# Representing knowledge with functions and Boolean arrays

by K. Fordyce
J. Jantzen
G. A. Sullivan, Sr.
G. A. Sullivan, Jr.

**Over the past eighteen years a variety of advanced decision support systems have been built with knowledge-based expert system (KBES) components. For the past eight years, a knowledge representation and manipulation (KRM) scheme called FABA (Functions And Boolean Arrays) has been used. It has two basic principles. First, knowledge is viewed as a functional mapping between input and output variables, where the functions are expressed as *fact tables or bases* and *procedure modules*. Second, the function network can be represented with Boolean arrays. The basics of FABA, its implementation in APL2, and a simple example of FABA's application in a manufacturing dispatch application for IBM's semiconductor facility in Burlington, Vermont, are described in this paper.**

## 1. Introduction

The knowledge representation and manipulation (KRM) scheme based on functions and Boolean arrays (FABA) has been used successfully in a number of applications. of which the best known is the Logistics Management System (LMS) [1–3]. LMS is an advanced decision support system which serves as a dispatcher, monitoring and controlling the manufacturing flow of the IBM Burlington semiconductor facility. Appendix 1 provides an overview of LMS.

Other applications include Real-time Enrollment and Training (RET), Executive Information Network (EIN), Modeling Allowable Resources (MARS) [4], Real-time Control of Urban Drainage [5], and Expert Aided Adaptive Control [6]. (The last application is being done in LISP.)

The individual concepts in FABA have origins in a wide spectrum of programming techniques and "tool of thought notations" [7]. The use of functions as a base unit of organization of knowledge has roots in functional programming languages such as APL2 and LISP, and the mathematical concept of functions. Brown, Pakin, and Polivka [8] provide a general introduction to APL2.

The concept of organizing rules or knowledge modules in a tree structure or network is used in a variety of KRM schemes [9–13]. The application of Boolean arrays to store and manipulate networks has a long history in the APL community [7, 14–19]. The use of Boolean arrays and operations to efficiently handle logical processing is also well established [7, 20–22].

The use of tables to store knowledge has origins in general array theory [23], relational databases, APL2 [21, 24–28], and Prolog [29]. A general introduction to APL2 and knowledge representation with general arrays and tables can be found in [30].

The programming language concepts of avoiding data type dependencies, linking data and procedure to generate a "natural" object, and adaptability to change have origins in APL2 and object-oriented programming [31].

The origins of FABA are described in [22, 32–34]. A detailed description of different aspects of FABA can be found in [35, 36].

Others in the knowledge-based expert system field have independently developed approaches with some similarities to FABA; for example, see [37–40].

## 2. A brief review of mathematical functions

Functions are a description of the mapping between one set of independent (input) variables and a dependent (output) variable. For each set of input variable values, there is a mapping into only one output variable value. Different input variable value sets may map into the same output variable value. The set of possible input variable value sets is called the *domain*. The set of possible output variable values is called the *range*.

The reader is probably most familiar with functions where the mapping description is expressed as an algebraic equation, and the domain and range are numbers. For example,

$$W = X^2 + 4Y.$$

Sometimes input variables are linked to an output variable through more than one equation, requiring the establishment of an intermediate output variable. For example,

$$W = f(X,Y) = X^2 + 4Y, \tag{1}$$

$$Z = g(W,V) = 3W + 2V \tag{2}$$

is a system of two equations expressing a functional mapping of the output variable $Z$ from the input variables $V$, $X$, and $Y$. $Z$ has a direct dependency on $V$, but its dependency on $X$ and $Y$ is through the intermediate output variable $W$. Formally, this is called a composite function:

$$Z = h(X,Y,V) = g \circ f = g[f(X,Y),V], \tag{3}$$

where $h(X,Y,V)$ is the composite of the functions $f$ and $g$. To determine the $Z$ value for a set of input values, caculate $W$ from Equation (1), and then use this $W$ with $V$ in Equation (2) to calculate $Z$.

When the function mapping is written in algebra, we often carry out an algebraic simplification, e.g,

$$Z = h(X,Y,V) = g \circ f = g[f(X,Y),V], \tag{3}$$

$$Z = h(X,Y,V) = 3(X^2 + 4Y) + 2V, \tag{4a}$$

$$Z = h(X,Y,V) = 3X^2 + 12Y + 2V. \tag{4b}$$

In this case the intermediate output variable $W$ is eliminated, and we can calculate $Z$ directly.

## 3. Tables and procedure modules as functions

"*Tables*" or "*fact bases*" represent a tabular representation of a functional relation between input and output variables, where the domains and ranges are a finite set of elements. *PM*s are small procedure modules used to describe functional relationships which carry out standard conditional logic and computation on the input variables to generate the output variables. The linkages between functions represent composite function operations.

An example set of functions is shown below:

TABLE 1 (T1)

| CHIPTYPE | STAGE | -> SETUP |
|---|---|---|
| tiger | 1 | 3 |
| tiger | 2 | 2 |
| lion | 1 | 4 |
| lion | 2 | 4 |

TABLE 2 (T2)

| SETUP | -> SETUP_TIME |
|---|---|
| 2 | 20 |
| 3 | 20 |
| 4 | 50 |

TABLE 3 (T3)

| CHIPTYPE | SETUP | -> PROCESS_TIME |
|---|---|---|
| tiger | 2 | 60 |
| tiger | 3 | 50 |
| tiger | 4 | NA |
| lion | 2 | NA |
| lion | 3 | NA |
| lion | 4 | 60 |

PROCEDURE MODULE 1 (PM1)

```
∇PM1 [□]∇
    ∇
[0]    PM1
[1]   ⍝ THIS FUNCTION DETERMINES
[2]   ⍝ THE APPROPRIATE SETUP CONDITION
[3]   ⍝
[4]   ⍝ VALUE CALCULATED (OUTPUT VARIABLE):   SETUP_COND
[5]   ⍝
[6]   ⍝ POSSIBLE OPTIONS FOR OUTPUT VARIABLE: long
[7]   ⍝                                       short
[8]   ⍝
[9]   ⍝ VALUES USED      ( INPUT VARIABLE): SETUP_TIME
[10]  ⍝                                     PROCESS_TIME
[11]   SETUP_COND←⊂'LONG'
[12]  ⍝ SETUP_COND IS INITIALLY ASSIGNED THE VALUE LONG
[13]   CONDS←(SETUP_TIME<25),(SETUP_TIME<4×PROCESS_TIME)
[14]  ⍝ IF SETUP_TIME < 25
[15]  ⍝ THEN GENERATE A 1 (TRUE), ELSE 0 (FALSE)
[16]  ⍝ IF SETUP_TIME < (4 × PROCESS_TIME)
[17]  ⍝ THEN GENERATE A 1, ELSE 0
[18]  ⍝ ASSIGN THIS PAIR OF RESULTS TO COND
[19]   →(∧/CONDS)/L010
[20]  ⍝ IF BOTH CONDITIONS ARE TRUE BRANCH TO L010
[21]   →0
[22]  L010:
[23]   SETUP_COND←⊂'SHORT'
[24]  ⍝ SETUP_COND IS ASSIGNED THE VALUE SHORT
[25]   →0
```

Table 1 (T1) is the mapping of the input variables CHIPTYPE and STAGE into the output variable SETUP. Table 2 (T2) is the mapping of the input variable SETUP into the output variable SETUP_TIME. Table 3 (T3) is the mapping of the input variables CHIPTYPE and SETUP into the output variable PROCESS_TIME. Procedure Module 1 (PM1) is the mapping of the input variables SETUP_TIME and PROCESS_TIME into the output variable SETUP_COND.

These relationships can be written in the following functional notation:

SETUP = T1(CHIPTYPE,STAGE);

SETUP_TIME = T2(SETUP);

PROCESS_TIME = T3(CHIPTYPE,SETUP);

SETUP_COND = PM1(SETUP_TIME, PROCESS_TIME).

The concept of a composite function exists within our table-and-procedure-module method of describing functions.

For example, the functional relationship between the input variables CHIPTYPE and STAGE and the output variable SETUP_TIME can be found by using Tables 1 and 2 and viewing the variable SETUP as an intermediate output variable:

SETUP_TIME = TC1(CHIPTYPE,STAGE) = T1 ∘ T2.

The concept of "algebraic simplification" can sometimes be applied by generating a new table. For example, the composite function TC1 would result in the following table:

| TABLE COMPOSITE 1 (TC1) | | |
|---|---|---|
| CHIPTYPE | STAGE | -> SETUP_TIME |
| tiger | 1 | 20 |
| tiger | 2 | 20 |
| lion | 1 | 50 |
| lion | 2 | 50 |

In APL2 [8, 30], tables map directly into two-dimensional general arrays. Table 1 can be generated with the statement

```
T1 ← 4 3 ⍴ 'tiger' 1 3 'tiger' 2 2
  'lion' 1 4 'lion' 2 4
```

T1 is a matrix with a shape (⍴) of four rows and three columns. The matrix is filled in row by row (row major), as follows:

| | COL 1 | COL 2 | COL 3 |
|---|---|---|---|
| ROW 1 | tiger | 1 | 3 |
| ROW 2 | tiger | 2 | 2 |
| ROW 3 | lion | 1 | 4 |
| ROW 4 | lion | 2 | 4 |

APL2 provides indexing into any portion of the matrix, and a variety of comparison operations. The following statements will access column 1 and check whether any element in column 1 is equal to lion:

```
COL1 ← T1 [;1]
MATCH1 ← (⊂'lion') ≡¨ COL1
```

In the first statement, the variable COL1 is assigned (←) the values in column 1 ([;1]) of T1. COL1 is a vector with four elements. The second statement matches (≡) the character string or value *lion* against each (¨) element in the variable COL1. MATCH1 is a vector with four elements (0011), one for each member of COL1. An element of MATCH1 is a 1 if the corresponding element of COL1 has the value lion; else a 0.

Tables can also easily be represented and searched with simple character arrays (one character per element in an array) in APL2; Appendix 2 provides an example.

In APL2 a procedure module can be executed at any time with the execute primitive (⍎). The system functions ⎕*EA* and ⎕*EC* provide for the "protected" execution of a PM.

## 4. Generating the finite domain of a function

Assume that there are two input values to the function: CHIPTYPE and PRIORITY. The variable CHIPTYPE can be one of two values: tiger or lion. The variable PRIORITY can be one of three values: hot, warm, or cold. Then, each combination of values represents one point in the domain or state space [38]. In this case, there are 6 (2 × 3) possible unique combinations:

tiger hot,
tiger warm,
tiger cold,
lion hot,
lion warm,
lion cold.

The set of APL2 functions in Appendix 3 generates all elements in the domain without using recursion. The key APL2 primitive is encode (⊤).

Using a related process, we can identify duplicate entries in a table using the APL2 function shown in Appendix 4. The key APL2 primitives are decode (⊤) and *n*-wise reduction (2 = /).

## 5. Generating the network of functions

Using the following two Boolean arrays and some Boolean array operations in APL2, we can determine automatically how the different functions relate to one another [33–35] and automatically generate the dependency network.

The first item generated is a Boolean matrix called INMATIP (IP stands for INPUT). This matrix records which variables are input variables for which functions. INMATIP has one row for each variable and one column for each function (table or procedure module). A cell is assigned a 1 if the variable is in the "input portion" of a function; else a 0. For this example INMATIP would be

| | T1 | T2 | T3 | P1 |
|---|---|---|---|---|
| CHIPTYPE | 1 | 0 | 1 | 0 |
| STAGE | 1 | 0 | 0 | 0 |
| SETUP | 0 | 1 | 1 | 0 |
| SETUP_TIME | 0 | 0 | 0 | 1 |
| PROCESS_TIME | 0 | 0 | 0 | 1 |
| SETUP_COND | 0 | 0 | 0 | 0 |

The second item generated is a Boolean matrix called INMATOP (OP stands for OUTPUT). This matrix records which variables are output variables for which functions. INMATOP has one row for each variable and one column for each function (table or procedure module). A cell is assigned a 1 if the variable is in the "output portion" of a function; else a 0. For this example, INMATOP would be

| | T1 | T2 | T3 | P1 |
|---|---|---|---|---|
| CHIPTYPE | 0 | 0 | 0 | 0 |
| STAGE | 0 | 0 | 0 | 0 |
| SETUP | 1 | 0 | 0 | 0 |
| SETUP_TIME | 0 | 1 | 0 | 0 |
| PROCESS_TIME | 0 | 0 | 1 | 0 |
| SETUP_COND | 0 | 0 | 0 | 1 |

(These two arrays can easily be generated automatically by writing a small APL2 program to parse the description of the functions.) With the two arrays we can determine the variable SETUP_TIME as a function of the variables CHIPTYPE and STAGE through the intermediate variable SETUP and other second-order dependencies. The following APL2 expression gives this information:

```
INMATVAR ← INMATOP ∨.∧ ⍉ INMATIP

LEVEL_1_LINKS ← INMATVAR

LEVEL_2_LINKS
  ← LEVEL_1_LINKS ∨.∧ INMATVAR
```

INMATVAR is then

INPUT VARIABLES

| | | CHIP TYPE | STAGE | SETUP | SETUP _TIME | PROCESS _TIME | SETUP _COND |
|---|---|---|---|---|---|---|---|
| O V | CHIPTYPE | 0 | 0 | 0 | 0 | 0 | 0 |
| U A | STAGE | 0 | 0 | 0 | 0 | 0 | 0 |
| T R | SETUP | 1 | 1 | 0 | 0 | 0 | 0 |
| P I | SETUP_TIME | 0 | 0 | 1 | 0 | 0 | 0 |
| U A | PROCESS_TIME | 1 | 0 | 1 | 0 | 0 | 0 |
| T B | SETUP_COND | 0 | 0 | 0 | 1 | 1 | 0 |

INMATVAR contains one row and one column for each variable in the system. Rows reference output conditions; columns reference input conditions. INMATVAR shows "first-order" dependencies between variables. If a variable is a direct input variable (the columns) to an output variable (the rows), then that cell has the value 1. If not, the value is 0. For example, CHIPTYPE and STAGE are direct input variables to the variable SETUP through T1. Therefore, the cells (SETUP, CHIPTYPE — row 3, column 1) and (SETUP, STAGE — row 3, column 2) have the value 1.

The APL function ⍉ generates the transpose of a matrix. ⍉ INMATIP is

```
1 1 0 0 0 0
0 0 1 0 0 0
1 0 1 0 0 0
0 0 0 1 1 0
```

The APL2 operator dot ( . ) carries out the inner product operation, and the APL expression ∨ . ∧ is the primitive for Boolean matrix multiplication. Let us look in detail at how the value in the first row and first column of INMATVAR is calculated.

The first row of INMATOP is 0 0 0 0. The first column of ⍉ INMATIP is 1 0 1 0. We then "and" (∧) the corresponding elements, giving 0 0 0 0, and "or" across (∨/0 0 0 0) this result, giving 0.

As an alternative to using the APL expression ∨ . ∧, we could use the APL function ODA (Appendix 5).

LEVEL_1_LINKS is identical to INMATVAR. It provides the information about level 1 links or first-order dependencies between variables.

LEVEL_2_LINKS is

INPUT VARIABLES

| | | CHIP TYPE | STAGE | SETUP | SETUP _TIME | PROCESS _TIME | SETUP _COND |
|---|---|---|---|---|---|---|---|
| O V | CHIPTYPE | 0 | 0 | 0 | 0 | 0 | 0 |
| U A | STAGE | 0 | 0 | 0 | 0 | 0 | 0 |
| T R | SETUP | 0 | 0 | 0 | 0 | 0 | 0 |
| P I | SETUP_TIME | 1 | 1 | 0 | 0 | 0 | 0 |
| U A | PROCESS_TIME | 1 | 1 | 0 | 0 | 0 | 0 |
| T B | SETUP_COND | 1 | 0 | 1 | 0 | 0 | 0 |

LEVEL_2_LINKS contains one row and one column for each variable in the system. Rows reference an output condition; columns reference an input condition. LEVEL_2_LINKS shows second-order dependencies between variables. If a variable is a second-order input variable to an output variable, then that cell has the value 1. If not, the value is 0. For example, CHIPTYPE and STAGE are second-order input variables to the variable SETUP_TIME. SETUP_TIME depends directly on SETUP (T2). SETUP depends directly on CHIPTYPE and STAGE (T1). Therefore, SETUP_TIME has a second-order dependency on CHIPTYPE and STAGE (T2 to T1), where the *linking* variable is SETUP, and the cells (SETUP_TIME, CHIPTYPE — row 4, column 1) and (SETUP_TIME, STAGE — row 4, column 2) have the value 1.

The linking variables can be found using the following expression (the function LINKVAR is shown in Appendix 6):

```
LEVEL_1_LINK_VAR ← INMATVAR

LEVEL_2_LINK_VAR
  ← LEVEL_1_LINK_VAR LINKVAR INMATVAR
```

For first-order dependencies, the linking variable is the linked variable. Therefore, LEVEL_1_LINK_VAR is the same as INMATVAR.

LEVEL_2_LINK_VAR is

INPUT VARIABLES

| | | CHIP TYPE | STAGE | SETUP | SETUP _TIME | PROCESS _TIME | SETUP _COND |
|---|---|---|---|---|---|---|---|
| O V | CHIPTYPE | 0 | 0 | 0 | 0 | 0 | 0 |
| U A | STAGE | 0 | 0 | 0 | 0 | 0 | 0 |
| T R | SETUP | 0 | 0 | 0 | 0 | 0 | 0 |
| P I | SETUP_TIME | 3 | 3 | 0 | 0 | 0 | 0 |
| U A | PROCESS_TIME | 3 | 3 | 0 | 0 | 0 | 0 |
| T B | SETUP_COND | 5 | 0 | 4 5 | 0 | 0 | 0 |

LEVEL_2_LINK_VAR contains one row and one column for each variable in the system. Rows reference an output condition; columns reference an input condition. LEVEL_2_LINK_VAR shows the linking variable for the second-order dependencies between variables. The number 1 refers to CHIPTYPE, the number 2 refers to STAGE, etc. For example, this table shows us that the linking variable for the second-order dependency between the variable CHIPTYPE and the variable SETUP_TIME is the variable SETUP (number 3).

Third-order dependencies are found as follows:

```
LEVEL_3_LINKS
  ← LEVEL_2_LINKS ∨.∧ INMATVAR
```

LEVEL_3_LINKS is

```
0 0 0 0 0 0
0 0 0 0 0 0
0 0 0 0 0 0
0 0 0 0 0 0
0 0 0 0 0 0
1 1 0 0 0 0
```

Third-order linking variables are found by

```
LEVEL_3_LINK_VAR
  ← LEVEL_2_LINK_VAR LINKVAR INMATVAR
```

LEVEL_3_LINK_VAR is

```
0 0 0 0 0 0
0 0 0 0 0 0
0 0 0 0 0 0
0 0 0 0 0 0
0 0 0 0 0 0
3 3 0 0 0 0
```

The following APL2 statement finds fourth-order dependencies. (Notice that this matrix contains only 0s, which means that there are no more dependencies to find.)

```
LEVEL_4_LINKS
  ← LEVEL_3_LINKS ∨.∧ INMATVAR
```

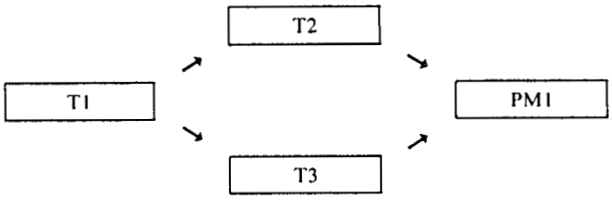
LEVEL_4_LINKS is

```
0 0 0 0 0 0
0 0 0 0 0 0
0 0 0 0 0 0
0 0 0 0 0 0
0 0 0 0 0 0
0 0 0 0 0 0
```

Using the information in these relationship matrices, we can, for example, determine the component variables that influence the value of the variable SETUP_TIME: SETUP_TIME has a direct dependency on the variable SETUP via function T2 and a second-order dependency on the variables CHIPTYPE and STAGE through the connecting variable SETUP via function T1.

The approach described above provides for simple, rapid, automatic generation of the dependency network (including "simultaneous" conditions; see Appendix 7) from just the knowledge of first-order dependencies.

Using a similar approach, we can build the following network for the functions of this example:




Using the same Boolean arrays and a slightly more complicated set of Boolean operations, we can automatically and quickly generate an ordering of the functions based on "relative independence" and thus "focus a knowledge network."

To explain "relative independence," let us look at an example. Given the equations

$$\text{VOLUME} = \text{AREA} \times \text{HEIGHT}, \quad \text{(a)}$$

$$\text{PERIMETER} = (2 \times \text{LENGTH}) + (2 \times \text{WIDTH}), \quad \text{(b)}$$

$$\text{AREA} = \text{LENGTH} \times \text{WIDTH}, \quad \text{(c)}$$

and

$$\text{HEATING COST} = 4 \times \text{VOLUME}, \quad \text{(d)}$$

we would need to execute Equation (a) before Equation (d), and Equation (c) before Equation (a). We could view Equations (b) and (c) as making up the most independent group or class of rules, since their input variables (LENGTH and WIDTH) are not calculated by any other equation. Equation (a) would be in the second group or class, since its input variables are either not calculated by another equation (HEIGHT) or calculated by an equation already ordered (AREA). Equation (d) would make up the third group.

For our example the functions are ordered as follows:

CLASS 1: T1
CLASS 2: T2 T3
CLASS 3: PM1

To explain "focusing," we return to Equations (a)–(d). If we have specified values for LENGTH, WIDTH, and HEIGHT, then the example equation set or network will calculate values for AREA, PERIMETER, VOLUME, and HEATING COST. If we tell the equation network to focus on the variable HEATING COST, it will drop Equation (b) and not calculate PERIMETER, since

PERIMETER has no link to HEATING COST. An automatic "focuser" is shown in Appendix 8.

The information provided from the automatic ordering of functions and network focusing is very useful in debugging knowledge sets, organizing them into logical groups, and identifying parallel content in inferencing.

## 6. Inferencing by forward chaining

In knowledge-based expert systems (KBES), an *inference engine* is used to determine which pieces of knowledge to use in what order to solve a problem or answer a question. Fordyce, Norden, and Sullivan [41] provide a detailed description of inference engines.

The task of a forward-chaining inference engine in a KBES is to monitor facts in a database, transaction stream, or interactive session with a user, and then determine which pieces of knowledge should be invoked in what order in response to a stimulus from the environment. Invoking or executing a piece of knowledge is called *firing.* In our example, the inference engine would monitor changes in values for variables CHIPTYPE, STAGE, etc., and then determine which functions to execute in what order.

The task of a backward-chaining or goal-driven inference engine is to monitor inquiries from a user as to whether a specified outcome, goal, or fact is true or false. In this case, the inference engine searches through the database and the knowledge base to verify or determine whether there is sufficient evidence to conclude that the specified goal is true. In our example, the user might ask "Does the variable PROCESS_TIME have a value of *short*?" or the more general question, "What is the value for PROCESS_TIME?" The inference engine would then search the existing functions and values for variables to determine the value for PROCESS_TIME.

In this section we describe how the network information described in the preceding section can be used efficiently to do inferencing by forward chaining. We will see that forward chaining is accomplished essentially by following the function network built from the Boolean arrays INMATIP and INMATOP. The backward-chaining procedure is shown in Appendix 9.

For this example we make use of the four-function knowledge base described in Section 3 and the Boolean arrays presented in Section 5. Assume that we initially know the following information:

| VARIABLE | VALUE |
|---|---|
| CHIPTYPE | tiger |
| STAGE | 2 |
| SETUP | unknown |
| SETUP_TIME | unknown |
| PROCESS_TIME | unknown |
| SETUP_COND | unknown |

From this information we generate the Boolean vector KUVEC (known–unknown vector). This tells us whether or not a variable has a value. There is one cell for each variable. A cell is assigned a 1 if the variable has a value; else a 0. For this example KUVEC is 1 1 0 0 0 0.

• *Cycle 1: Deciding which functions to fire*

First, the inference engine (IE) must decide which functions are candidates for executing or firing. A function is a candidate if all input variables for the function have values. POTEN is the variable that holds the list of legitimate candidates for firing. For this cycle POTEN is T1. The APL expression which generates POTEN is

```
POTEN ← ∧/ [1] KUVEC ≥ [1] INMATIP
```

where INMATIP is the matrix described in Section 2.

Second, the IE eliminates from POTEN those functions which have previously fired, and which it has no reason to fire again. (This is the first cycle, so this step can be skipped for this cycle.) If POTEN is empty at the end of this step, the inferencing process is finished.

Third, the IE decides which of the functions in POTEN should be fired in this cycle. This decision is often called *conflict resolution.* Our method is to identify the "most independent" function class represented in POTEN, and then keep in POTEN only the functions in that class. In this cycle there is only one function class represented in POTEN, so POTEN is now T1.

Fourth, the IE executes the functions that have been selected. The variable SETUP now has the value 2.

In this case, executing the function means searching Table 1 to determine whether there is a row which has the value tiger in the first column and the value 2 in the second column. If a match is found, the value in column 3 of the "matched" row is assigned to the variable SETUP. "Searching" through the tables to find a match is easy in APL2, as shown below.

Table 1 (T1) is a matrix with four rows and three columns:

```
tiger  1  3
tiger  2  2
lion   1  4
lion   2  4
```

We can access columns 1 and 2 as follows:

```
TABLE1[;1 2]

tiger  1
tiger  2
lion   1
lion   2
```

The variable $X$ can be assigned the known values for

CHIP_TYPE and STAGE,

```
X←'tiger' 2.
```

By the expression

```
SETUP ← (TABLE1[;1 2]∧.≡X) / TABLE1[;3]
```

The value for SETUP is then found to be 2.

If no match is found in Table 1, we still consider the function to have been executed, but we assign to the variable SETUP the value "no_value_found." (There are alternatives to this approach, but we do not discuss them in this paper.)

• *Cycle 1: Updating summary information*

Since the values of some of the variables have been altered by the function firings, the summary information is no longer current, and the IE must update each of these data structures.

First, it updates an item called FIRELOG. FIRELOG keeps track of which functions were candidates for firing and which functions were in fact fired at each cycle. FIRELOG is now

| CYCLE | FUNCTION | STATUS |
|---|---|---|
| 1 | T1 | 1 |

The first column shows the cycle, the second the function, and the third whether the function was fired (1) or not (0).

Second, the IE generates a vector called VCVEC (variable change vector). There is one position for each variable, in the same order as the rows for INMATIP. A cell is assigned a 1 if the variable was affected by the function firings; else a 0. A variable is affected if it is in the output portion of a function that has fired. The IE can get this information by using POTEN and INMATOP. From POTEN we know which functions were fired, and from INMATOP we know which variables are in the "then," or output, portion of each function. For this cycle VCVEC is 0 0 1 0 0 0:

| VARIABLE | CHANGE STATUS |
|---|---|
| CHIPTYPE | 0 |
| STAGE | 0 |
| SETUP | 1 |
| SETUP_TIME | 0 |
| PROCESS_TIME | 0 |
| SETUP_COND | 0 |

The APL2 expression which generates VCVEC is

```
∨/[2] INMATOP[;POTEN]
```

or

```
∨/[2] (⊂POTEN) ⌷[2] INMATOP.
```

Third, the IE updates KUVEC, which is now 1 1 1 0 0 0.

Fourth (and last), the IE keeps a history log of variables that have been affected. The change status for a variable at a given cycle is assigned a value of 1 if the variable was affected in the corresponding cycle; else a 0. The IE uses this information to determine whether a function that has been fired should be fired again. The information is stored in the matrix VCMAT (variable change matrix), which has one column for each cycle.

VCMAT is now

| VARIABLE | CYCLE 1 |
|---|---|
| CHIPTYPE | 0 |
| STAGE | 0 |
| SETUP | 1 |
| SETUP_TIME | 0 |
| PROCESS_TIME | 0 |
| SETUP_COND | 0 |

The IE has now finished updating its summary information and is ready to proceed to the next inference cycle.

• *Cycle 2: Deciding which functions to fire*

First, the IE determines candidates for firing. POTEN is initially T1, T2, and T3.

Second, the IE eliminates from POTEN those functions which have previously fired, and which it has no reason to fire again. POTEN is now T2 and T3.

Third, the IE decides which of the functions in POTEN should be fired in this cycle. Both T2 and T3 are in the same function order class, so POTEN is still T2 and T3.

Fourth, the IE executes the functions that have been selected. The variable SETUP_TIME now has the value 20. The variable PROCESS_TIME now has the value 60.

• *Cycle 2: Updating summary information*

First, the IE updates FIRELOG, which becomes

| CYCLE | FUNCTION | STATUS |
|---|---|---|
| 1 | T1 | 1 |
| 2 | T2 | 1 |
| 2 | T3 | 1 |

Second, the IE generates VCVEC, which becomes, for this cycle, 0 0 0 1 1 0:

| VARIABLE | CHANGE STATUS |
|---|---|
| CHIPTYPE | 0 |
| STAGE | 0 |
| SETUP | 0 |
| SETUP_TIME | 1 |
| PROCESS_TIME | 1 |
| SETUP_COND | 0 |

Third, the IE updates KUVEC, which becomes 1 1 1 1 1 0.

Fourth, the IE updates the history log of variables that have been affected. VCMAT is now

| VARIABLE | CYCLE 1 | CYCLE 2 |
|---|---|---|
| CHIPTYPE | 0 | 0 |
| STAGE | 0 | 0 |
| SETUP | 1 | 0 |
| SETUP_TIME | 0 | 1 |
| PROCESS_TIME | 0 | 1 |
| SETUP_COND | 0 | 0 |

The IE has now completed the second inference cycle and is ready to proceed to the third.

- *Cycle 3: Deciding which functions to fire*

First, the IE determines candidates for firing. POTEN is initially T1, T2, T3, and PM1.

Second, the IE eliminates from POTEN those functions which have previously fired, and which it has no reason to fire again. POTEN is now PM1.

Third, the IE decides which of the functions in POTEN should be fired in this cycle. POTEN is still PM1.

Fourth, the IE executes the functions that have been selected. The variable SETUP_COND now has the value *short*. In this case the APL2 function PM1 is run.

- *Cycle 3: Updating summary information*

First, the IE updates FIRELOG, which is now

| CYCLE | FUNCTION | STATUS |
|---|---|---|
| 1 | T1 | 1 |
| 2 | T2 | 1 |
| 2 | T3 | 1 |
| 3 | PM1 | 1 |

Second, the IE generates VCVEC; for this cycle VCVEC is 0 0 0 0 0 1:

| VARIABLE | CHANGE STATUS |
|---|---|
| CHIPTYPE | 0 |
| STAGE | 0 |
| SETUP | 0 |
| SETUP_TIME | 0 |
| PROCESS_TIME | 0 |
| SETUP_COND | 1 |

Third, the IE updates KUVEC, which becomes 1 1 1 1 1 1.

Fourth, the IE updates VCMAT, which becomes

| VARIABLE | CYCLE 1 | CYCLE 2 | CYCLE 3 |
|---|---|---|---|
| CHIPTYPE | 0 | 0 | 0 |
| STAGE | 0 | 0 | 0 |
| SETUP | 1 | 0 | 0 |
| SETUP_TIME | 0 | 1 | 0 |
| PROCESS_TIME | 0 | 1 | 0 |
| SETUP_COND | 0 | 0 | 1 |

The IE has now completed the third inference cycle, and is ready to proceed to the next.

- *All done*

The IE now determines that there is no reason to fire any more functions and ends processing. Given the new set of values for CHIP_TYPE and STAGE, the appropriate values for SETUP, SETUP_TIME, PROCESS_TIME, and SETUP_COND have all been determined by processing the knowledge in our four functions.

## 7. Integrating FABA and transactions

In the following example we illustrate how FABA is integrated with a manufacturing-lot transaction stream and tracking database. (Appendix 1 describes the manufacturing dispatch system.)

The manufacturing facility comprises lots to be processed, operations to be performed on the lots, and machines which carry out the operations. For each lot that is put into the manufacturing stream, the following information is recorded:

| LOT TRACKING DATA BASE | | | | | |
|---|---|---|---|---|---|
| LOT_ID | LOT_FAMILY | PRIORITY | DUE_DATE | OPR_FAMILY | MACH_FAMILY |
| 11129 | tiger | | | | |
| 11130 | tiger | | | | |
| 11132 | lion | | | | |

| ... LOT TRACKING DATA BASE | | | | | |
|---|---|---|---|---|---|
| LOT_ID | SETUP | SETUP_TIME | EST_LV | TIME | DELTA_SCHD |
| 11129 | | | | | |
| 11130 | | | | | |
| 11132 | | | | | |

When a lot is put into production, the first two variables (LOT_ID and LOT_FAMILY) in the record are given values. In this example there are three lots in the tracking database. (The field LOT_ID is listed twice for the convenience of the reader.)

The value for PRIORITY is a transaction received from production control (PC). An initial value between 1 and 100 is placed in PRIORITY when the lot is launched; this value can change once a day. We will give the name PRIORITY_PC to the variable holding the initial priority value transmitted from production control. The program module PM11 translates this value into one of four values, HOT, WARM, COLD, or ICE:

```
PM11

output variable: PRIORITY

input variable:  PRIORITY_PC

translate
PRIORITY_PC
(a value between 1 and 100) to
PRIORITY
(one of the following values:
HOT, WARM, COLD, or ICE)
```

The value for DUE_DATE is a transaction received from the order book (OB). An initial value is placed in DUE_DATE when the lot is launched; this value can change at any time. The value for DUE_DATE takes the form YY/DAY (last two digits of the year and Julian day; for example, 89/134). We call the variable holding the initial date transmitted from the order book DUE_DATE_OB. PM12 translates this value into the form MM/DD/YY (for example, 05/14/89):

```
PM12

output variable: DUE_DATE

input variable:  DUE_DATE_OB

translate
DUE_DATE_OB
(in the form YY/DAY) to
DUE_DATE
(in the form MM/DD/YY)
```

The value for the variable OPR_FAMILY (operation family) is changed in the record each time the lot enters a new manufacturing operation. This transaction is sent from the floor tracking system (FTS). The value for MACH_FAMILY (machine family) is based on OPR_FAMILY. Table 11 (T11) provides this value:

TABLE 11 (T11)

| OPR_FAMILY | -> MACH_FAMILY |
|---|---|
| bend | xxx |
| bake | yyy |
| test | zzz |

The value for the variable SETUP (required configuration for the machine to run this lot at this operation) is based on LOT_FAMILY and OPR_FAMILY. Table 12 (T12) provides this value:

TABLE 12 (T12)

| LOT_FAMILY | OPR_FAMILY | -> SETUP |
|---|---|---|
| tiger | bend | brown |
| lion | bend | blue |
| tiger | bake | yellow |
| lion | bake | red |
| tiger | test | green |
| lion | test | green |

The value for the variable SETUP_TIME (time to reconfigure this machine to the setup needed for this lot) is based on MACH_FAMILY. Table 13 (T13) provides this value:

TABLE 13 (T13)

| MACH_FAMILY | -> SETUP_TIME |
|---|---|
| xxx | 20 |
| yyy | 15 |
| zzz | 40 |

EST_LV is the estimated time until the lot leaves the operation where it is currently located. This value is obtained from PM13. PM13 makes use of information from manufacturing process specification variables such as raw process time and machine availability, the lot tracking database, and some decision rules to estimate when the lot will leave this operation. EST_LV is a run-time variable, and is recalculated whenever a query is made against this field. (Note that PM13 combines data and procedures on data into one natural unit. This is a common practice in APL2 and a key principle of object-oriented programming.)

PM13

output variable: EST_LV

input variable: LOT_ID
LOT_FAMILY
OPR_FAMILY
RUNTIME

function: determine an estimated leave time for this lot from this operation.

outside data used: raw process time
machine availability

DELTA_SCHD is an estimate of the number of days the lot is behind (minus to schedule) or ahead (plus to schedule) of schedule. This value is calculated by PM14 and is a time-dependent variable. It is recalculated on request from the user, or when either DUE_DATE or PRIORITY changes value, and otherwise every four hours. TIME stores the last time at which an update has been made to the DELTA_SCHED variable.

PM14

output variable: DELTA_SCHD

input variable: LOT_ID
LOT_FAMILY
PRIORITY
DUE_DATE
TIME

function: determine an estimated number of days the lot is behind or ahead of schedule.

outside data used: process flow data
machine availability

These relationships can be written in the following functional notation:

LOT_ID = input field when lot is launched.

LOT_FAMILY = input field when lot is launched.

PRIORITY = PM11 (PRIORITY_PC).

DUE_DATE = PM12 (DUE_DATE_OP).

OPR_FAMILY = input field when lot changes operation.

MACH_FAMILY = T11 (OPR_FAMILY).

SETUP = T12 (LOT_FAMILY, OPR_FAMILY).

SETUP_TIME = T13 (MACH_FAMILY)

= T13 ∘ T1 (OPR_FAMILY).

EST_LV = PM13 (LOT_ID, LOT_FAMILY, OPR_FAMILY, RUNTIME).

TIME = records last time an update is made to DELTA_SCHED.

DELTA_SCHD = PM14 (LOT_ID, LOT_FAMILY, PRIORITY, DUE_DATE, TIME).

INMATIP is

| | PM11 | PM12 | T11 | T12 | T13 | PM13 | PM14 |
|---|---|---|---|---|---|---|---|
| LOT_ID | 0 | 0 | 0 | 0 | 0 | 1 | 1 |
| LOT_FAMILY | 0 | 0 | 0 | 1 | 0 | 1 | 1 |
| PRIORITY_PC | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| PRIORITY | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| DUE_DATE_OB | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| DUE_DATE | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| OPR_FAMILY | 0 | 0 | 1 | 1 | 0 | 1 | 0 |
| MACH_FAMILY | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| SETUP | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| RUNTIME | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| SETUP_TIME | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| EST_LV | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| TIME | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| DELTA_SCHD | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

INMATOP is

| | PM11 | PM12 | T11 | T12 | T13 | PM13 | PM14 |
|---|---|---|---|---|---|---|---|
| LOT_ID | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| LOT_FAMILY | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| PRIORITY_PC | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| PRIORITY | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| DUE_DATE_OB | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| DUE_DATE | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| OPR_FAMILY | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| MACH_FAMILY | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| SETUP | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| SETUP_TIME | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| RUNTIME | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| EST_LV | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| TIME | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| DELTA_SCHD | 0 | 0 | 0 | 0 | 0 | 0 | 1 |

Let us look at an example. Assume that lot 11129 enters the operation "bake" and that a new due date or delivery date is set. The variable OPR_FAMILY for lot 11129 is assigned the value *bake*. The variable DUE_DATE_OP for lot 11129 is assigned the value 89/145. This triggers the following inference cycle:

- PM12 is fired, generating the value 05/25/89 for the variable DUE_DATE. T11 is fired, generating the value *yyy* for MACH_FAMILY. T12 is fired, generating the value *yellow* for SETUP.
- T13 is fired, generating the value 15 for SETUP_TIME. PM13 is fired, generating a value for EST_LV. PM14 is fired, generating a value for DELTA_SCHD.

## 8. Summary

In this paper we have described a knowledge representation and manipulation (KRM) scheme called FABA. This KRM scheme serves as the base for a multiple-goal-advocate approach to dispatch decision making [2, 3, 6].

FABA uses tables and procedure modules to store knowledge. We have found that these two methods of storing knowledge have some significant advantages over rules [36, 42]. For example, each of these items can be viewed as a function. First-order dependencies or links between functions are represented with two simple Boolean arrays. The information in these arrays can be obtained easily and automatically by writing a small APL2 program to parse the description of the functions. Working with these two arrays, some simple APL2 Boolean operations can quickly generate the entire function network. Deciding which functions to execute and when to execute them requires only some simple manipulations of the two Boolean arrays and a means for keeping track of when transactions occur. We have found this approach particularly effective in real-time, transaction-based, knowledge-based systems.

The FABA KRM scheme is clearly a product of the rich data structures provided by APL2. APL2 array data structures and associated operations permit the easy storage and manipulation of tables, and APL2 functions do the same for procedure modules. APL2 is clearly a notation for thought [7, 43] that facilitates a new view of problems and their solution in KRM [20].

## Appendix 1: Overview of LMS

The Logistics Management System (LMS) is a real-time, imbedded-transaction-based, integrated-decision and knowledge-based expert support system which serves as a dispatcher, monitoring and controlling the manufacturing flow of the IBM semiconductor facility near Burlington, Vermont. This facility develops and manufactures semiconductor memory and logic subsystems for current and future IBM products. Burlington produces some of the world's most complex

computer components, which are used throughout the IBM product line.

Dispatch scheduling decisions concerning monitoring and controlling of the actual manufacturing flow, or logistics. Decisions are made concerning trade-offs between running test lots for changes in an existing product or a new product, and running regular manufacturing lots, prioritizing late lots, and positioning preventive maintenance downtime, production of similar product to reduce setup time, assigning personnel to machines, covering for absences, and reestablishing steady production flow after a machine has been down.

LMS captures and stores in real time all manufacturing transactions, and maintains and provides access to knowledge bases and models. It permits the user and analyst to easily update knowledge bases and models as needed. LMS provides the dispatch decision makers with easy and flexible access to

1. Relational databases that contain the latest manufacturing transactions, such as the status of a machine, the location of a lot, the due date of a lot, or the availability of an operator.
2. Knowledge bases that contain such information as how to characterize a transaction (Is it a lot movement, a change in the status of a machine, or a change in an order?), how to characterize the lot type (Is it a test lot from the lab, a test lot from manufacturing engineering, an express lot for an important order, etc.?), the setup required for a lot, setup time, rework requirements, test requirements, alert conditions, product routing, throughput rates, preferred tools, operator training, operator schedules, average downtime for a machine, and how to calculate elapsed time. (Elapsed time would be defined as the calculation present time minus adjusted elapsed time; adjustments can be made for machine availability, second-shift work, a holiday, etc.).
3. Models that estimate how far ahead or behind schedule a lot is and the relative priority status of a lot, identify lots with the same setup requirements, establish global flow control levels (protective work in progress, recommended output from a work cell for the day, etc.) to guide production and avoid local optimization to the detriment of the global system, and assess the impact of machine dedication.
4. Heuristics to integrate the data, knowledge, and models to identify opportunities.

LMS provides dispatch decision makers with fast, flexible, and integrated access to this information. Support takes two forms: passive or decision-support, and intervention. In the decision-support mode, LMS passively waits for the user to make a request for information. In the intervention mode, LMS monitors the transaction stream and actively uses its knowledge bases and models to issue alerts and recommend what actions to take next.

*Example 1*

Assume that tester 1 is running lot type A and has a lot type B in its queue, and that tester 2 is running lot type B and has a lot type A in its queue. The testers are identical. There is an opportunity to eliminate two machine setups by exchanging the waiting lots, but an information system is needed to identify this opportunity because (1) the lot type is unknown until the box it is stored in is opened and the associated "paperwork" read, and (2) the two testers are not located close to each other.

*Example 2*

Assume that the stepper machine is almost finished processing a lot of type A. It has two lots of type C waiting to be processed. Both lots are ahead of schedule, but if they are not processed shortly, they will require recoating (a 15-minute process). The coating machine is almost finished with processing a lot of type A, and has five other A's waiting. All of the A's are behind schedule. The opportunity is to send the C's back to be recoated and keep the stepper set up to handle the A's.

*Example 3*

Assume that the tester is set up to process lot type B and has five more lots of that type waiting. But it also has an express lot of type C waiting. The laboratory is waiting for this lot for a critical development project. None of the type B lots are behind schedule. The retooling time is 20 minutes. The opportunity is to retool and test lot type C.

*Example 4*

Assume that the following sequence occurs for a photolithography machine:

1. Load a specific mask (contains the image of the circuit pattern required) on the machine.
2. Make a test run for low-tolerance lots.
3. Send the test run to be inspected, leaving the machine idle.
4. Wait for the inspection results.
5. Complete the low-tolerance lots.
6. Complete the high-tolerance lots that require the same mask, but require only a visual inspection at the machine location.

The production rate would improve if high-tolerance lots could be processed while the low-tolerance lots were being inspected, but to do this, it must be known which

lots in the queue have the same setup requirement, what tests are required for each lot, what the raw process time for each lot will be on this machine, and what the relative priorities are among all waiting lots.

LMS has helped IBM Burlington improve its throughput and ability to meet delivery schedules (serviceability). It is a critical component in running major areas of the manufacturing facility. The process operator on the manufacturing floor receives advice on scheduling. The manager is alerted to opportunities that have appeared or are about to emerge. The maintenance technician is given information on the impact of machines that are out of service, and is alerted to their conditions.

## Appendix 2: Handling tables with simple arrays

Following is an example of setting up and searching a table with simple character arrays:

```
T1 ← 4 10 ⍴ 'tiger 1 3 tiger 2 2 lion 1 4 lion 2 4 '
```

T1 is

```
tiger  1  3
tiger  2  2
lion   1  4
lion   2  4
```

T1 is a matrix with four rows and ten columns; each element T1 is a single EBCDIC character. The first five columns of T1 contain the values for CHIPTYPE. Columns 6 and 7 contain the values for STAGE, and Columns 8 and 9 contain the values for SETUP. Therefore, COL1 is obtained as follows:

```
COL1 ← T1[;⍳5]
```

The APL function iota (⍳) generates the values 1 2 3 4 5. COL1 is

```
tiger
tiger
lion
lion
```

The variable MATCH1 can be generated as follows:

```
MATCH1 ← COL1∧.= 'lion'
```

MATCH1 is 0 0 1 1.

Alternatively, MATCH1 can be calculated as

```
MATCH1 ← ∧/ [2] 'lion' = [2] COL1
```

## Appendix 3: APL functions for finding state space

Assume three input variables, V1, V2, and V3. The variable V1 can be one of three values: BILL, BOB, or JOE. The variable V2 can be one of two values: NYC or BOSTON. The variable V3 can be one of three values: USA, EUROPE, or ASIA. In this case, there are 18 ($3 \times 2 \times 3$) possible combinations. These variable values are stored in the variable VARLIST as a single character vector. The slash (/) is used as a delimiter between variables:

```
VARLIST←'BILL BOB JOE / NYC BOSTON
 / USA EUROPE ASIA/ '
```

The APL2 function SPACE generates the state space from the information provided in VARLIST:

SPACE VARLIST

| | | |
|---|---|---|
| BILL | NYC | USA |
| BILL | NYC | EUROPE |
| BILL | NYC | ASIA |
| BILL | BOSTON | USA |
| BILL | BOSTON | EUROPE |
| BILL | BOSTON | ASIA |
| BOB | NYC | USA |
| BOB | NYC | EUROPE |
| BOB | NYC | ASIA |
| BOB | BOSTON | USA |
| BOB | BOSTON | EUROPE |
| BOB | BOSTON | ASIA |
| JOE | NYC | USA |
| JOE | NYC | EUROPE |
| JOE | NYC | ASIA |
| JOE | BOSTON | USA |
| JOE | BOSTON | EUROPE |
| JOE | BOSTON | ASIA |

```
[0]   Z←SPACE X;DIVIDER;LIST;INDX;JK;JK1;JK2;JK3
[1]   ⍝
[2]   ⍝ THIS FN DETERMINES ALL POSSIBLE COMBINATIONS FOR
[3]   ⍝ SET OF VALUES FOR DIFFERENT VARIABLES.
[4]   ⍝
[5]   ⍝ X IS THE SET OF VALUES FOR EACH VARIABLE.
[6]   ⍝ AN EXAMPLE OF X IS:
[7]   ⍝ BILL JIM BOB / NYC BOSTON /
```

```
[8]   ⍝ VARIABLE 1 IS NAMES AND THE POSSIBLE VALUES ARE:
[9]   ⍝             BILL JIM BOB
[10]  ⍝ VARIABLE 2 IS CITIES AND THE POSSIBLE VALUES ARE:
[11]  ⍝             NYC BOSTON
[12]  ⍝ THE DIVIDER BETWEEN EACH VARIABLE IN THIS CASE IS /
[13]  ⍝ IN GENERAL IT IS THE LAST NON-BLANK CHARACTER IN X
[14]  ⍝
[15]  ⍝ Z WILL HAVE THE SOLUTION:
[16]  ⍝    BILL NYC
[17]  ⍝    BILL BOSTON
[18]  ⍝    JIM  NYC
[19]  ⍝    JIM  BOSTON
[20]  ⍝    BOB  NYC
[21]  ⍝    BOB  BOSTON
[22]  ⍝
[23]  ⍝ ↓ ↓ INITIALIZATION
[24]   X←DBL X
[25]   X←,(' '≠X)⊂X
[26]   DIVIDER←¯1↑X
[27]   LIST←⍳⍴X
[28]  ⍝
[29]   JK←(DIVIDER = ¨X)/LIST
[30]   JK1←¯1+|2-/0,JK
[31]  ⍝ JK1 IS THE NUMBER OF POSSIBLE VALUES FOR EACH VARIABLE
[32]  ⍝ FOLLOWING OUR EXAMPLE PROBLEM JK1 IS 3 2
[33]   INDX←COMBINATIONS JK1
[34]  ⍝ THIS FINDS ALL UNIQUE COMBINATIONS IN NUMERIC FORM
[35]  ⍝ FOR OUR EXAMPLE INDX IS:
[36]  ⍝      1 1
[37]  ⍝      1 2
[38]  ⍝      2 1
[39]  ⍝      2 2
[40]  ⍝      3 1
[41]  ⍝      3 2
[42]   JK2←+\0,¯1↓JK1
[43]  ⍝ THIS CALCULATES RELATIVE DISPLACEMENT FOR EACH VALUE
[44]  ⍝ FOR OUR EXAMPLE JK2 IS 0 3
[45]   INDX←JK2+[2]INDX
[46]  ⍝ THIS ADDS JK2[1] TO COL1 OF INDX ETC.
[47]  ⍝ IN OUR EXAMPLE INDX IS NOW
[48]  ⍝      1 4
[49]  ⍝      1 5
[50]  ⍝      2 4
[51]  ⍝      2 5
[52]  ⍝      3 4
[53]  ⍝      3 5
[54]   JK3←LIST~JK
[55]   X←X[JK3]
[56]  ⍝ X IS THE DIFFERENT VALUES FOR THE VARIABLES, WITHOUT DIVIDERS
[57]   Z←(⍴INDX)⍴X[,INDX]
[58]  ⍝ Z IS THE ANSWER
[0]    A←DBL B
[1]   ⍝ A IS THE CHARACTER VECTOR B WITH
[2]   ⍝ LEADING/TRAILING BLANKS REMOVED
```

```
[3]   ⍝ AND INTERMEDIATE BLANK SUBSTRINGS REPLACED BY SINGLE BLANKS
[4]    A←¯1↓1↓(~' '  ' ∈ B)/B←' ',B,' '
```

```
[0]    Z←COMBINATIONS X;JK
[1]   ⍝ THIS FIND ALL POSSIBLE COMBINATIONS
[2]   ⍝ WHERE X[1] IS THE NUMBER OF MEMBERS OF SET 1
[3]   ⍝       X[2] IS THE NUMBER OF MEMBERS OF SET 2
[4]   ⍝       ....
[5]   ⍝       X[N] IS THE NUMBER OF MEMBERS OF SET N
[6]   ⍝
[7]   ⍝ COL 1 OF Z REFERS TO THE SET CORRESPONDING TO X[1]
[8]   ⍝ COL 2 OF Z REFERS TO THE SET CORRESPONDING TO X[2]
[9]   ⍝            ETC.
[10]  ⍝
[11]  ⍝ THE APL2 PRIMITIVE ⊤ DOES BASE NUMBER SYSTEM WORK
[12]  ⍝
[13]   X←,X
[14]   JK←×/X
[15]   JK←0,⍳(¯1+JK)
[16]   Z←X⊤JK
[17]   Z←⍉Z
[18]   Z←Z+1
```

## Appendix 4: APL function for finding duplicate table entries

We can represent a table with an integer substituted for each variable value. A is such a representation of a table:

```
A IS

2  1  3
1  1  2
2  1  3
3  2  3
2  2  3
3  2  3
2  1  3
```

B is the number of different values for each variable:

```
B IS

3  2  3
```

The function DUPLICATE finds all duplicate entries. Running DUPLICATE with A and B, we obtain

```
B DUPLICATE A
2  1  3
2  1  3
3  2  3
```

In this example the table entry 2 1 3 appears twice and the table entry 3 2 3 appears once. The function DUPLICATE is listed below:

```
      ∇DUPLICATE[⎕]∇
[0]    Z←X DUPLICATE Y;ID;INDX;W
[1]   ⍝ THIS FINDS DUPLICATES OF RULES
[2]   ⍝ WHEN THE RULES ARE IN THE FORM
[3]   ⍝   1  1  2
[4]   ⍝   2  1  3
[5]   ⍝   .  .  .
[6]   ⍝   1  2  1
[7]   ⍝
[8]   ⍝ Y IS THE RULES
[9]   ⍝ X IS THE NUMBER OF STATES FOR EACH
      ⍝ VARIABLE
[10]   ID←X⊥¯1+⍉Y
[11]   INDX←⍋ID
[12]   Y←Y[INDX;]
[13]   ID←ID[INDX]
[14]   ID←2=/ID
[15]   ID←0,ID
[16]   Z←ID/[1]Y
```

## Appendix 5: APL function ODA

The function ODA is written

```
[0]    Z←X ODA Y;JK;JK1;N;I
[1]   ⍝ THIS FUNCTION IS AN ALTERNATIVE TO X ∨.∧ Y
[2]    Z←((1↑⍴X),(¯1↑⍴Y))⍴0
```

```
[3]   ⍝ INITIALIZE THE OUTCOME MATRIX
[4]   ⍝ THIS IS INITIALIZED TO ALL ZEROS (0).
[6]   ⍝ IT HAS THE SAME NUMBER OF ROWS AS X,
[7]   ⍝ AND THE SAME NUMBER OF COLUMNS AS Y
[8]    JK←⍳¯1↑⍴X
[9]   ⍝ THIS IS A LIST OF THE COLUMNS IN X
[10]  ⍝ IF X HAS SIX COLUMNS THEN THIS JK IS 1 2 3 4 5 6
[11]   N←1↑⍴Z
[12]  ⍝ NUMBER OF ROWS IN Z
[13]   I←1
[14]  ⍝ I IS THE CYCLE COUNTER
[15]  L10:
[16]  ⍝ START OF LOOP WHICH PRODUCES Z
[17]   JK1←X[I;]/JK
[18]   →(0=1↑⍴JK1)/L20
[19]   JK1←Y[,JK1;]
[20]   JK1←∨/[1]JK1
[21]   Z[I;]←JK1
[22]  L20:
[23]   →(N≥I←I+1)/L10
[24]  ⍝ CHECK IF COMPLETED EACH ROW OF X, IF NOT BRANCH TO L10
[25]  →0
```

In this function the program loops or cycles through the matrix $X$ one row at a time to build the outcome matrix ($Z$). We can examine how this works with

```
INMATVAR ← INMATOP ODA ⍉INMATIP
```

INMATVAR is initialized to be all zeros; i.e., we assume that a variable does not have a first-order dependency on another variable until it is proven otherwise.

In cycle one ($I = 1$), we read the first row of $X$ (`X[I;]`). This is the CHIP row; it has the value 0 0 0 0. If a cell has a 1, the variable CHIP is an outcome variable for the corresponding function. If not, the cell has a value 0. We thus wish to find the corresponding column numbers where that element of `X[I;]` is not 0. This is done by the APL2 expression

```
JK1←X[I;]/JK  ⍝ REMEMBER JK IS 1 2 3 4
              ⍝ THIS EXPRESSION PASSES
              ⍝ BACK INTO JK1 THOSE
              ⍝ VALUES OF JK WHERE THE
              ⍝ CORRESPONDING ELEMENT
              ⍝ OF X[I;] IS 1
```

In this cycle JK1 is a null vector. This tells us that the variable CHIP is not an outcome variable from any other function. It therefore has no first-order dependencies, and we can stop processing this row of $X$. The APL2 expression

```
→(0=1↑⍴JK1)/L20
```

tells us that JK1 is a null vector and branches to L20. At L20 we increment the cycle counter, check whether we have reached the end of $X$ and, if not, start the process over again for the next row in $X$.

In cycle 2 ($I = 2$), we examine the STAGE row of INMATOP. This is all zeros, so there are no first-order dependencies. In cycle 3 ($I = 3$), we examine the SETUP row of INMATOP, which has the value 0 1 0 0. The value for JK1 (corresponding column number) is 1. This tells us that the variable STAGE is an outcome variable for the function with an index of 1 (function T1). If function 1 has any input variables, SETUP has a first-order dependency on these variables.

The matrix ⍉INMATIP provides the following information: Each row in ⍉INMATIP corresponds to a function. If a variable is an input variable to that function, its cell has a value of 1; if not, it has a value of 0.

ODA then reads any row of ⍉INMATIP in which the variable STAGE is an output variable of the corresponding function. (In this case we are only interested in row 1.) This is done by the APL expression

```
JK1←Y[,JK1;]
```

JK1 has the value 1 1 0 0 0 0. This tells us that the variables CHIPTYPE and STAGE are input variables to function 1, and thus SETUP has a first-order dependency on them. Therefore, these cells in $Z$ are changed to 1s.

The APL2 expression

```
JK1←∨/[1]JK1
```

handles the situation in which a variable being

investigated is an output variable of more than one function. It essentially ensures that as long as a variable is an input variable to at least one of these functions, it is listed as a first-order dependency.

The APL2 expression

```
Z[1;]←JK1
```

replaces the row of $Z$ which corresponds to the row of $X$ being investigated with the new information on first-order dependencies.

## Appendix 6: APL function LINKVAR

The function LINKVAR is written

```
[0]    Z←X LINKVAR Y;JK;JK1;JK2;N;I
[1]    ⍝ THIS FN FINDS THE LINKING
       ⍝ VARIABLE FOR N-ORDER
       ⍝ DEPENDENCIES
[2]    Z←((1↑⍴X),(¯1↑⍴Y))⍴⊂1⍴0
[3]    JK←⍳¯1↑⍴X
[4]    N←1↑⍴Z
[5]    I←1
[6]    L10:
[7]    JK1←X[I;]/JK
[8]    →(0=1↑⍴JK1)/L20
```

```
[9]    JK2←Y[,JK1;]
[10]   JK2←JK1×[1]JK2
[11]   JK2←⊂[1]JK2
[12]   JK2←ELIMX0⍣¯1⊢JK2
[13]   Z[I;]←JK2
[14]   L20:
[15]   →(N≥I←I+1)/L10
[16]   →0
[0]    Z←ELIMX0 X;JK1
[1]    ⍝ THIS IS CALLED BY LINKVAR
[2]    X←,X
[3]    JK1←X≠0
[4]    ⍎(0=∨/JK1)/'JK1[1]←1'
[5]    Z←JK1/X
```

LINKVAR operates similarly to ODA, except that it keeps track of the linking variable and not the new dependency.

## Appendix 7: Simultaneous conditions

An example of a simultaneous set of equations would be

PROFIT = REVENUE − (EXPENSE + BONUS)

BONUS = .05 × PROFIT

In this case REVENUE and EXPENSE are known variables, and PROFIT and BONUS are unknown.

An example using function notation might be the following:

V1 = f(V2, V3)
V2 = g(V1, V3)
V3 = h(V1, V2)

As an example of such a condition in manufacturing, assume that the machine to which a lot is assigned depends on the current estimate of how far behind or ahead (delta) of schedule the lot is. The delta schedule estimate depends on the machine to which the lot is assigned.

## Appendix 8: APL functions used in focusing

First, we use the following APL2 function to generate a matrix showing all dependencies between functions (tables and procedure modules) in our knowledge base:

```
      ∇RULEALL[⎕]∇
[0]    RULELINKS←INMATIP RULEALL INMATOP;JK
[1]    ⍝ THIS FUNCTION DETERMINES ALL THE RULES
[2]    ⍝ THAT INFLUENCE ANY SPECIFIC RULE
[3]    RULELINKS←(⍉INMATIP∨INMATOP)∨.∧INMATOP
[4]    L10:
[5]    JK←RULELINKS
[6]    RULELINKS←RULELINKS∨(RULELINKS∨.∧RULELINKS)
[7]    →(~JK ≡ RULELINKS)/L10
[8]    →0
```

The result is

| | T1 | T2 | T3 | P1 |
|---|---|---|---|---|
| T1 | 1 | 0 | 0 | 0 |
| T2 | 1 | 1 | 0 | 0 |
| T3 | 1 | 0 | 1 | 0 |
| P1 | 1 | 1 | 1 | 1 |

In RULELINKS there is one row and one column for each function in the system. Rows reference an output condition; columns reference an input condition. RULELINKS shows all dependencies between functions. If a function is an input function to an output function, the corresponding cell has the value 1; if not, its value is 0. For example, T3 has SETUP as an input variable. T1 calculates SETUP; therefore, T1 is an input function to T3 and the cell (T3, T1) has a 1.

Next, we select the row of INMATOP which corresponds to the variable on which we want to focus,

and store it as the vector VF. For this example assume that we wish to focus on the variable PROCESS_TIME. Then VF is

```
VF←INMATOP[5;]
```

or 0 0 1 0. The following APL2 statement generates the list of functions which affect the variable PROCESS_TIME:

```
(∨/[1] VF ∧ [1] RULELINKS)
 /'T1' 'T2' 'T3' 'P1'
```

The result is T1 and T3.

## Appendix 9: Outline of backward chaining

Backward chaining is easy to implement. If, for example, we needed to determine a value for PROCESS_TIME, we would follow the following steps:

- Find a function for which PROCESS_TIME is an output variable. If there is none, end processing.
  - In this case the answer is TABLE 3.
- Determine the input variables for this function.
  - In this case the answer is CHIP_TYPE and SETUP.
- Determine whether these variables have values.
  - CHIP_TYPE has a value.
  - SETUP does not have a value.
- If they do have values, execute the function. If not, repeat the process on the variables without values. In this case, we need to find a value for SETUP.
  - Find a function where SETUP is an output variable.
    - In this case the answer is TABLE 1.
  - Determine the input variables for this function.
    - In this case the answer is CHIP_TYPE and STAGE.
  - Determine whether these variables have values.
    - CHIP_TYPE has a value.
    - STAGE has a value.
  - Execute the function TABLE 1.
    - SETUP now has the value 2.
  - Feed this new information to the calling step.
- Now that SETUP has a value, execute function TABLE 3.
  - PROCESS_TIME now has the value 60.

## References


1. E. Feigenbaum, P. McCorduck, and P. Nii, *The Rise of the Expert Company: How Visionary Companies Are Using Expert Systems to Make Huge Profits*, Times Books, New York, 1988.
2. K. Fordyce and G. Sullivan, "Logistics Management System: Implementing the Technology of Logistics with Knowledge Based Expert Systems," in *Innovative Applications of Artificial Intelligence*, H. Schorr and A. Rappaport, Eds., AAAI and MIT Press, Cambridge, MA, 1989, pp. 183–200.
3. K. Fordyce and G. Sullivan, "Logistics Management System: Implementing the Technology of Logistics with an Advanced Decision Support System," *Interfaces* **20** (1989), to be published.
4. K. Fordyce and G. Sullivan, "Looking at Worksheet Modeling with Expert System Eyes," in *Expert Systems for Business*, B. Silverman, Ed., Addison-Wesley Publishing Co., Reading, MA, 1987, pp. 246–285.
5. J. Jantzen and L. Amdisen, "Real-Time Control of Urban Drainage," working paper, Electric Power Engineering Dept., Technical University of Denmark, KD-2800, Lyngby, Denmark, 1989.
6. G. Sullivan, Jr., "Expert Aided Adaptive Control," working paper, Rensselaer Polytechnic Institute, Troy, NY, 1989.
7. K. Iverson, "1979 Turing Award Lecture: Notation as a Tool for Thought," *Commun. ACM* **23,** No. 8, 444–465 (1980).
8. J. Brown, S. Pakin, and R. Polivka, *APL2 at a Glance*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1988.
9. E. Charniak and D. McDermott, *Introduction to Artificial Intelligence*, Addison-Wesley Publishing Co., Reading, MA, 1985.
10. C. Forgy, "Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem," *Artificial Intelligence* **9,** No. 1, 17–38 (1982).
11. M. Genesereth and N. Nilson, *Logical Foundations of Artificial Intelligence*, Morgan-Kaufman, Los Altos, CA, 1987.
12. N. Nilson, *Principles of Artificial Intelligence*, Morgan-Kaufman, Los Altos, CA, 1981.
13. J. Sowa, *Conceptual Structure,* Addison-Wesley Publishing Co., Reading, MA, 1984.
14. F. Evans and P. Larsen, *Structural Design of Control Systems*, lecture notes, Electric Power Engineering Department, Technical University of Denmark, DK-2800, Lyngby, Denmark, 1983.
15. O. Franksen, P. Falster, and F. Evans, "Qualitative Aspects of Large Scale Systems," in *Lecture Notes in Control and Information Sciences, Vol. 17*, A. Balakrishnan and M. Thoma, Eds., Springer-Verlag, Heidelberg, 1979.
16. R. Tarjan, "Testing Flow Graph Reducibility," *J. Computer & Syst. Sci.* **9.** No. 3, 355–365 (December 1974).
17. K. Fordyce and G. Sullivan, "Equation Manipulating Expert Systems," Internal Technical Report, IBM Data Systems Division, Kingston, NY, 1982.
18. S. Lichtenthal, *APL2 Helping to Cope with Business Complexity: AIM—Advanced Interactive Modeling*, IBM Latin America, Mt. Pleasant, NY 10591, 1989, available through IBM branch offices.
19. P. Ravn, *A Structural Approach to Financial Planning Systems*, Ph.D. Thesis, Electric Power Engineering Department, Technical University of Denmark, DK-2800, Lyngby, Denmark, 1983.
20. J. Brown and M. Alfonseca, "Solutions to Logic Problems in APL2," APL87 Conference Proceedings, *APL Quote Quad* **17,** No. 4, 356–361 (1987).
21. E. Eusebi, "Inductive Reasoning from Relations," APL87 Conference Proceedings, *APL Quote Quad* **17,** No. 4, 386–390 (1987).
22. K. Fordyce and G. Sullivan, "Artificial Intelligence Development Aids," APL85 Conference Proceedings, *APL Quote Quad* **15,** No. 4, 106–113 (1985).
23. T. More, "A Theory of Arrays with Applications to Databases," *IBM Cambridge Scientific Center Report*, Order No. G320-2106, September 1975, available through IBM branch offices.
24. J. Brown, "A Generalization of APL," Ph.D. Dissertation, Dept. of Computer and Information Science, Syracuse University, Syracuse, NY, Clearing House 74h0004942 AD-770488./5, 1971.
25. James A. Brown, "A Development of APL2 Syntax," *IBM J. Res. Develop.* **29,** No. 1, 37–48 (1985).
26. M. Berry, "APL and the Search for Truth: A Set of Functions to Play New Eleusis," APL81 Conference Proceedings, *APL Quote Quad* **12,** No. 1, 47–53 (1981).
27. E. Eusebi and J. Brown, "APL2 and AI: A Study of Search," APL86 Conference Proceedings, *APL Quote Quad* **16,** No. 4, 295–300 (1987).
28. Manuel Alfonseca, "Frames, Semantic Networks, and Object-Oriented Programming in APL2," *IBM J. Res. Develop.* **33,** No. 5, 502–510 (1989).

29. W. Clocksin and C. Mellish, *Programming in Prolog*, 2nd ed., Springer-Verlag, New York, 1984.
30. J. Brown, E. Eusebi, K. Fordyce, and G. Sullivan, "APL and Expert Systems," *AIEXPERT* **2,** No. 7, 72–84 (1987).
31. B. Cox, *Object Oriented Programming*, Addison-Wesley Publishing Co., Reading, MA, 1987.
32. K. Fordyce and G. Sullivan, "A Boolean Array Based Algorithm in APL for Forward Chaining in Rule Based Production Expert Systems," *APL Quote Quad* **16,** No. 3, 5–12 (1986).
33. K. Fordyce and G. Sullivan, "Boolean Array Structures for a Rule Based Forward Chaining Inference Engine," APL87 Conference Proceedings, *APL Quote Quad* **17,** No. 4, 185–195 (1987).
34. J. Jantzen, "Inference Planning Using Digraphs and Boolean Arrays," APL89 Conference Proceedings, *APL Quote Quad* **19,** No. 4, 200–203 (1989).
35. K. Fordyce and G. Sullivan, "Boolean Array Based Inference Engines," Internal Technical Report, IBM Data Systems Division, Kingston, NY, 1989; to be published in *Proceedings of the APL and Expert Systems Conference*, Syracuse University, August 1989.
36. K. Fordyce and G. Sullivan, "Table Based Function Mappings and Knowledge Representation," Internal Technical Report, IBM Data Systems Division, Kingston, NY, 1989; to be published in *Proceedings of the APL and Expert Systems Conference*, Syracuse University, August 1989.
37. T. Rish, R. Reboh, P. Hart, and R. Duda, "A Functional Approach to Integrating Database and Expert Systems," *Commun. ACM* **31,** No. 12, 1424–1437 (1988).
38. J. Seagle and P. Duchessi, *Acquiring Expert Rules with the Aid of Decision Tables*, Monograph, Department of Management Science and Information Systems, School of Business, State University of New York, Albany, NY, 1989.
39. R. Braun, "Expert System Tools for Knowledge Analysis," *AIEXPERT* **4,** No. 10, 22–29 (1989).
40. G. Goldbogen, "Rules vs. Methods," Technical Note, Rensselaer Polytechnic Institute, Troy, NY, 1988.
41. K. Fordyce, P. Norden, and G. Sullivan, "Review of Expert Systems for the Management Science Practitioner," *Interfaces* **17,** No. 2, 64–77 (1987).
42. M. Alfonseca, "Procedural or Non-Procedural Languages: The Mixed Solution," *Cybernetics and Systems: The Way Ahead, Vol. 1*, J. Rose, Ed., Thales Publications, Lytham St. Annes, England, pp. 35–38.
43. C. Jansky, "Guest Editorial: A Tool for Thought," *ESD: The Electronic System Design Magazine*, pp. 10–11 (October 1989).

*Received July 31, 1989; accepted for publication November 30, 1989*

**Kenneth Fordyce** *IBM Data Systems Division, Engineering / Scientific Computing Center, Kingston, New York 12401.* Dr. Fordyce consults with IBM customers and internal IBM groups in the technical computing areas of knowledge-based expert systems, decision support, statistics, and operations research. He has published a number of papers in these areas. He is currently an advisor for *Expert System Applications* journal, and an area editor for *Production and Operations Management* journal. He has served as an adjunct faculty member at Marist College, and at the Albany and New Paltz campuses of the State University of New York. Dr. Fordyce's education includes an M.S. (1979) in operations research and statistics and a Ph.D. (1985) in administrative and engineering systems from Union College, Schenectady, New York.

**Jan Jantzen** *Technical University of Denmark, Electric Power Engineering Department, DK-2800 Lyngby, Denmark.* Since 1986 Dr. Jantzen has been a professor at the Technical University of Denmark, where he is currently teaching and doing research in fuzzy control and systems design using digraphs. He received both his M.S. in electrical engineering in 1979 and his Ph.D. in systems science in 1982 from the Technical University. Dr. Jantzen has held positions as visiting scientist at the IBM Thomas J. Watson Research Center (1986), computer consultant at SIM-CORP, Inc., Copenhagen (1984–1985), Queen's Quest visiting scholar at Queen's University, Kingston, Ontario, Canada (1982–1983), and systems designer at LK-NES, Inc., Copenhagen (1979–1982).

**Gerald A. Sullivan, Sr.** *IBM General Technology Division, Essex Junction, Vermont 05452.* Mr. Sullivan is a senior engineer and manager of IBM Burlington's Advanced Industrial Engineering Department. He directs a number of advanced decision-support system projects as part of the Burlington CFM/CIM strategy, and is an active participant in company-wide activities in this area. Additionally, he consults directly with IBM customers on issues in manufacturing and knowledge-based expert systems. In 1989 Mr. Sullivan received awards from the American Association of Artificial Intelligence (AAAI) and The Institute of Management Science (TIMS), recognizing his work in developing and implementing a manufacturing dispatch system called LMS. From 1977 through 1980 Mr. Sullivan was assigned to the IBM Americas/Far East Corporation, where he developed and implemented a series of executive-decision-support systems for the multilingual operations of A/FE. From 1973 through 1976, he developed and implemented a rules-based system for plans and controls in the IBM Field Engineering Division. From 1971 to 1973, he developed a system to link plants and education centers via electronic mail to keep planning databases stable. Mr. Sullivan received outstanding contribution awards from IBM for both these systems. He has authored a number of papers and made presentations at numerous conferences on topics in manufacturing decision support, and expert systems.

**Gerald A. Sullivan, Jr.** *Department of Mechanical Engineering, Aeronautical Engineering, and Mechanics, Rensselaer Polytechnic Institute (RPI), Troy, New York 12180.* Mr. Sullivan is a Ph.D. student in mechanical engineering at RPI. His dissertation topic, expert aided adaptive control, involves developing real-time knowledge-based systems (KBS) that provide two-way communication between the KBS and sensors and algorithms and use time as a key decision variable. Additionally, Mr. Sullivan is working for the NASA Lewis Laboratory on the development of Space Shuttle main-engine diagnostics. He received his B.S. in mechanical engineering in 1985 from the University of Vermont, where he also received outstanding achievement awards in physics and mechanical engineering and graduated summa cum laude. He received his M.S. in mechanical engineering in 1987 from RPI.