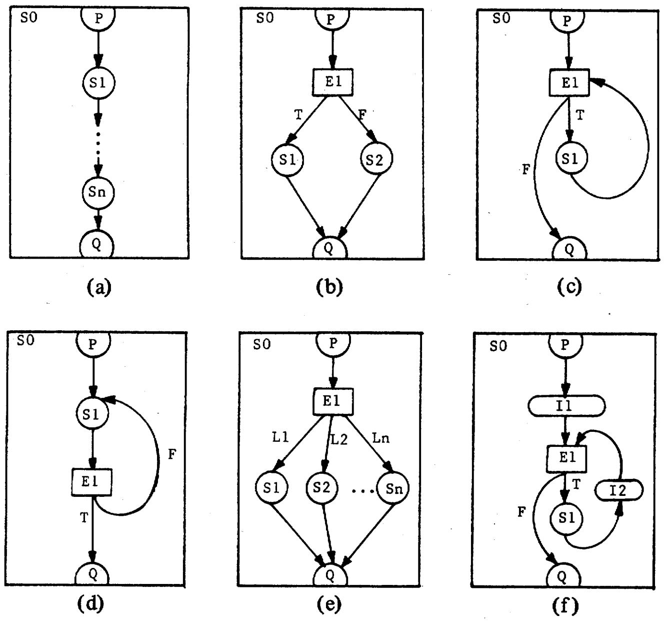

Fig. 2. RG representations for the control flow in the six statement types. (a) COMPOUND. (b) IF. (c) WHILE. (d) REPEAT. (e) CASE. (f) FOR.

(X) node. We consider E and B nodes to be *elementary* since they have definite input and output sets. An E node represents a decision point, and an S node stands for one of the statement types in the implementation language. A P node serves as the input port and a Q node as the output port for a graph. An I node represents the initialization or incrementation of an index variable in a FOR SCG. An X node is used when an unconditional jump goes to a node outside the statement graph in which the jump originates. A simple block, or B node, is a sequence of assignment statements which can end with an unconditional jump and only the first statement can be labeled. Therefore, if a simple block is executed, all of its included statements are executed in sequence. A B node is the most elementary node in a statement in which variables are set.

A label on an arc in an SCG gives the condition associated with the arc. For example, in Fig. 2(e), L1 represents the condition associated with the case element S1. A label can also be used to represent the ultimate destination of an exit jump.

We include two restrictions on the use of the unconditional jump: a) any unconditional jumps from statement nodes in statement graphs which are not COMPOUND graphs must be to

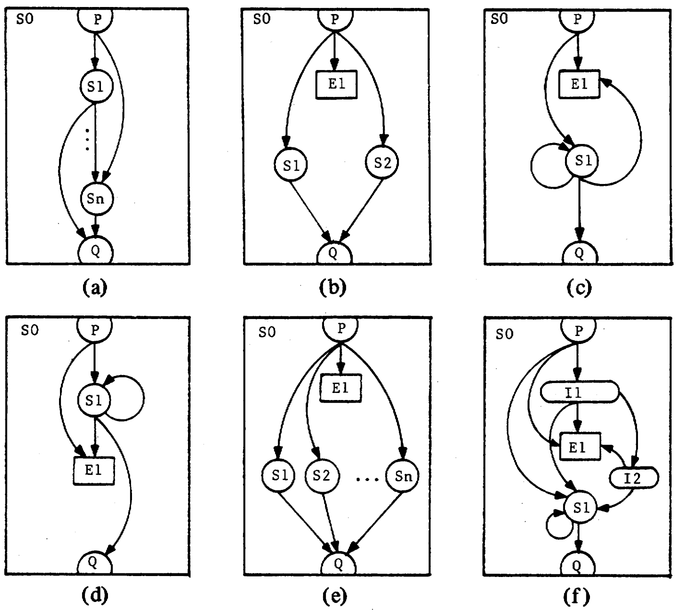

Fig. 3. RG representations for the data flow in the six statement types. (a) COMPOUND. (b) IF. (c) WHILE. (d) REPEAT. (e) CASE. (f) FOR.

an exit port in the graph and b) an unconditional jump must terminate at the input or exit port of a graph. Given these restrictions, it can be shown that an unconditional jump can only go to a node at a higher level in the hierarchy if the jump did not originate in a COMPOUND graph. If the jump originated in a COMPOUND graph, it could go to another node in the same COMPOUND graph. These two restrictions are not severe; however, they simplify the tracing of unconditional jumps through a program and the representation of the data flow in a program with unconditional jumps.

The Data-Flow Representation

Like the control-flow description, the data-flow representation is in terms of the RG and the equivalent collection of Codd relations [32], [33]. We define a *statement data-flow graph* (SDG) for each statement type. The same node types used for the control-flow description are used to describe the data flow in a module. However, the arcs in a statement data-flow graph represent paths along which data can flow, not necessarily corresponding to control-flow paths. Fig. 3 shows the SDG's for six statement types. For the representation of data-flow arcs we define three notions: *data arc*, *arc set*, and *node sets*.

A *data arc* is simply a path along which data can flow in an SDG. There are a finite number of such paths in any SDG. The tail of a data arc identifies a node in which data could possibly be defined and the head of the arc gives the node in which the data could be used. An *arc set* is the set of data objects associated with a data arc.

We define three types of node sets: *input set* (DDi), *definite-output set* (DDo), and *possible-output set* (DPo). These sets are constructed according to whether or not a variable is used (referenced) or defined (given a value) within a node. The set DDi includes those variables which must be defined prior to entering the node since their values may be used during the execution of the node. The set DDo consists of those variables which are always defined within the node, no matter what the execution sequence within the node is. The set DPo contains those variables which may or may not be defined within the node, depending upon the execution sequence within the node. Node sets are constructed for statements in the order in which they would ordinarily be processed by a compiler using depth-first traversal. Therefore, when the model is originally constructed, the node sets can be constructed during compilation. The following equations are used to construct the node sets for the corresponding SDG's in Fig. 3:

a) COMPOUND

$$\mathrm{DDi(S0)} = \bigcup_{1 \leqslant i \leqslant n} \{\mathrm{DDi}(S_i) - \bigcup_{1 \leqslant j < i} \mathrm{DDo}(S_j)\}$$

$$\mathrm{DDo(S0)} = \bigcup_{1 \leqslant i \leqslant n} \mathrm{DDo}(S_i)$$

$$\mathrm{DPo(S0)} = \bigcup_{1 \leqslant i \leqslant n} \mathrm{DPo}(S_i) - \mathrm{DDo(S0)}.$$

b) IF

$$\mathrm{DDi(S0)} = \mathrm{DDi(E1)} \cup \mathrm{DDi(S1)} \cup \mathrm{DDi(S2)}$$

$$\mathrm{DDo(S0)} = \mathrm{DDo(S1)} \cap \mathrm{DDo(S2)}$$

$$\begin{aligned}\mathrm{DPo(S0)} &= \mathrm{DPo(S1)} \cup \mathrm{DPo(S2)} \cup \{(\mathrm{DDo(S1)} \cup \mathrm{DDo(S2)}) - (\mathrm{DDo(S1)} \cap \mathrm{DDo(S2)})\} \\ &= \{\mathrm{DPo(S1)} \cup \mathrm{DPo(S2)} \cup \mathrm{DDo(S1)} \cup \mathrm{DDo(S2)}\} - \mathrm{DDo(S0)}.\end{aligned}$$

c) WHILE

$$\mathrm{DDi(S0)} = \mathrm{DDi(E1)} \cup \mathrm{DDi(S1)}$$

$$\mathrm{DDo(S0)} = \{\ \}$$

$$\mathrm{DPo(S0)} = \mathrm{DPo(S1)} \cup \mathrm{DDo(S1)}.$$

d) REPEAT

$$\mathrm{DDi(S0)} = \mathrm{DDi(S1)} \cup \{\mathrm{DDi(E1)} - \mathrm{DDo(S1)}\}$$

$$\mathrm{DDo(S0)} = \mathrm{DDo(S1)}$$

$$\mathrm{DPo(S0)} = \mathrm{DPo(S1)}.$$

e) CASE

$$\mathrm{DDi(S0)} = \mathrm{DDi(E1)} \cup \{\bigcup_{1 \leqslant i \leqslant n} \mathrm{DDi}(S_i)\}$$

$$\mathrm{DDo(S0)} = \bigcap_{1 \leqslant i \leqslant n} \mathrm{DDo}(S_i)$$

$$\begin{aligned}\mathrm{DPo(S0)} &= \{\bigcup_{1 \leqslant i \leqslant n} \mathrm{DPo}(S_i)\} \cup \{\bigcup_{1 \leqslant i \leqslant n} \mathrm{DDo}(S_i) - \bigcap_{1 \leqslant i \leqslant n} \mathrm{DDo}(S_i)\} \\ &= \{\bigcup_{1 \leqslant i \leqslant n} (\mathrm{DPo}(S_i) \cup \mathrm{DDo}(S_i))\} - \mathrm{DDo(S0)}.\end{aligned}$$

f) FOR

$$\mathrm{DDi(S0)} = \mathrm{DDi(I1)} \cup \mathrm{DDi(E1)} \cup \mathrm{DDi(S1)}$$

$$\mathrm{DDo(S0)} = \{\ \}$$

$$\mathrm{DPo(S0)} = \mathrm{DDo(S1)} \cup \mathrm{DPo(S1)}.$$

The Control Flow and Data Flow Within Simple Blocks

To describe the control flow within a simple block, we identify the simple statements and the order in which they are executed. This can be done using a COMPOUND graph where the statement nodes are assignment statements. To describe data flow within simple blocks, the node sets are constructed for each simple statement in a simple block. The construction of data arcs and data-arc sets is done in the same way for a sequence of statements in a COMPOUND graph.

Data flow within a simple statement may be described using relations or a parse tree, as is used in many compilers [36]. The exact representation will depend upon the way in which the substatement information is to be used and time/space constraints of the computing environment in which the model is implemented.

The Representation of Program Objects

Program objects are represented by Codd relations. These relations are defined in terms of the production rules of the grammar of the implementation language in the manner described in Section II. For example, a set of relations is used to describe user-defined constants, as in Pascal.

The Operations Defined on the Model

In this section we will define the operations which involve the creation, deletion, and replacement of nodes, arcs, and labels in the model. The purpose of these operations is to provide a means of manipulating the structures defined in the model. They are purposely low-level and are not intended to be seen by the maintenance programmer who is modifying the program being modeled. The user should only see higher level operations which are built on top of these low-level operations.

The operations are defined as follows, with simple blocks, expressions, and labels assumed to be atomic.

To Create a Node:

1) CRCM(NID): Create node NID as a COMPOUND SCG.

2) CRIF(NID,EXP,THEN,ELSE): Create node NID as an IF SCG, expression EXP, then clause THEN, and else clause ELSE.

3) CRR(NID,EXP,BODY): Create node NID as a REPEAT SCG, expression EXP, and body BODY.

4) CRW(NID,EXP,BODY): Create node NID as a WHILE SCG, expression EXP, and body BODY.

5) CRC(NID, CLIST, EXP, ELEMENT): Create node NID as a CASE SCG, case label list CLIST, expression EXP and one element ELEMENT.

To Insert a Node:

6) INCM(G,NID,SEQ): To insert a node NID into COMPOUND SCG G in position SEQ.

7) INC(G, NID, CLIST): To insert a node NID into CASE SCG G with case label list CLIST.

To Delete a Node:

8) DLN(NID): To delete node NID.

To Replace a Node:

9) RPN(G,OLDN,NEWN): To replace node OLDN with node NEWN in SCG G.

To Insert an Arc:

10) INCA(G,AID,NID1,NID2): To insert control arc AID from node NID1 to node NID2 in SCG G.

11) INDA(G,AID,NID1,NID2): To insert data arc AID from node NID1 to node NID2 in SDG G.

To Delete an Arc:

12) DLA(G,AID): To delete arc AID in graph G.

To Label an Arc:

13) LBA(G,AID,LABEL): To label arc AID with label LABEL in graph G.

To Delete a Label:

14) DLL(G,AID): To delete the label on arc AID in graph G.

When one of these operations is specified, information from the database is used to determine if the modification would leave the graph structurally complete. This is done using a decision table for each type of statement graph which contains the possible structural *conditions*, the *operations*, and the *actions* taken for each possible condition-operation pair.

IV. An Example

In this section, we will construct a part of the model for the example Pascal program SAMPLE shown in Fig. 4. SAMPLE reads an array of 20 integers and finds the minimum and the maximum. The integers are read from one line of the text. If there are less than 20 integers on the input line, an error message is written and the program halts. The labels to the right of the program text in Fig. 4 give the statement number, simple-block number, or the expression number of the particular line. For example, line 22 contains expression E11, where E11 is a part of S11. Simple block B3 on line 23 is also a part of S11.

Figs. 5-9 give the RG representation for a part of the model of program SAMPLE; Tables I-VIII give the relational representations. The RG representation for the control flow in SAMPLE is shown in Figs. 5-8 and Table I gives the relational representation. Fig. 9 is the RG form for the data flow in procedure MINMAX; Table II is a part of the corresponding relational representation. Table III is the relational representation for identifiers in SAMPLE. Table IV contains the relations used to describe program and procedure declarations and Table V shows the relations for label and constant definitions. The relations for type definitions are in Table VI and variable declarations are represented by Table VII. Table VIII contains the relational representation for statements. The numbers in the nodes of Figs. 5 and 6 identify the keys used to locate the definition or declaration part in the associated relation. For example, in Fig. 5 the key for "VAR-PRT" is 1; this identifies a set of tuples in relation "VAR-PRT" in Table VII.

The control flow in procedure MINMAX is represented as two relations, one for nodes and one for arcs. In relation "NODE," "NTYPE" specifies the type of node and "NPTR" gives the node in which the given node exists. For example, node S5 is a WHILE statement node which is contained in node S1. In relation "CARC," the start and the end of each control arc are given. For example, arc 4 begins at node E5 and ends at node S6; the condition associated with arc 4 is E5. The arc associated with the negation of E5 has ~E5 in the "LABEL" column.

The data flow for a part of MINMAX is represented by the tables in Table II, representing node sets, data arcs, data sets, and module sets. For example, node S24 has {G,J,N} for its input set, {J} for its possible output set, and no definite out-

```
 1  PROGRAM SAMPLE(INPUT,OUTPUT);
 2  LABEL  10;
 3  CONST        N = 20;
 4  TYPE      LIST = ARRAY[1..N] OF INTEGER;
 5           RANGE = 0..500;
 6  VAR        A,B : LIST;
 7       COUNT,MIN,MAX : RANGE;
 8
 9  PROCEDURE MINMAX(VAR G: LIST; VAR J,K:INTEGER);
10  VAR   I : 1..N;
11      U,V : RANGE;
12  BEGIN                                                     S1
13       J:=G[1];                                         B1  S1
14       K:=J;                                            B1  S1
15       I:=2;                                            B1  S1
16       WHILE I<N DO                                 E5  S5  S1
17       BEGIN                                        S6  S5  S1
18            U:=G[I];                            B2  S6  S5  S1
19            V:=G[I+1];                          B2  S6  S5  S1
20            IF U>V                          E10 S9  S6  S5  S1
21               THEN BEGIN                   S10 S9  S6  S5  S1
22                    IF U>K          E11 S11 S10 S9  S6  S5  S1
23                       THEN K:=U;   B3  S11 S10 S9  S6  S5  S1
24                    IF V<J          E13 S13 S10 S9  S6  S5  S1
25                       THEN J:=V    B4  S13 S10 S9  S6  S5  S1
26                    END                     S10 S9  S6  S5  S1
27               ELSE BEGIN                   S15 S9  S6  S5  S1
28                    IF V>K          E15 S16 S15 S9  S6  S5  S1
29                       THEN K:=V;   B5  S16 S15 S9  S6  S5  S1
30                    IF U<J          E17 S18 S15 S9  S6  S5  S1
31                       THEN J:=U    B6  S18 S15 S9  S6  S5  S1
32                    END;                    S15 S9  S6  S5  S1
33            I:=I+2                              B7  S6  S5  S1
34       END;                                     S6  S5  S1
35       IF I=N                                   E20 S21 S1
36          THEN IF G[N]>K                    E21 S22 S21 S1
37                  THEN K:=G[N]              B8  S22 S21 S1
38                  ELSE IF G[N]<J        E25 S24 S22 S21 S1
39                          THEN J:=G[N]  B9  S24 S22 S21 S1
40  END; {MINMAX}                                             S1
41  BEGIN                                                     S26
42       COUNT:=0;                                        B10 S26
43       REPEAT BEGIN                             S40 S28 S26
44              COUNT:=COUNT+1;               B11 S40 S28 S26
45              IF EOLN                   E31 S30 S40 S28 S26
46                 THEN BEGIN             S31 S30 S40 S28 S26
47                      WRITELN(´ERROR´);S32  S31 S30 S40 S28 S26
48                      GOTO 10       B12 S31 S30 S40 S28 S26
49                      END;              S31 S30 S40 S28 S26
50              READ(A[COUNT]);               S34 S40 S28 S26
51              WRITE(A[COUNT]:3)             S35 S40 S28 S26
52              END                               S40 S28 S26
53       UNTIL COUNT=N;                           E37 S28 S26
54       WRITELN;                                         S36 S26
55       MINMAX(A,MIN,MAX);                               S37 S26
56       WRITELN(MIN,MAX,MAX-MIN);                        S38 S26
57  10:                                                   S39 S26
58  END.                                                      S26
```

Fig. 4. The example Pascal program SAMPLE.

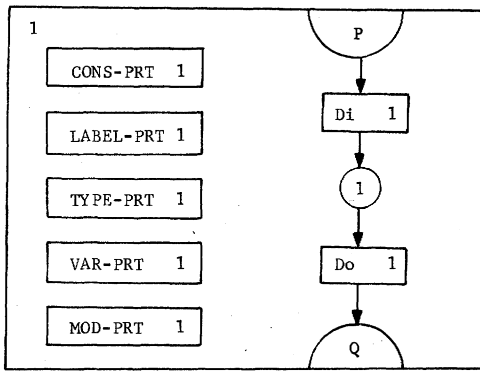

Fig. 5. RG representation for the uppermost level of program SAMPLE.

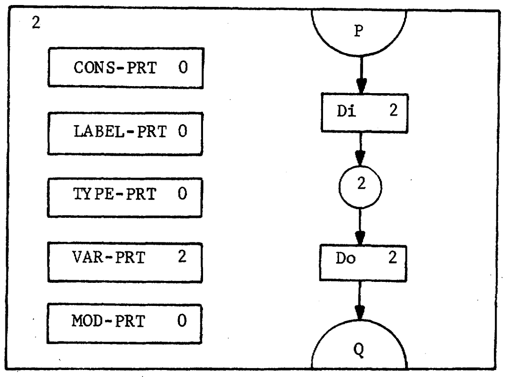

Fig. 6. RG representation for the uppermost level of procedure MINMAX.

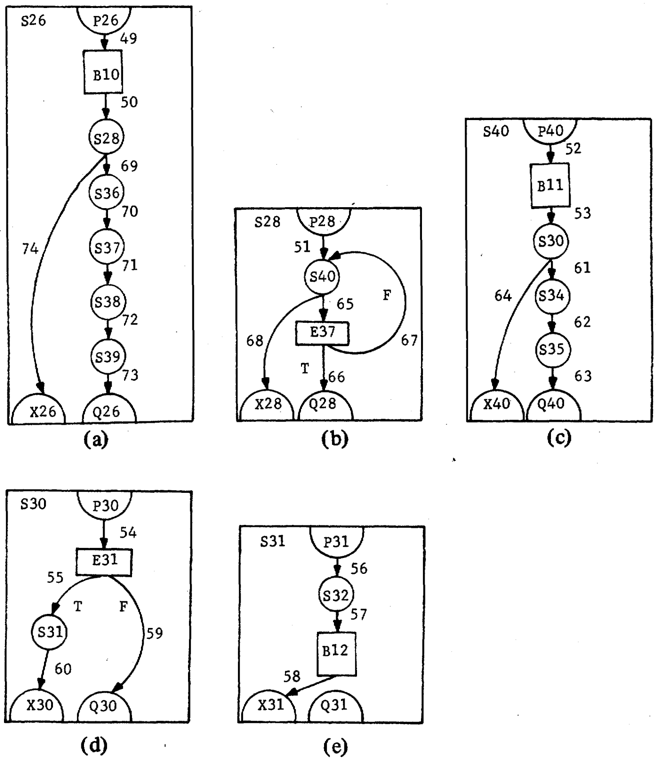

Fig. 7. SCG's for the statement block of SAMPLE.

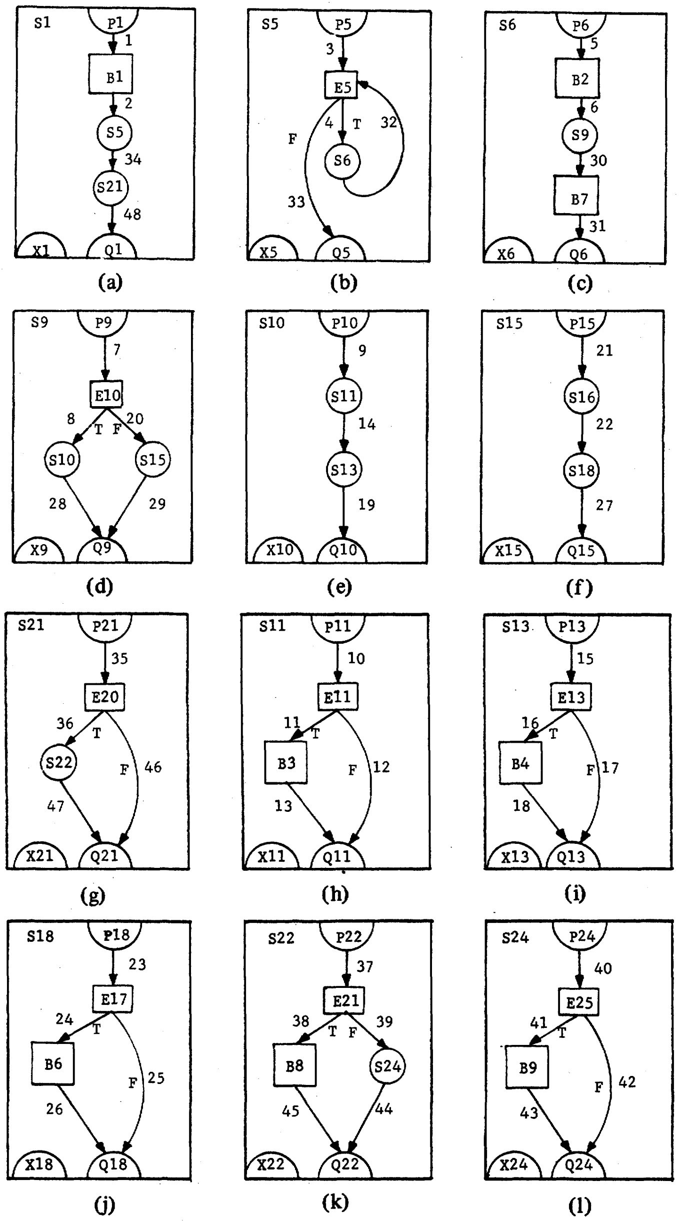

Fig. 8. The SCG's for the statement block of MINMAX.

put. Data arc 6 starts at P22 and ends at S24 with data set {G,J,N}.

Relations "IDENT" and "IDENT-LST" in Table III represent identifiers in SAMPLE. The first six tuples in "IDENT" are predefined indentifiers in Pascal. The other identifiers are user-defined. Table IV contains the relations used to describe program and procedure declarations. Relation "BLOCK" corresponds to the RG in Fig. 1.

In the relations of Tables IV-VIII, the name of the attribute or the value of a "CLASS" identifies the relation to which the attribute refers. The value of the attribute identifies a tuple or a collection of tuples in the referenced relation. For example, in (d) of Table IV, attribute "TYPE-PRT-ID" refers to relation "TYPE-PRT" in Table VI; the value 1 for "TYPE-PRT-ID" in relation "BLOCK" identifies two tuples in relation "TYPE-PRT." In relation "TYPE" in Table VI, the tuple with "TYPE-ID" equal to 3 identifies the tuple in relation "SUBR" with key "SUBR-ID" equal to 2.

V. Discussion

The representation for a program presented in this paper is part of an effort to develop a comprehensive model for software which can clearly specify the objects and relations involved in any analysis or modifications to that software. The comprehensive model will form a foundation for developing effective maintenance techniques for large-scale software, including generation of modification proposals, ripple effect analysis, and testing during the software maintenance phase.

We have altered a Pascal compiler which runs on a DEC VAX 11/780 computer to construct the relations given in the Appendix plus the relations for the syntactic units which make up expressions. These relations were then used to construct the relations given by Tables I, II, and VIII (c) in the example given in Section IV. We are continuing to refine this implementation and to define the relations within a relational database system.

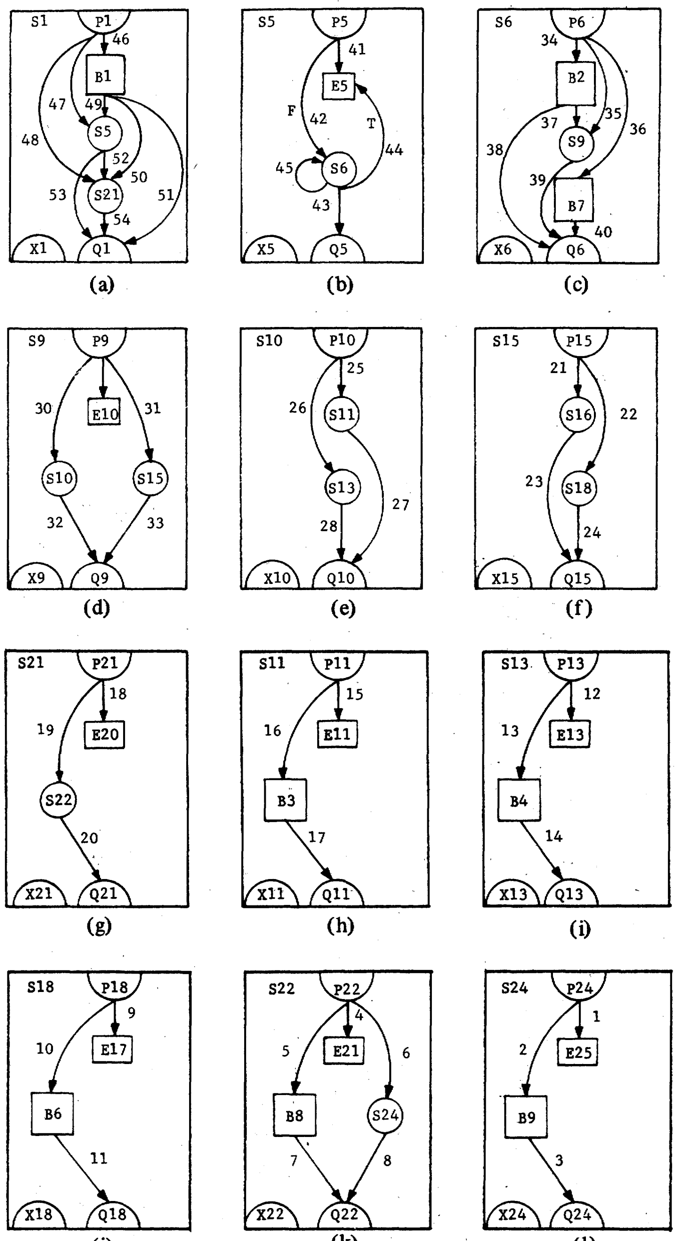
Fig. 9. The SDG's for the statement block of MINMAX.

TABLE I
THE CONTROL-FLOW RELATIONS FOR PROCEDURE MINMAX IN SAMPLE

(a) NODE : Nodes

NODE-ID	NTYPE	NPTR
S1	CMPND	M1
S5	WHILE	S1
S6	CMPND	S5
S9	IF	S6
S10	CMPND	S9
S11	IF	S10
S13	IF	S10
S15	CMPND	S9
S16	IF	S15
S18	IF	S15
S21	IF	S1
S22	IF	S21
S24	IF	S22
E5	EXP	S5
E10	EXP	S9
E11	EXP	S11
E13	EXP	S13
E15	EXP	S16
E17	EXP	S18
E20	EXP	S21
E21	EXP	S22
E25	EXP	S24
B1	BLK	S1
B2	BLK	S6
B3	BLK	S11
B4	BLK	S13
B5	BLK	S16
B6	BLK	S18
B7	BLK	S6
B8	BLK	S22
B9	BLK	S24

S : STMT NODE
M : MOD NODE
B : BLK NODE
E : EXP NODE

(b) CARC : Control arcs

CARC-ID	BEGIN	END	LABEL
1	P1	B1	
2	B1	S5	
3	P5	E5	
4	E5	S6	E5
5	P6	B2	
6	B2	S9	
7	P9	E10	
8	E10	S10	E10
9	P10	S11	
10	P11	E11	
11	E11	B3	E11
12	E11	Q11	~E11
13	B3	Q11	
14	S11	S13	
15	P13	E13	
16	E13	B4	E13
17	E13	Q13	~E13
18	B4	Q13	
19	S13	Q10	
20	E10	S15	~E10
21	P15	S16	
22	S16	S18	
23	P18	E17	
24	E17	B6	E17
25	E17	Q18	~E17
26	B6	Q18	
27	S18	Q15	
28	S10	Q9	
29	S15	Q9	
30	S9	B7	
31	B7	Q6	
32	S6	E5	
33	E5	Q5	~E5
34	S5	S21	
35	P21	E20	
36	E20	S22	E20
37	P22	E21	
38	E21	B8	E21
39	E21	S24	~E21
40	P24	E25	
41	E25	B9	E25
42	E25	Q24	~E25
43	B9	Q24	
44	S24	Q22	
45	B8	Q22	
46	E20	Q21	~E20
47	S22	Q21	
48	S21	Q1	

Future research needs to be done to represent expressions in a way which is efficient and is compatible with the relational representation. We also need to experiment with the set of relations to see which relations can be combined or split. These problems will not be resolved until we use the model for program analysis. Then the tradeoffs will become more clear and we can more easily assess the effect of a particular set of relations on the model's performance. Furthermore, we need to model the environment in which the program is run, namely, the interaction with external files and other programs.

We have begun to define the operations on the model in terms of SQL, a variation of SEQUEL2 [37] which lies between relational calculus and relational algebra. This is the language used by the database system which we have on the DEC VAX computer. We are also in the process of defining operations which modify program objects which are not statements, such as type definitions. Our current set of operations is only for statement constructs. Data-flow algorithms commonly used in program optimization [5], [6] and algorithms which check for completeness and consistency are being written in terms of the abstractions used in the model. Also, we are developing a uniform way of representing interactions with external files.

There are limitations to our model. Some of them are inherent in the abstractions used, others are by choice, and some stem from problems which are not easily solved. For example, using many relations to represent a program can cause "information explosion." This, however, is a common problem in relational database systems, and hence the problem is not unique to the model. There are also performance problems which are related to the relational representation. However, this again is really a database problem, more related to the physical organization of the information rather than to its logical structure. Another limitation is self-imposed, namely, the decision to not allow function calls with side effects. Our reasons for this restriction stem from a desire to keep the data-flow representation as simple as possible. Function calls with side effects, although commonly used by programmers, are

TABLE II
THE DATA-FLOW REPRESENTATION FOR PROCEDURE MINMAX IN SAMPLE

(a) NODE : Node sets for procedure MINMAX in SAMPLE.

NODE-ID	DI	DPo	DDo
B1	{G}	--	{I,J,K}
B2	{G,I}	--	{U,V}
B3	{U}	--	{K}
B4	{V}	--	{J}
B5	{V}	--	{K}
B6	{U}	--	{J}
B7	{I}	--	{I}
B8	{G,N}	--	{K}
B9	{G,N}	--	{J}
E5	{I,N}	--	--
E10	{U,V}	--	--
E11	{K,U}	--	--
E13	{J,V}	--	--
E15	{K,V}	--	--
E17	{J,U}	--	--
E20	{I,N}	--	--
E21	{G,K,N}	--	--
E25	{G,J,N}	--	--
S11	{K,U}	{K}	{ }
S13	{J,V}	{J}	{ }
S10	{J,K,U,V}	{J,K}	{ }
S16	{K,V}	{K}	{ }
S18	{J,U}	{J}	{ }
S15	{J,K,U,V}	{J,K}	{ }
S9	{J,K,U,V}	{J,K}	{ }
S6	{G,I,J,K}	{J,K}	{I,U,V}
S5	{G,I,J,K,N}	{I,J,K,U,V}	{ }
S24	{G,J,N}	{J}	{ }
S22	{G,J,K,N}	{J,K}	{ }
S21	{G,I,J,K,N}	{J,K}	{ }
S1	{G,N}	{U,V}	{I,J,K}

(b) DARC : Some data arcs and their data sets

DARC-ID	DEFINED	USED	DATA-SET
1	P24	E25	{G,J,N}
2	P24	B9	{G,N}
3	B9	Q24	{J}
4	P22	E21	{G,K,N}
5	P22	B8	{G,N}
6	P22	S24	{G,J,N}
7	B8	Q22	{K}
8	S24	Q22	{J}
9	P18	E17	{J,U}
10	P18	B6	{U}
11	B6	Q18	{J}

(c) Di : MODULE INPUT SET

Di-ID	Di-SET
1	{A}
2	{G,N}

(d) Do : MODULE OUTPUT SET

Do-ID	Do-SET
1	{MIN,MAX}
2	{J,K}

TABLE III
THE RELATIONS FOR IDENTIFIERS IN SAMPLE

(a) IDENT : IDENTIFIER

IDENT-ID	IDENTIFIER	CLASS
1	INTEGER	PRE
2	EOLN	PRE
3	READ	PRE
4	READLN	PRE
5	WRITE	PRE
6	WRITELN	PRE
7	SAMPLE	PROG
8	INPUT	FILE
9	OUTPUT	FILE
10	N	CONS
11	LIST	TYPE
12	RANGE	TYPE
13	A	VAR
14	B	VAR
15	COUNT	VAR
16	MIN	VAR
17	MAX	VAR
18	MINMAX	PROC
19	G	FPRM
20	J	FPRM
21	K	FPRM
22	I	VAR
23	U	VAR
24	V	VAR

(b) IDENT-LST : IDENTIFIER LIST

IDENT-LST-ID	SEQ	IDENT-ID
1	1	13
1	2	14
2	1	15
2	2	16
2	3	17
3	1	22
4	1	23
4	2	24
5	1	8
5	2	9

TABLE IV
THE RELATIONS FOR PROGRAM AND PROCEDURE DEFINITIONS IN SAMPLE

(a) PROG : PROGRAM

PROG-ID	IDENT-ID	IDENT-LST-ID	BLOCK-ID
1	7	5	1

(b) MOD-PRT : PROCEDURE/FUNCTION DECLARATION PART

MOD-PRT-ID	**SEQ**	CLASS	CLASS-ID
1	1	PROC	1

(c) PROC : PROCEDURE

PROC-ID	IDENT-ID	FPRM-LST-ID	BLOCK-ID
1	18	1	2

(d) BLOCK

BLOCK-ID	LABEL-PRT-ID	CONS-PRT-ID	TYPE-PRT-ID	VAR-PRT-ID	MOD-PRT-ID	STMT-PRT-ID	Di	Do
1	1	1	1	1	1	1	1	1
2	0	0	0	2	0	2	2	2

(e) FPRM-LST : FORMAL PARAMETER LIST

FPRM-LST-ID	**SEQ**	PAR-TYPE	CLASS	CLASS-ID
1	1	P-VAR	PARAM-GROUP	1
1	2	P-VAR	PARAM-GROUP	2

(f) PARAM-GROUP : PARAMETER GROUP

PARAM-GROUP-ID	**SEQ**	IDENT-ID1	IDENT-ID2
1	1	19	11
2	1	20	1
2	2	21	1

TABLE V
THE RELATIONS FOR LABEL AND CONSTANT DEFINITIONS IN SAMPLE

(a) LABEL-PRT : LABEL PART

LABEL-PRT-ID	**LABEL**
1	10

(b) CONS-PRT : CONSTANT PART

CONS-PRT-ID	**SEQ**	IDENT-ID	CONS-ID
1	1	10	1

(c) CONS : CONSTANT

CONS-ID	CLASS	CLASS-ID
1	INT	1
2	INT	2
3	IDENT	10
4	INT	3
5	INT	4
6	INT	5
7	IDENT	10

(d) INT : UNSIGNED INTEGER

INT-ID	LITERAL
1	20
2	1
3	0
4	500
5	1

TABLE VI
THE RELATIONS FOR TYPE DEFINITIONS IN SAMPLE

(a) TYPE-PRT : TYPE DEFINITION PART

TYPE-PRT-ID	**SEQ**	IDENT-ID	TYPE-ID
1	1	11	1
1	2	12	3

(b) TYPE

TYPE-ID	CLASS	CLASS-ID
1	STRUC	1
2	IDENT	1
3	SUBR	2
4	IDENT	11
5	IDENT	12
6	SUBR	3
7	IDENT	12

(c) SUBR : SUBRANGE TYPE

SUBR-ID	CONS-ID1	CONS-ID2
1	2	3
2	4	5
3	6	7

(d) STRUC : STRUCTURE TYPE

STRUC-ID	PACKED	CLASS	CLASS-ID
1	NO	ARRAY-TYPE	1

(e) ARRAY-TYPE

ARRAY-TYPE-ID	**SEQ**	CLASS	CLASS-ID	TYPE-ID
1	1	SUBR	1	2

TABLE VII
THE RELATIONS FOR VARIABLE DECLARATIONS IN SAMPLE

(a) VAR-PRT : VARIABLE DECLARATION PART

VAR-PRT-ID	**SEQ**	VAR-DECL-ID
1	1	1
2	2	2
3	1	3
4	2	4

(b) VAR-DECL : VARIABLE DECLARATION

VAR-DECL-ID	IDENT-LST-ID	TYPE-ID
1	1	4
2	2	5
3	3	6
4	4	7

TABLE VIII
THE RELATIONS FOR THE STATEMENT BLOCK IN SAMPLE

(a) STMT-PRT : STATEMENT PART

STMT-PRT-ID	CMPND-STMT-ID
1	5
2	1

(b) STMT : STATEMENT

STMT-ID	LABEL-ID	CLASS	CLASS-ID
1	0	CMPND	1
2	0	ASSIGN	1
3	0	ASSIGN	2
4	0	ASSIGN	3
5	0	WHILE	1
6	0	CMPND	2
7	0	ASSIGN	4
8	0	ASSIGN	5
9	0	IF	1
10	0	CMPND	3
11	0	IF	2
12	0	ASSIGN	6
13	0	IF	3
14	0	ASSIGN	7
15	0	CMPND	4
16	0	IF	4
17	0	ASSIGN	8
18	0	IF	5
19	0	ASSIGN	9
20	0	ASSIGN	10
21	0	IF	6
22	0	IF	7
23	0	ASSIGN	11
24	0	IF	8
25	0	ASSIGN	12
26	0	CMPND	5
27	0	ASSIGN	13
28	0	REPEAT	1
29	0	ASSIGN	13
30	0	IF	9
31	0	CMPND	6
32	0	WRITE	1
33	0	GOTO	1
34	0	READ	1
35	0	WRITE	2
36	0	WRITE	3
37	0	P-STMT	1
38	0	WRITE	4
39	1	EMPTY	1
40	0	CMPND	7

(c) BLK : SIMPLE BLOCK

BLK-ID	SEQ	STMT-ID
1	1	2
1	2	3
1	3	4
2	1	7
2	2	8
3	1	12
4	1	14
5	1	17
6	1	19
7	1	20
8	1	23
9	1	25
10	1	27
11	1	29
12	1	33

(d) VAR : VARIABLE

VAR-ID	CLASS	CLASS-ID
1	IDENT	20
2	IDENT	19
3	INDEX	1
4	IDENT	21
5	IDENT	22
6	IDENT	23
7	INDEX	2
8	IDENT	24
9	INDEX	3
10	INDEX	4
11	INDEX	5
12	INDEX	6
13	INDEX	7
14	IDENT	15
15	IDENT	13
16	INDEX	8
17	INDEX	9
18	IDENT	16
19	IDENT	17

(e) INDEX : INDEXED VARIABLE

INDEX-ID	VAR-ID	EXP-LST-ID
1	2	1
2	2	2
3	2	3
4	2	4
5	2	5
6	2	6
7	2	7
8	15	8
9	15	9

(f) EXP-LST : EXPRESSION LIST

EXP-LST-ID	SEQ	EXP-ID
1	1	2
2	1	7
3	1	9
4	1	22
5	1	24
6	1	26
7	1	28
8	1	33
9	1	35

(g) EXP : EXPRESSION

EXP-ID	LITERAL
1	G[1]
2	1
3	J
4	2
5	I<N
6	G[I]
7	I
8	G[I+1]
9	I+1
10	U>V
11	U>K
12	U
13	V<J
14	V
15	V>K
16	V
17	U<J
18	U
19	I+2
20	I=N
21	G[N]>K
22	N
23	G[N]
24	N
25	G[N]<J
26	N
27	G[N]
28	N
29	0
30	COUNT+1
31	EOLN
32	A[COUNT]
33	COUNT
35	A[COUNT]
36	COUNT
37	COUNT=N
38	A
39	MIN
40	MAX

(h) P-STMT : PROCEDURE STATEMENT

P-STMT-ID	IDENT-ID	ACT-PAR-LST-ID
1	18	1

(i) ACT-PAR-LST : ACTUAL PARAMETER LIST

ACT-PAR-LST-ID	SEQ	CLASS	CLASS-ID
1	1	EXP	38
1	2	EXP	39
1	3	EXP	40

(j) GOTO : GOTO STATEMENT

GOTO-ID	LABEL-ID	STMT-ID
1	1	39

(k) EMPTY : EMPTY STATEMENT

EMPTY-ID
1

(l) IF : IF STATEMENT

IF-ID	EXP-ID	STMT-ID1	STMT-ID2
1	10	10	15
2	11	12	0
3	13	14	0
4	15	17	0
5	17	19	0
6	20	22	0
7	21	23	24
8	25	25	0
9	31	31	0

(m) WHILE : WHILE STATEMENT

WHILE-ID	EXP-ID	STMT-ID
1	5	6

(n) REPEAT : REPEAT STATEMENT

REPEAT-ID	EXP-ID	STMT-LST-ID
1	37	1

(o) STMT-LST : STATEMENT LIST

STMT-LST-ID	SEQ	CLASS	CLASS-ID
1	1	BLK	11
1	2	STMT	30
1	3	STMT	34
1	4	STMT	35

(p) CMPND : COMPOUND STATEMENT

CMPND-ID	SEQ	CLASS	CLASS-ID
1	1	BLK	1
1	2	STMT	5
1	3	STMT	21
2	1	BLK	2
2	2	STMT	9
2	3	BLK	7
3	1	STMT	11
3	2	STMT	13
4	1	STMT	16
4	2	STMT	18
5	1	BLK	10
5	2	STMT	28
5	3	STMT	36
5	4	STMT	37
5	5	STMT	38
6	1	STMT	32
6	2	BLK	12
7	1	BLK	11
7	2	STMT	30
7	3	STMT	34
7	4	STMT	35

(q) ASSIGN : ASSIGN STATEMENT

ASSIGN-ID	CLASS	CLASS-ID	EXP-ID
1	VAR	1	1

not necessarily desirable from the standpoint of achieving reliable and maintainable software. Although we cannot prove our contention, we believe that such a restriction is reasonable. There are also limitations with the data-flow representation itself, since resolution is only to the variable identifier (i.e., the individual elements of an array are not distinguished). However, this is a common problem with data-flow algorithms in general.

Other work for developing the model include the use of interactive graphics, a language-oriented editor, the ability to produce the program text for a modeled program without storing the code as a text file, and the extension of the model to handle systems of programs by specifying the individual environment for each program.

Appendix
Relations Derived from a BNF Form of Pascal

The following relations were derived from the BNF form of Pascal from [35]. The production rules which were used to define the relation precede the relation. In most cases more than one production rule was used to define a relation. The primary key is in boldface for each relation. If there is more than one attribute in a primary key, these attributes are ordered from left to right. When an atribute has a name with "-ID" at the end (e.g., TYPE-ID), it means that the value for that attribute specifies the key value of a tuple in the relation identified by the attribute name (e.g., TYPE). The attributes with an asterisk (*) are attributes which are unique to the model and are not directly derived from the production rules.

<program> ::= <program heading> <block>
<program heading> ::= **program** <identifier> (<file identifier> {,<file identifier>});
<file identifier> ::= <identifier>

(1) PROG | **PROG-ID** IDENT-LST-ID BLOCK-ID

<identifier> ::= LITERAL

(2) IDENT | **IDENT-ID** IDENTIFIER CLASS*

CLASS : {PRE, FILE, CONS, FLD, TYPEVAR, PROG, FUNC, PROG}

(3) IDENT-LST | **IDENT-LIST-ID SEQ** IDENT-ID

<block> ::= <label declaration part> <constant definition part>
<type definition part> <variable declaration part>
<procedure and function declaration part>
<statement part>

(4) BLOCK | **BLOCK-ID** LABEL-ID CONS-PRT-ID TYPE-PRT-ID
VAR-PRT-ID MOD-PRT-ID STMT-PRT-ID Di* Do*

<label declaration part> ::= <empty> | **label** <label> {,<label>};
<label> ::= <unsigned integer>

(5) LABEL-PRT | **LABEL-PRT-ID INT-ID**

<constant definition part> ::= <empty> |
const <constant definition>
{;<constant definition>};
<constant definition> ::= <identifier> = <constant>

(6) CONS-PRT | **CONS-PRT-ID SEQ** IDENT-ID CONS-ID

<constant> ::= <unsigned number> | <sign> <unsigned number> |
<constant identifier> |
<sign> <constant identifier> | <string>
<unsigned number> ::= <unsigned integer> | <unsigned real>
<sign> ::= + | -
<constant identifier> ::= <identifier>

(7) CONS | **CONS-ID** SIGN CLASS CLASS-ID

SIGN : {+,-};
CLASS : {INT,REAL,STRING,IDENT}

<unsigned integer> ::= LITERAL

(8) INT | **INT-ID** LITERAL

<unsigned real> ::= LITERAL

(9) REAL | **REAL-ID** LITERAL

<string> ::= LITERAL

(10) STRING | **STRING-ID** LITERAL

<type definition part> ::= <empty> | **type** <type definition> {;<type definition>};

<type definition> ::= <identifier> = <type>

(11) TYPE-PRT | **TYPE-PRT-ID SEQ** IDENT-ID TYPE-ID

<type> ::= <simple type> | <structured type> | <pointer type>

<simple type> ::= <scalar type> | <subrange type> | <type identifier>

<type identifier> ::= <identifier>

(12) TYPE | **TYPE-ID** CLASS CLASS-ID

CLASS : {SCALR,SUBR,IDENT,STRUC,PTR-TYPE};

<scalar type> ::= (<identifier> {, <identifier>})

(13) SCALAR | **SCALR-ID** IDENT-LST-ID

<subrange type> ::= <constant>..<constant>

(14) SUBR | **SUBR-ID** CONS-ID 1 CONS-ID 2

<structured type> ::= <unpacked structured type> | **packed** <unpacked structured type>

<unpacked structured type> ::= <array type> | <record type> | <set type> | <file type>

(15) STRUC | **STRUC-ID** PACKED CLASS CLASS-ID

PACKED : {YES,NO};
CLASS : {ARRAY-TYPE,REC-TYPE,SET-TYPE,FILE-TYPE};

<array type> ::= **array** [<index type> {, <index type>}] **of** <component type>

<index type> ::= <simple type>

<component type> ::= <type>

<simple type> ::= –see above–

<type identifier> ::= –see above–

(16) ARRAY-TYPE | **ARRAY-TYPE-ID SEQ** CLASS CLASS-ID TYPE-ID

CLASS : {SCALR,SUBR,IDENT};

<record type> ::= **record** <field list> **end**

<field list> ::= <fixed part> | <fixed part> ;<variant part> |
<variant part>

(17) REC-TYPE | **REC-TYPE-ID** FIX-PRT-ID VARIANT-PRT-ID

<fixed part> ::= <record section> {;<record section>}

<record section> ::= <field identifier> {,<field identifier>} :
<type> | <empty>

<field identifier> ::= <identifier>

(18) FIX-PRT | **FIX-PRT-ID SEQ** IDENT-LST-ID TYPE-ID

<variant part> ::= **case** <tag field> <type identifier> **of**
<variant> {;<variant>}

<type identifier> ::= –see above–

<tag field> ::= <field identifier> : | <empty>

<field identifier> ::= –see above–

(19) VARIANT-PRT | **VARIANT-PRT-ID** IDENT-ID1 IDENT-ID2 VARIANT-LST-ID

<variant> ::= <case label list> : (<field list>) | <empty>
<field list> ::= –see above–

(20) VARIANT-LST | **VARIANT-LST-ID SEQ** CASE-LABL-LST-ID
FIX-PRT-ID VARIANT-PRT-ID

<case label list> ::= <case label> {,<case label>}

<case label> ::= <constant>

(21) CASE-LABL-LST | **CASE-LABL-LST-ID SEQ** CONS-ID

<set type> ::= **set of** <base type>

<base type> ::= <simple type>

<simple type> ::= –see above–

<type identifier> ::= –see above–

(22) SET-TYPE | **SET-TYPE-ID** CLASS CLASS-ID

CLASS : {SCALR,SUBR,IDENT};

<file type> ::= **file of** <type>

(23) FILE-TYPE | **FILE-TYPE-ID** TYPE-ID

<pointer type> ::= <type identifier>
<type identifier> ::= –see above–

(24) PTR-TYPE | **PTR-TYPE-ID** IDENT-ID

<variable declaration part> ::= <empty> |
var <variable declaration> {;<variable declaration>};

(25) VAR-PRT | **VAR-PRT-ID SEQ** VAR-DECL-ID

<variable declaration> ::= <identifier> {,<identifier>} : <type>

(26) VAR-DECL | **VAR-DECL-ID** IDENT-LST-ID TYPE-ID

<procedure and function declaration part> ::=
{<procedure or function declaration> ;}
<procedure or function declaration> ::= <procedure declaration> |
<function declaration>

(27) MOD-PRT | **MOD-PRT-ID SEQ** CLASS CLASS-ID

CLASS : {PROC,FUNC}
<procedure declaration> ::= <procedure heading> <block>
<procedure heading> ::= **procedure** <identifier> ; |
procedure <identifier> (<formal parameter section>
{;<formal parameter section> }) ;

(28) PROC | **PROC-ID** IDENT-ID FPRM-LST BLOCK-ID

<formal parameter section> ::= <parameter group> |
var <parameter group> |
function <parameter group> |
procedure <identifier> {, <identifier>}

(29) FPRM-LST | **FPRM-LST-ID SEQ** PAR-TYPE CLASS CLASS-ID

CLASS : {PARAM-GROUP,IDENT-LST};
PAR-TYPE : {NULL,VAR,FUNC,PROC};
<parameter group> ::= <identifier> {, <identifier>} :
<type identifier>
<type identifier> ::= –see above–

(30) PARAM-GROUP | **PARAM-GROUP-ID** IDENT-LST-ID IDENT-ID2

<function declaration> ::= <function heading> <block>
<function heading> ::= **function** <identifier> : <result type> |
function <identifier> (<formal parameter section>
{;<formal parameter section>}) : <result type> ;
<formal parameter section> ::= –see above–
<result type> ::= <type identifier>
<type identifier> ::= –see above–

(31) FUNC | **FUNC-ID** IDENT-ID1 FPRM-LST-ID BLOCK-ID IDENT-ID

<statement part> ::= <compound statement>

(32) STMT-PRT | **STMT-PRT-ID** CMPND-ID

<statement> ::= <unlabeled statement> |
<label> : <unlabeled statement>
<unlabeled statement> ::= <simple statement> |
<structured statement>

```
<simple statement> ::= <assignment statement> |
                             <procedure statement> |
                                 <go to statement> |
                                 <empty statement> |
<structured statement> ::= <compound statement> |
          <conditional statement> | <repetitive statement> |
          <with statement>
<condition statement> ::= <if statement> | <case statement>
<repetitive statement> ::= <while statement> |
                  <repeat statement> | <for statement>
```

(33) STMT | **STMT-ID** LABEL-ID CLASS CLASS-ID

CLASS : {ASSIGN,P-STMT,GOTO,EMPTY,CMPND,
IF,CASE,WHILE,REPEAT,FOR,WITH};

```
<assignment statement> ::= <variable> := <expression> |
                  <function identifier> := <expression>
<function identifier> ::= <identifier>
```

(34) ASSIGN | **ASSIGN-ID** CLASS CLASS-ID EXP-ID

CLASS : {VAR,IDENT};

```
<variable> ::= <entire variable> | <component variable> |
            <referenced variable>
<entire variable> ::= <variable identifier>
<variable identifier> ::= <identifier>
<component variable> ::= <indexed variable> | <field designator> |
                           <file buffer>
<file buffer> ::= <file variable>
<file variable> ::= <variable>
<referenced variable> ::= <pointer variable>
<pointer variable> ::= <variable>
```

(35) VAR | **VAR-ID** CLASS CLASS-ID

CLASS : {IDENT,INDEX,FLD-DSGN,VAR};

```
<indexed variable> ::= <array variable> [ <expression>
          {, <expression> }]
<array variable> ::= <variable>
```

(36) INDEX | **INDEX-ID** VAR-ID EXP-LST-ID

(37) EXP-LST | **EXP-LST-ID SEQ** EXP-ID

```
<expression> ::= LITERAL
```

(38) EXP | **EXP-ID** LITERAL

```
<field designator> ::= <record variable>.<field identifier>
<record variable> ::= <variable>
<field identifier> ::= –see above–
```

(39) FLD-DSGN | **FLD-DSGN-ID** VAR-ID IDENT-ID

<procedure statement> ::= <procedure identifier> |
    <procedure identifier> (<actual parameter>
    {, <actual parameter>})
<procedure identifier> ::= <identifier>

(40) P-STMT | **P-STMT-ID** IDENT-ID ACT-PAR-LST-ID

<actual parameter> ::= <expression> | <variable> |
    <procedure identifier> | <function identifier>
<procedure identifier> ::= –see above–
<function identifier> ::= –see above–

(41) ACT-PAR-LST | **ACT-PAR-LST-ID SEQ** CLASS CLASS-ID

CLASS : {EXP,VAR,IDENT};
<go to statement> ::= **goto** <label>

(42) GOTO | **GOTO-ID** LABEL-ID STMT-ID

<empty statement> ::= <empty>
<empty> ::=

(43) EMPTY | **EMPTY-ID**

<compound statement> ::= **begin** <statement> {;<statement>} **end**

(44) CMPND | **CMPND-ID-SEQ** CLASS* CLASS-ID*

CLASS : {BLK,STMT};
<if statement> ::= **if** <expression> **then** <statement> |
    if <expression> **then** <statement> **else** <statement>

(45) IF | **IF-ID** EXP-ID STMT-ID1 STMT-ID2

<case statement> ::= **case** <expression> **of** <case list element>
    {; <case list element>} **end**

(46) CASE | **CASE-ID** EXP-ID C-LST-ELMT-LST-ID

(47) C-LST-ELMT-LST | **C-LST-ELMT-LST-ID SEQ** C-LST-ELMT-ID

<case list element> :: <case label list> : <statement> |
    <empty>

(48) C-LST-ELMT | **C-LST-ELMT-ID** CASE-LABL-LST-ID STMT-ID

<case label list> ::= –see above–
CASE-LABL-LST : –see above–
<while statement> ::= **while** <expression> **do** <statement>

(49) WHILE | **WHILE-ID** EXP-ID STMT-ID

<repeat statement> ::= **repeat** <statement> {; <statement> }
until <expression>

(50) REPEAT | **REPEAT-ID** EXP-ID STMT-LST-ID

(51) STMT-LST | **STMT-LST-ID SEQ** STMT-ID

<for statement> ::= **for** <control variable> := <for list> **do**

<for list> ::= <initial value> **to** <final value> |
<initial value> **downto** <final value>

<control variable> ::= <identifier>

<initial value> ::= <expression>

<final value> ::= <expression>

(52) FOR | **FOR-ID** IDENT-ID CLASS EXP-ID1 EXP-ID2 STMT-ID

CLASS : {TO,DOWNTO};

<with statement> ::= **with** <record variable list> **do**

<record variable list> ::= <record variable> {, <record variable>}

<record variable> ::= –see above–

(53) WITH | **WITH-ID SEQ** VAR STMT-ID

References

[1] R. C. Linger, H. K. Mills, and B. I. Witt, *Structured Programming: Theory and Practice*. Reading, MA: Addison-Wesley, 1979.

[2] M. R. Paige, "On partitioning program graphs," *IEEE Trans. Software Eng.*, vol. SE-3, pp. 386-393, Nov. 1977.

[3] J. R. Brown and K. F. Fischer, "A graph theoretic approach to the verification of program structures," in *Proc. 3rd Int. Conf. Software Eng.*, May 10-12, 1978, pp. 136-141.

[4] Z. Manna, *Mathematical Theory of Computation*. New York: McGraw-Hill, 1974.

[5] M. S. Hecht, *Flow Analysis of Computer Programs*. Amsterdam, The Netherlands: Elsevier North-Holland, 1977.

[6] M. Schaefer, *A Mathematical Theory of Global Program Optimization*. Englewood Cliffs, NJ: Prentice-Hall, 1973.

[7] T. W. Pratt, "Definition of programming language semantics using grammars for hierarchical graphs," in *Lecture Notes in Computer Science*, vol. 73, G. Goos and J. Hartmanis, Eds. New York: Springer-Verlag, 1979.

[8] —, "A hierarchical graph model of the semantics of programs," in *Proc. Spring Joint Comput. Conf.*, vol. 34, Apr. 1969, pp. 813-825.

[9] —, "Pair grammars, graph languages, and string-to-graph translations," *J. Comput. Syst. Sci.*, vol. 5, pp. 560-595, 1971.

[10] T. L. Kunii and A. P. Buchmann, "Evolutionary drawing formalization in an engineering database environment," in *Proc. COMPSAC '79*, 1979, pp. 732-737.

[11] T. L. Kunii and M. Harada, "A design process formalization," in *Proc. COMPSAC '79*, 1979, pp. 367-373.

[12] —, "SID–A system for interactive design," in *Proc. Nat. Comput. Conf.*, 1980, pp. 30-40.

[13] G. Estrin, "A methodology for design of digital systems–Supported by SARA at the age of one," in *Proc. Nat. Comput. Conf.*, 1978, pp. 313-324.

[14] W. Ruggiero, G. Estrin, R. Fenchel, R. Raxouk, D. Schwabe, and M. Vernon, "Analysis of data flow models using the SARA graph model of behavior," in *Proc. Nat. Comput. Conf.*, vol. 48, 1979, pp. 975-988.

[15] L. D. Fosdick and L. J. Osterweil, "Data flow analysis in software reliability," *ACM Comput. Surveys*, vol. 8, pp. 305-330, Sept. 1976.

[16] —, "The detection of anomalous interprocedural data flow," in *Proc. 2nd Int. Conf. Software Eng.*, Oct. 1976, pp. 624-628.

[17] B. K. Rosen, "High-level data flow analysis," *Commun. Ass. Comput. Mach.*, vol. 20, pp. 712-724, Oct. 1977.

[18] J. M. Barth, "A practical interprocedural data flow analysis algorithm," *Commun. Ass. Comput. Mach.*, vol. 21, pp. 724-736, Sept. 1978.

[19] M. S. Hecht and J. D. Ullman, "A simple algorithm for global data flow analysis problems," *SIAM J. Comput.*, vol. 4, pp. 519-532, 1975.

[20] J. B. Kam and J. D. Ullman, "Global data flow analysis and interactive algorithms," *J. Ass. Comput. Mach.*, vol. 23, pp. 158-171, Jan. 1976.

[21] K. Kennedy, "A comparison of two algorithms for global data flow analysis," *Siam J. Comput.*, vol. 5, pp. 158-180, 1976.

[22] W. A. Babich and M. Jazayeri, "The method of attributes for data flow analysis, Part I and II," *Acta Informatica*, vol. 10, pp. 245-264, 265-272, 1978.

[23] H. A. Sholl and T. L. Booth, "Software performance modeling using computation structures," *IEEE Trans. Software Eng.*, vol. SE-1, pp. 414-420, Dec. 1975.

[24] T. J. Gilkey, J. R. White, and T. L. Booth, "Performance analysis as a software design tool," in *Proc. COMPSAC '77*, Oct. 1977, pp. 428-434.

[25] U. R. Kodres, "Analysis of real-time systems by data flowgraphs," *IEEE Trans. Software Eng.*, vol. SE-4, pp. 169-178, May 1978.

[26] M. Hamilton and S. Zeldin, "Higher order software–A methodology for defining software," *IEEE Trans. Software Eng.*, vol. SE-2, pp. 9-32, Mar. 1976.

[27] T. Bell, D. C. Bixler, and M. E. Dyer, "An extendable approach to computer-aided software requirements engineering," *IEEE Trans. Software Eng.*, vol. SE-3, pp. 49-59, Jan. 1977.

[28] M. W. Alford, "A requirements engineering methodology for real-time processing requirements," *IEEE Trans. Software Eng.*, vol. SE-3, pp. 60-69, Jan. 1977.

[29] D. T. Ross, "Structured analysis (SA): A language for communicating ideas," *IEEE Trans. Software Eng.*, vol. SE-3, pp. 16-34, Jan. 1977.
[30] M. Jackson, "The Jackson design methodology," *IEEE Tutorial on Software Design Techniques*, 2nd ed. New York: IEEE Press, 1977, pp. 219-234.
[31] M. Alford, "Towards formal foundations for the Michael Jackson design methodology," in *Proc. COMPSAC '79*, 1979.
[32] E. F. Codd, "A relational model for large shared data banks," *Commun. Ass. Comput. Mach.*, vol. 13, p. 377, 1970.
[33] —, "Further normalization of the data base relational model," in *Data Base Systems*, R. Rustin, Ed. Englewood Cliffs, NJ: Prentice-Hall, 1972, pp. 33-64.
[34] N. Wirth, "The programming language Pascal," *Acta Informatica*, vol. 1, pp. 34-63, 1971.
[35] N. Wirth and K. Jensen, *Pascal User Manual and Report.* New York: Springer-Verlag, 1974.
[36] A. V. Aho and J. D. Ullman, *Principles of Compiler Design.* Reading, MA: Addison-Wesley, 1979.
[37] D. D. Chamberlin, M. M. Astrahan, K. P. Eswaran, P. P. Griffiths, R. A. Lorie, J. W. Mehl, P. Reisner, and B. W. Wade, "SEQUEL2: A unified approach to data definition, manipulation, and control," *IBM J. Res. Develop.*, pp. 560-575, Nov. 1976.

Stephen S. Yau (S'60-M'61-SM'68-F'73), for a photograph and biography, see p. 435 of the July 1981 issue of this TRANSACTIONS.

Paul C. Grabow (S'79) received the B.A. degree in physics and mathematics from Luther College, Decorah, IA, in 1972, and the M.S. degree in computer science from Northwestern University, Evanston, IL, in 1980.

He is currently a Ph.D. candidate in computer science at Northwestern University. His interests include specification and design techniques with respect to maintainable software.

Mr. Grabow is a member of the Association for Computing Machinery, SIGPLAN, and SIGSOFT.

Collision-Free Access Control for Computer Communication Bus Networks

KAPALI P. ESWARAN, V. CARL HAMACHER, SENIOR MEMBER, IEEE, AND GERALD S. SHEDLER

Abstract—**This paper considers access control for local area computer communication networks. We propose two distributed access control schemes for a bus network. The schemes are simple and asynchronous, and provide for collision-free communication among ports. In addition, one of the schemes provides a bounded, guaranteed time to transmission for each port. We also show that this scheme is efficient in the use of the bus bandwidth, in the sense that there is only a small fraction of time during which the bus is idle when there is at least one packet available for transmission.**

Index Terms—**Access protocols, asynchronous distributed control, broadcast bus, collision avoidance, guaranteed time to transmission, local area networks.**

I. INTRODUCTION

IN LOCAL AREA computer communication networks [1]-[5], variable-length bit string messages called packets are exchanged among computers that typically are located hundreds of meters apart. In spite of the physical separation, the computers function cooperatively toward the goals of a single enterprise such as a plant site, a hospital, an office building, etc. The actual path that interconnects the machines is usually a ring or bus facility that provides high-speed, bit-serial communication over a coaxial cable or twisted pair of wires. This single physical path constitutes a shared resource that supports transmission of a single packet at any one time. There is a source and destination computer associated with each packet. At different times over the course of operation of the system, each computer may act as either a packet source or destination, and there is no presumed master computer that exerts any form of central control over use of the communication network. Each computer is attached to the communication facility through a specialized digital subsystem called a *port*, as shown schematically in Fig. 1 for the bus topology. A *tap* connects the port to the transmission bus.

This paper establishes properties of two access protocols developed by the authors [6] for local area bus networks. Control of access by ports to a shared bus is a nontrivial problem. This is because packets are generated at random times in the individual computers, and when two or more ports wish to transmit at the same time, some means is needed to ensure that exactly one of them proceeds. Schemes that do not restrict the number of simultaneous transmission attempts are available [5]. They rely on detection of collisions along with retry

Manuscript received January 12, 1981; revised May 8, 1981.
K. P. Eswaran and G. S. Shedler are with the IBM Research Laboratory, San Jose, CA 95193.
V. C. Hamacher is with the Departments of Electrical Engineering and Computer Science, University of Toronto, Toronto, Ont., Canada.