Prototype Software for Automatic Generation of On-line Control Programs for Discrete Manufacturing Processes

Gregg Ekberg and Bruce H. Krogh

CMU-RI-TR-87-3 (3)

Flexible Assembly Laboratory
The Robotics Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

February 1987

This work has been supported in part by General Motors Corporation, North American Philips Corporation, and the National Science Foundation under research grant DMC-8451493.

Table of Contents

List of Figures

Abstract

This report describes prototype software for automatically generating control programs for discrete manufacturing processes from a high-level description of the system control logic. The control logic is synthesized from a specification of the physical resource states required for each operation in the process. The software described in this report allows the user to specify interactively the operation sequencing logic and the actuators and sensors for each stage of the process. This information is then used to automatically generate code for on-line control computers. The current implementation supports binary sensor and actuator signals. The methodology is illustrated for the automatic generation of instruction list (IL) code to control a conveyor system in an existing robotic assembly plant.

1 Introduction

The writing and debugging of computer programs for sequential control accounts for a major component of the cost in implementing automated manufacturing systems. It is also time consuming and expensive to modify existing control programs. This report describes prototype software for reducing the time and cost involved in developing discrete control programs by automatically generating executable computer code from a high-level description of the system control logic. With this software the manufacturing engineer can specify the control logic in terms of the physical devices and operations from which the computer generates the programs for real-time control.

The prototype software described in this report is comprised of two programs: DBBUILD and PROGGEN. DBBUILD (Data Base BUILDer) is an interactive program used to build and modify a data base containing the system control description in terms of its physical devices and operations. PROGGEN (PROGram GENerator), executed from within DBBUILD, generates source code for the on-line control computer.

Normally, a skilled programmer performs the task of developing the controller program (usually in the Ladder Diagram Language) from the system designer's description of a discrete manufacturing system. Several problems can arise from the transfer of information to the programmer and the manual encoding of the system control logic. This is due to several factors, including:

- the designer's description of the system can be misinterpreted;
- the programmer's implementation may be inflexibly structured around the specific sensor/actuator realization, whereas the design engineer will maintain flexibility to meet changes in the operation of the system.
- the functional description of the system operation is not clearly reflected in the low-level control program.

These factors make it difficult to debug the control program or make changes in the sequencing of operations. Future modifications may be made difficult because the programmer did not anticipate possible changes in operation sequencing. The manufacturing engineer thinks more about how the sequencing of operations may affect future operating conditions.

The objective for developing the software described in this report is to eliminate the need for manually encoding the discrete control logic for manufacturing systems. This task is accomplished by the computer, allowing the system designer to specify and modify the control program using a high-level functional representation of the system. To maintain a systematic approach of generating system control programs, the code is generated for one operation at a time, using physical states of resources as enabling conditions. It is not necessary for the user to specify when to enable and disable the operation actuators; this task is performed automatically by PROGGEN.

Control of a discrete manufacturing system involves the coordination of multiple resources in a sequence of discrete operations. The initiation of each operation depends on the states of physical parts and devices (resources) within the system. A resource is any component within the manufacturing system that is involved in the system's operation: robots, fixtures, raw materials, controllers, etc. Following the execution of an operation, the states of the resources involved in the operation are changed; sensors are used to monitor changes the resource states.

We use Petri nets (PN) to model the discrete decision and control of a manufacturing system. Previous research has shown that PN models are effective for modeling the evolution of the state transitions in discrete systems [1]. PNs contain transitions, representing operations or events; places, representing conditions or states in the process; and directed arcs connecting the places and transitions. In the graphical representation of PNs, transitions are represented by vertical bars and places are represented by circles. The conditions enabling an operation are the resource states associated with the operations input transition. Upon completion of the operation the resources will be in the states associated within the output transition.

Recently, a systematic methodology was developed for synthesizing PN models of discrete manufacturing systems [2, 3, 4]. As presented by Beck [2], systematic approaches to developing the manufacturing system control logic can be synthesized from activity cycles for each resource. The resource activity cycles are developed, individually and then joined at common operations to synthesize the complete system control logic. We use this approach to define information that is entered into the database using DBBUILD.

The report is organized as follows. In section 2 we present an example of an automated conveyor system in an automobile paint shop which we use throughout the report to illustrate the functions of DBBUILD and PROGGEN. In section 3 we describe the structure and use of DBBUILD, and in section 4 we describe PROGGEN and discuss its performance in terms of the generated controller code. The performance criteria is based on correctness and gains or losses in efficiency compared to code developed manually by a programmer. In section 5 we propose methods for incorporating additional utilities such as timers, counters, and external functions into DBBUILD and PROGGEN. The structure of the database built by DBBUILD corresponds to a PN model of the system. Thus, PN techniques can be applied to determine if deadlocks or inconsistencies exist in the control logic. Current research into the application of PN theory for automatic evaluation and diagnosis of programming errors is discussed in the concluding section.

2 Control of an Automatic Conveyor

In this section we illustrate the Petri net methodology for an automatic conveyor system at the General Motors Truck & Bus Assembly Plant in Baltimore, MD. This example is used as an illustration throughout the remainder of the report. The conveyor system, illustrated in figure 1, indexes vans through a painting module consisting of a preparation booth, a base-coat booth, a clear-coat booth, and an observation booth. The preparation booth is used for final preparation of the vans before painting. Coats of pigment and resin are applied in the base-coat booth followed by the application of a coat of clear resin in the clear-coat booth. (All painting is performed by robots.) The purpose of the observation booth is to allow sufficient flash time so that the majority of the solvents can vaporize before the vans enter an oven for baking.

The conveyor system is presently controlled by an Allen-Bradley PLC-2/30. All sensor signals (from limit switches) and actuator commands (to pushers and mechanical stops) are binary. The controller coordinates the motion of the vans and the opening and closing of the doors between the booths. The doors must be closed during painting and a van must not be released into the next booth before the booth is availabel.

The conveyor chain, shown in figure 2, is a roller flight chain which allows a van to be held in place by mechanical stops while the chain, and other vans in the system, continue to move. Unpainted vans are held by a mechanical stop in the preparation booth and released when the base-coat booth becomes availabel. After entering the base-coat booth the van skid moves up to a set of grounding bars where the rear dog on the pusher catches the push plate on the skid (see figure 2). The van is then pushed into a secured painting position on the grounding bars. Prior to initiating the base-coat painting cycle the booth doors are closed and the pusher is retracted to prevent the buildup of paint on the cylinder shaft. Following the completion of the base-coat painting cycle, the doors are opened and the van skid is pushed off the grounding bars by the front dog of the pusher if the clear-coat booth is availabel. This sequence of events is repeated in the clear-coat booth. When the van moves into the observation booth, mechanical stops hold it in place while the solvents vaporize.

Using the PN methodology described in the introduction; a PN model of this system was synthesized from single resource activity cycles for the van, conveyor chain, mechanical stops in the preparation and observation booths, doors, and pushers in the base-coat and clear-coat booths. The base-booth portion of the PN for the conveyor control logic is shown in figure 3. Descriptions of the resource states and operations for this part of the net are given in appendix I. The PN for the clear coat and observation booths are similar.

3 DBBUILD

DBBUILD is an interactive program written in the C programming language and is used to enter the system description into a data base. The database is comprised of four major record types: 1) *operations*, containing information on input and output transitions, resource states, and actuators, 2) *resources*, containing information on the resource states and the sensor data required to define each state, 3) *sensors,* containing the address label of the sensor input port, and 4) *actuators,* containing the address label for the actuator output port. Diagrams of the four record types are shown in figures 4 through 7.

DBBUILD consists of procedures to create and modify these records. Each record is built using doubly linked lists established through pointers to structures. For example, and as shown in figure 4_f within the operation structure there are pointers to the next and previous operations, pointers to a list of the input transitions, pointers to a list of the output transitions, and pointers to a list of the associated actuators. In turn these structures have pointers to structures that contain information on the resource states and the actuators.

Attached to each each input and output transition of an operation are the resource states that are required to enable the transition. While building an operation the user; does not need to specify the sensors required to define the resource state. This information can be added at some other lime as a function of the resource state.

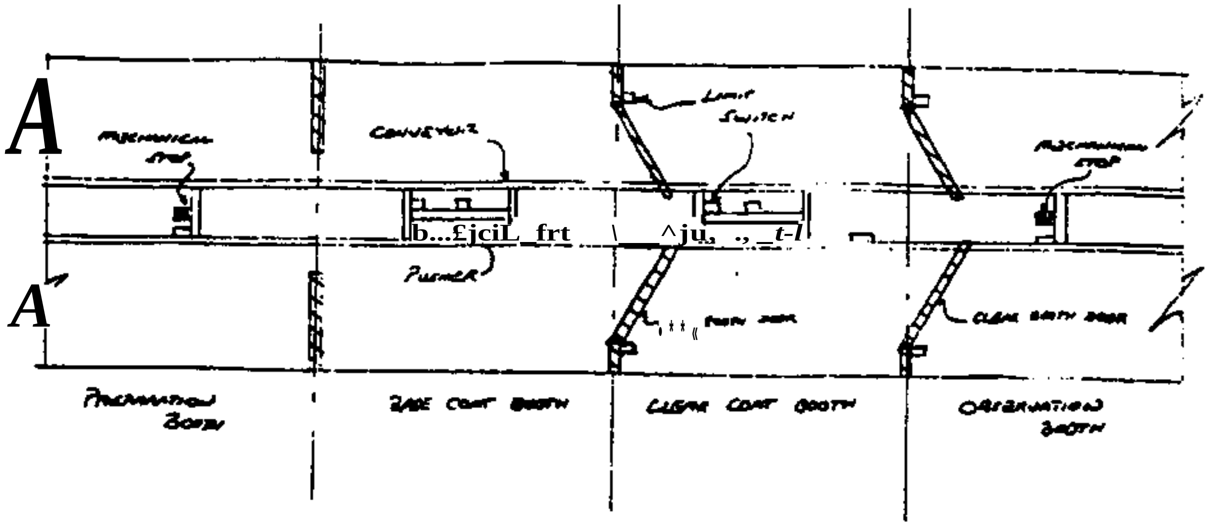

Figure 1: Modular paint shop conveyor system

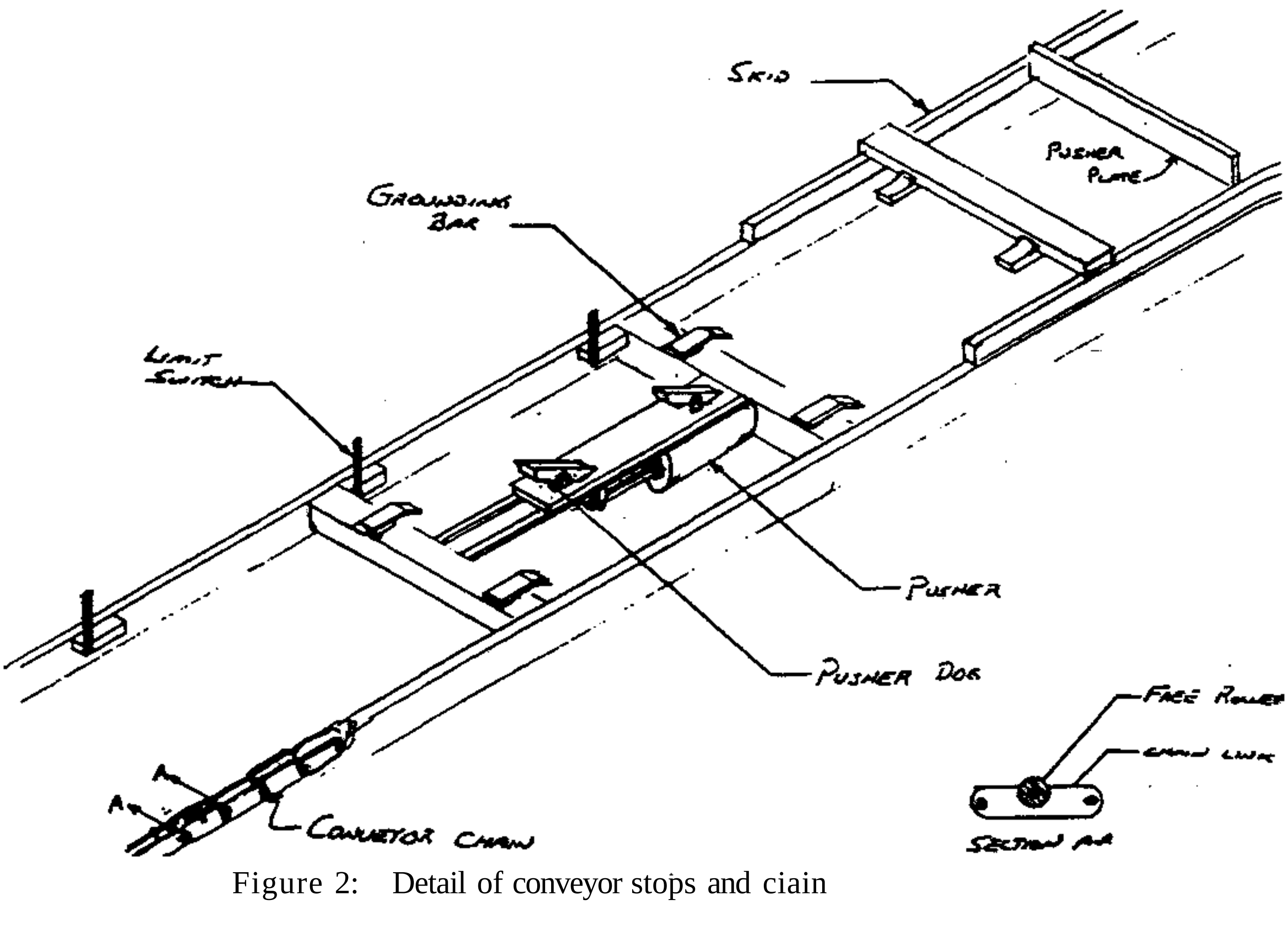

Figure 2: Detail of conveyor stops and ciain

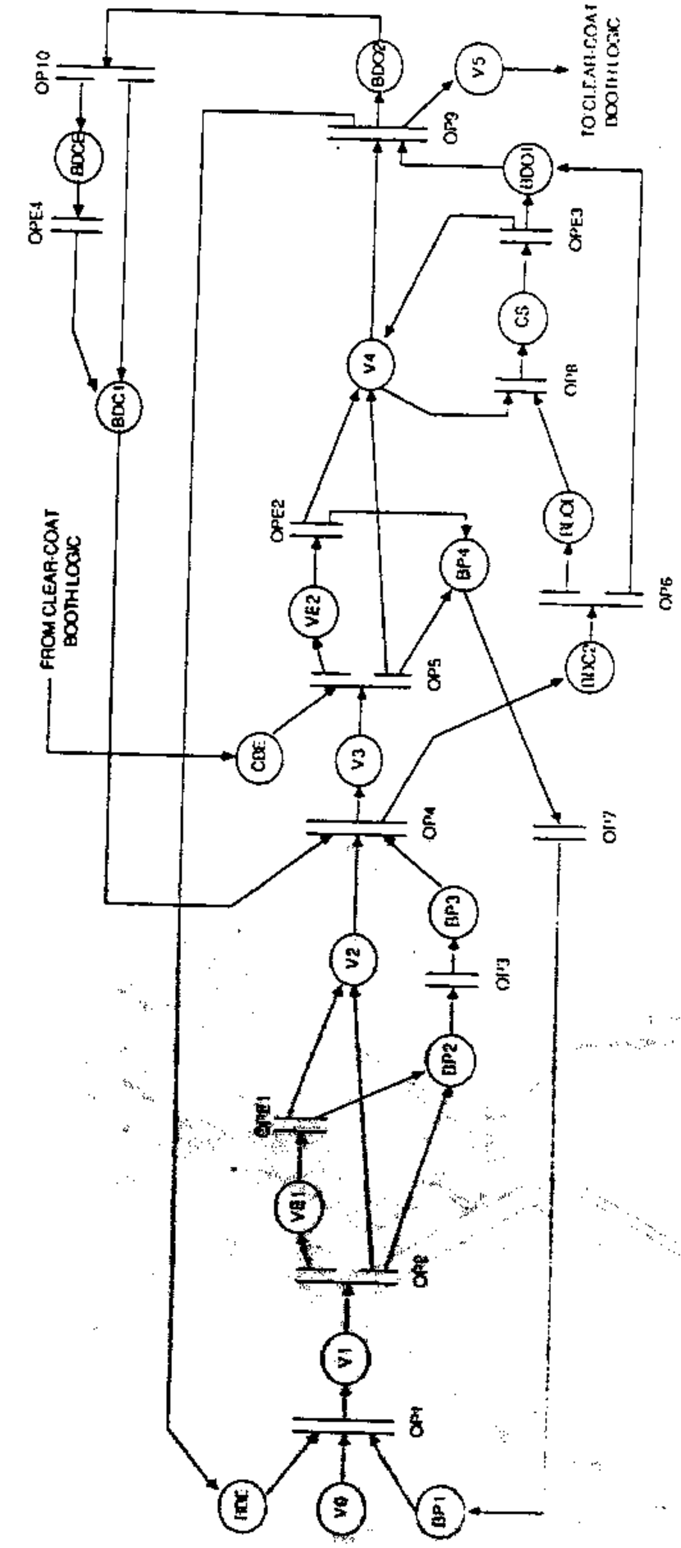

Figure 3: Pctri $_M$t model of conveyor control logic, for the base-coat booth

DBBUILD protects against entering incorrect conditions for identifying a resource state by accepting a sensor pointer only if the sensor has been entered in the data base. Similarly, an actuator cannot be referenced in an operation record unless it has been entered in the actuator database. Additionally, DBBUILD will inform the user if a state attached to an operation transition is, or is not, present in the resource data base. These checks help prevent confusion for the user and prevents errors from occurring in the controller code that is generated by PROGGEN. More information on DBBUILD is provided in the User's Manual in appendix II.

4 PROGGEN

4.1 Description

PROGGEN is written in the C programming language and is used to generate Instruction List (IL) code from a data base constructed using DBBUILD. Instruction List programs are executed sequentially and repeatedly by a programmable logic controller to generate and maintain the correct outputs to the system. The instructions used in this version of PROGGEN are per the International Electrotechnical Commission SC65A/WG6 Standard for Programmable Controllers [5]. The current version of PROGGEN supports the generation of a control program for a simple discrete process. It does not yet support operations requiring timers, counters, arithmetic functions, or logical comparison. Possible methods for incorporating these functions are described in section 5.

The basic logical flow of PROGGEN is shown in Figure 8. It looks at each operation separately, generating code to check the required resource states. Then, conditional on these states, code is generated to enable the desired actuator outputs. Setting (latching) the resultant resource states is based on the sensors associated with the resultant resource states, within a transition, and is performed to maintain the system state as defined in the Petri net.

The instructions within IL are used to develop conditional branches based on the system state. For example,

```
IF   [(limit switch 1 (LS1) is activated
AND  limit switch 2 (LS2) is not)
OR   (limit switch 1 is activated
AND  limit switch 3 (LS3) is activated)]   .
THEN turn on solenoid 1 (S1)
```

OPERATIONS

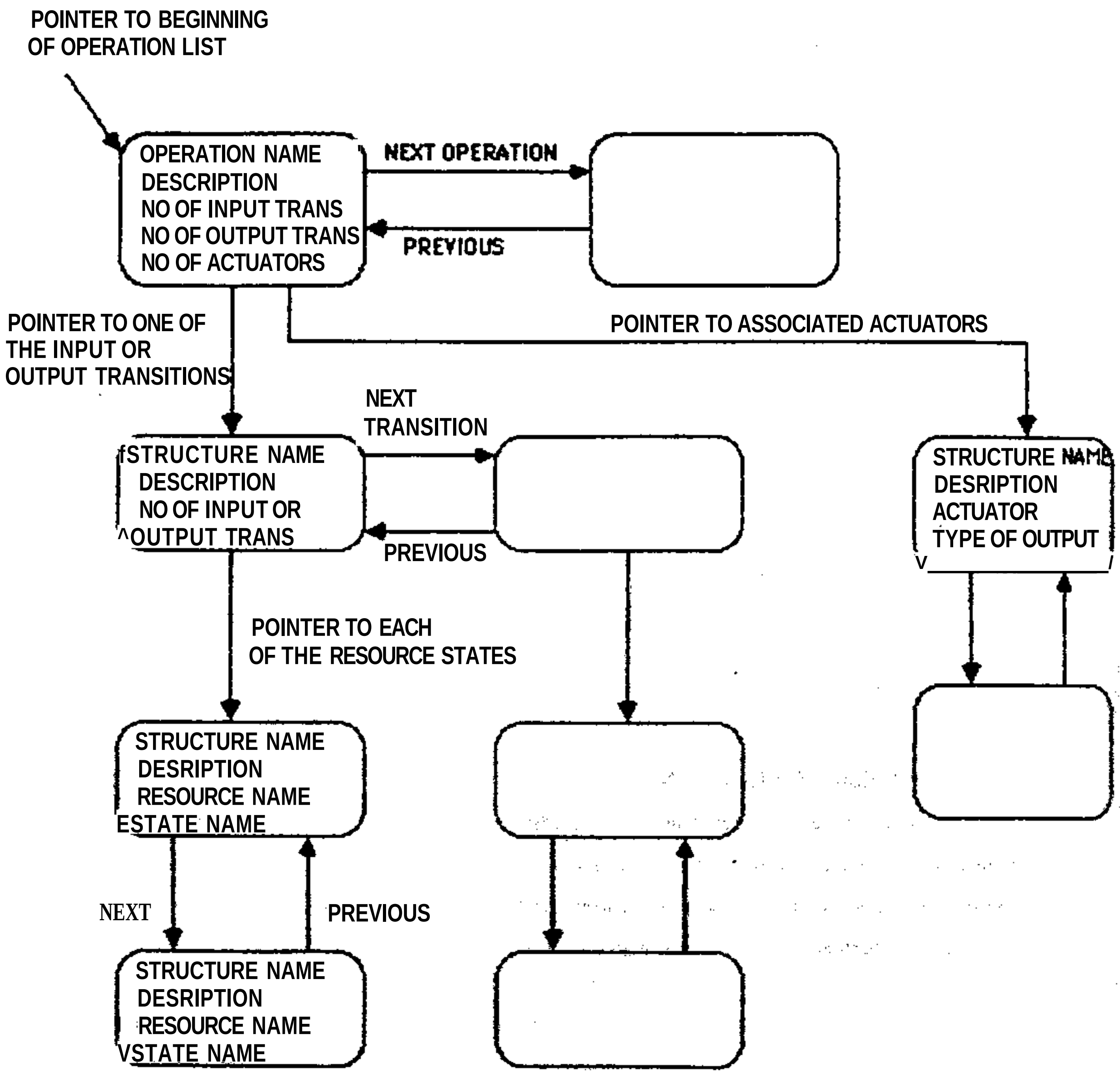

Figure 4: Database structures aad pointers: operation records

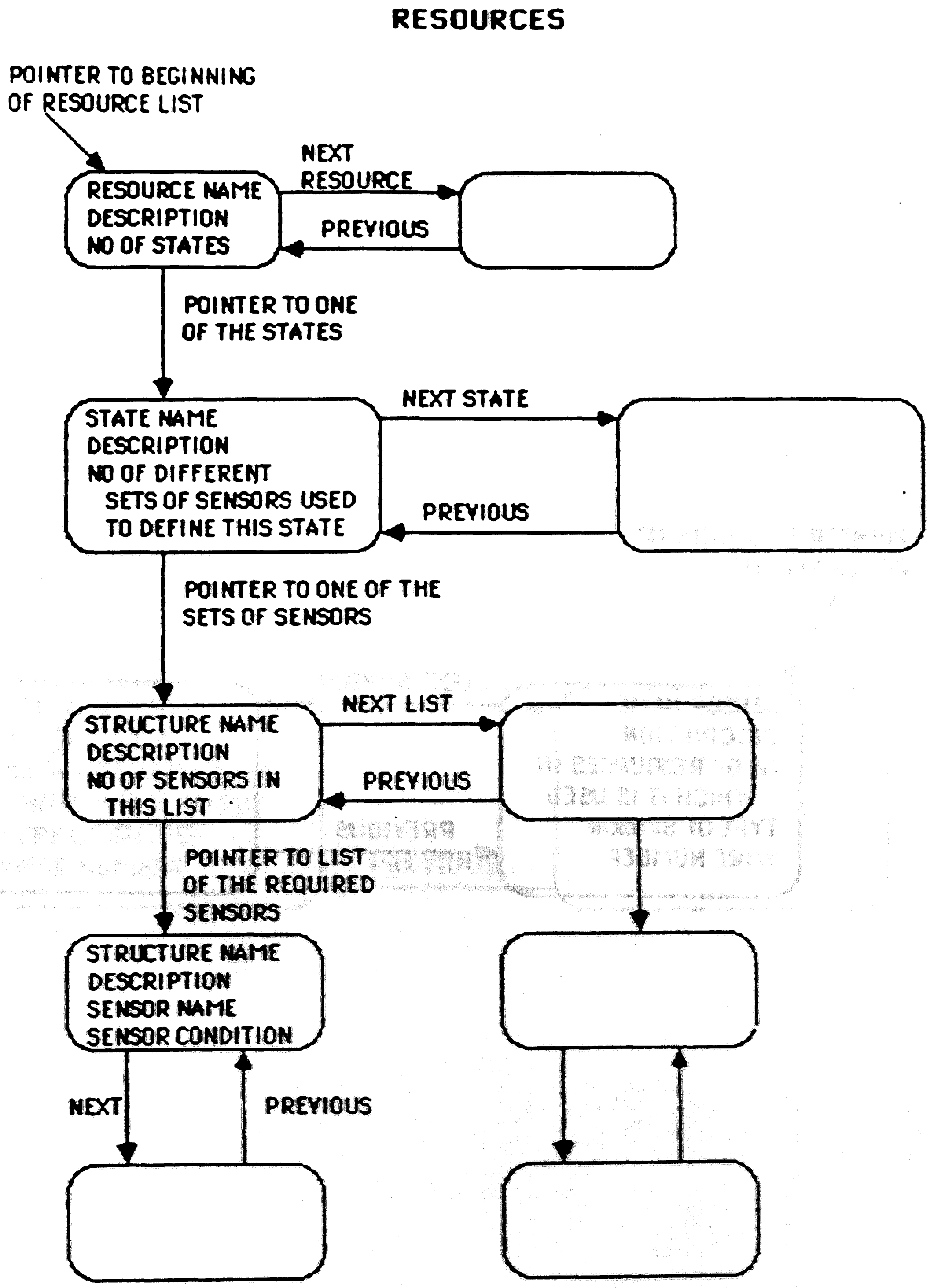

Figure 5: Database structures and pointers: resource records

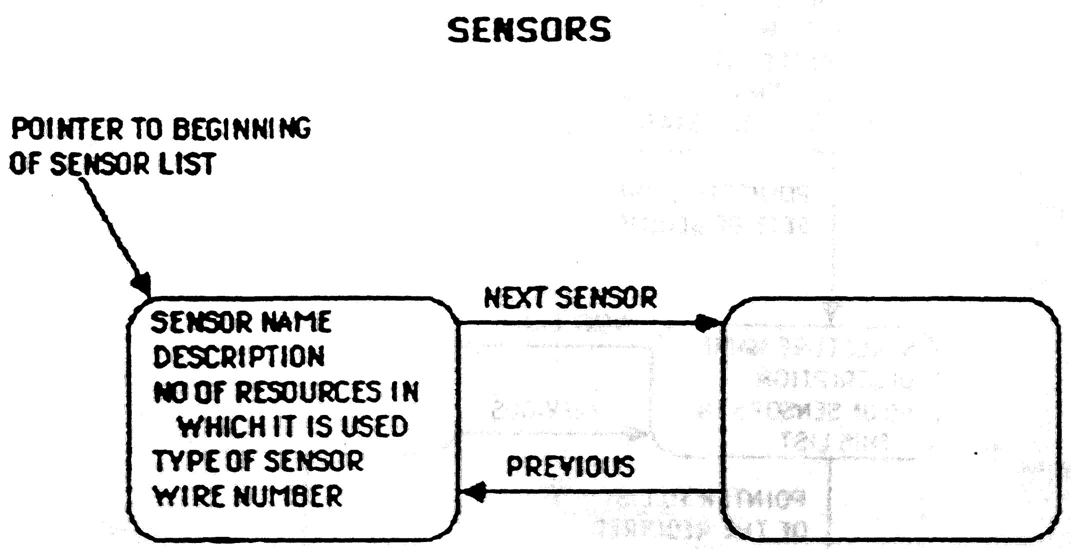

Figure 6: Database structures and pointers: sensor records

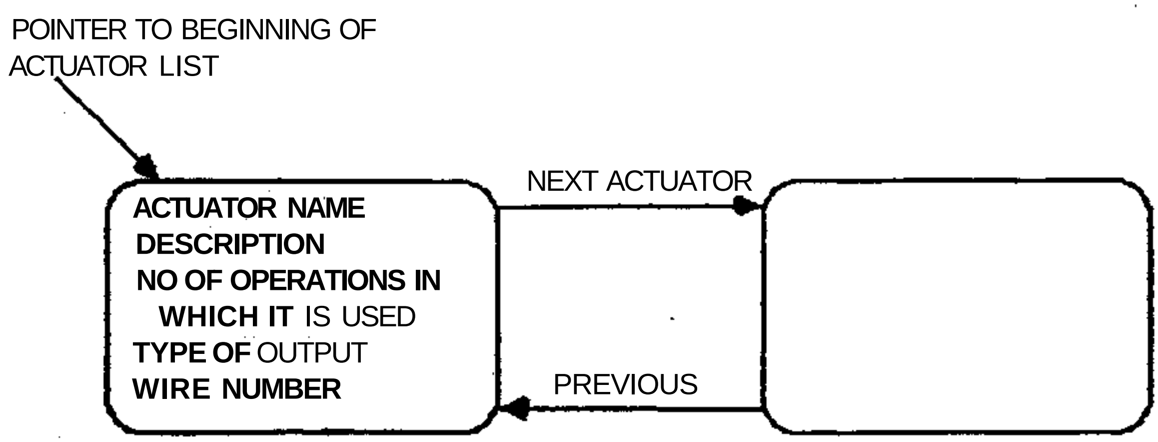

Figure 7: Database structures aad pointers: actuator records

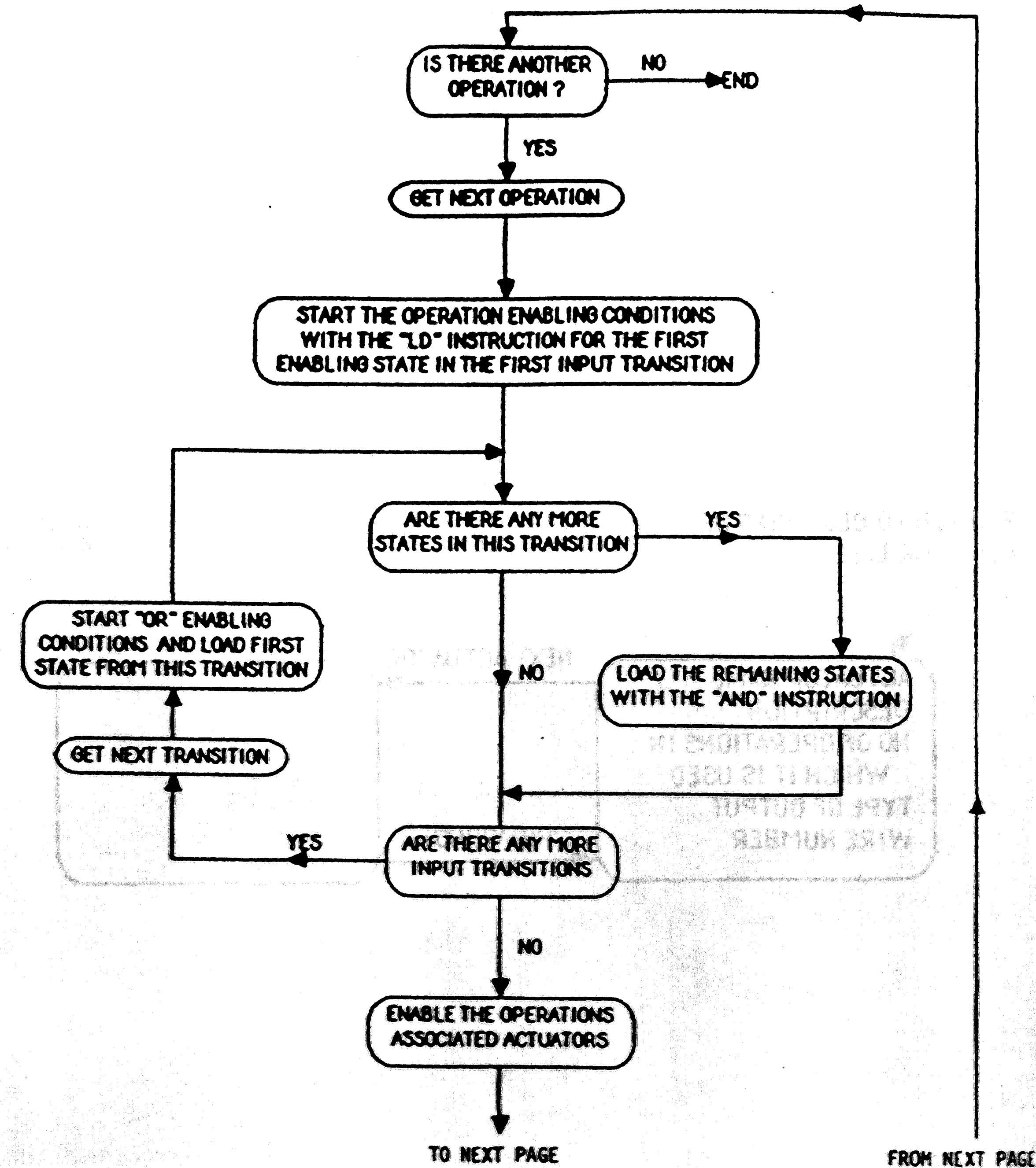

Figure 8: PROGGEN Flow Chart (Continued on next page)

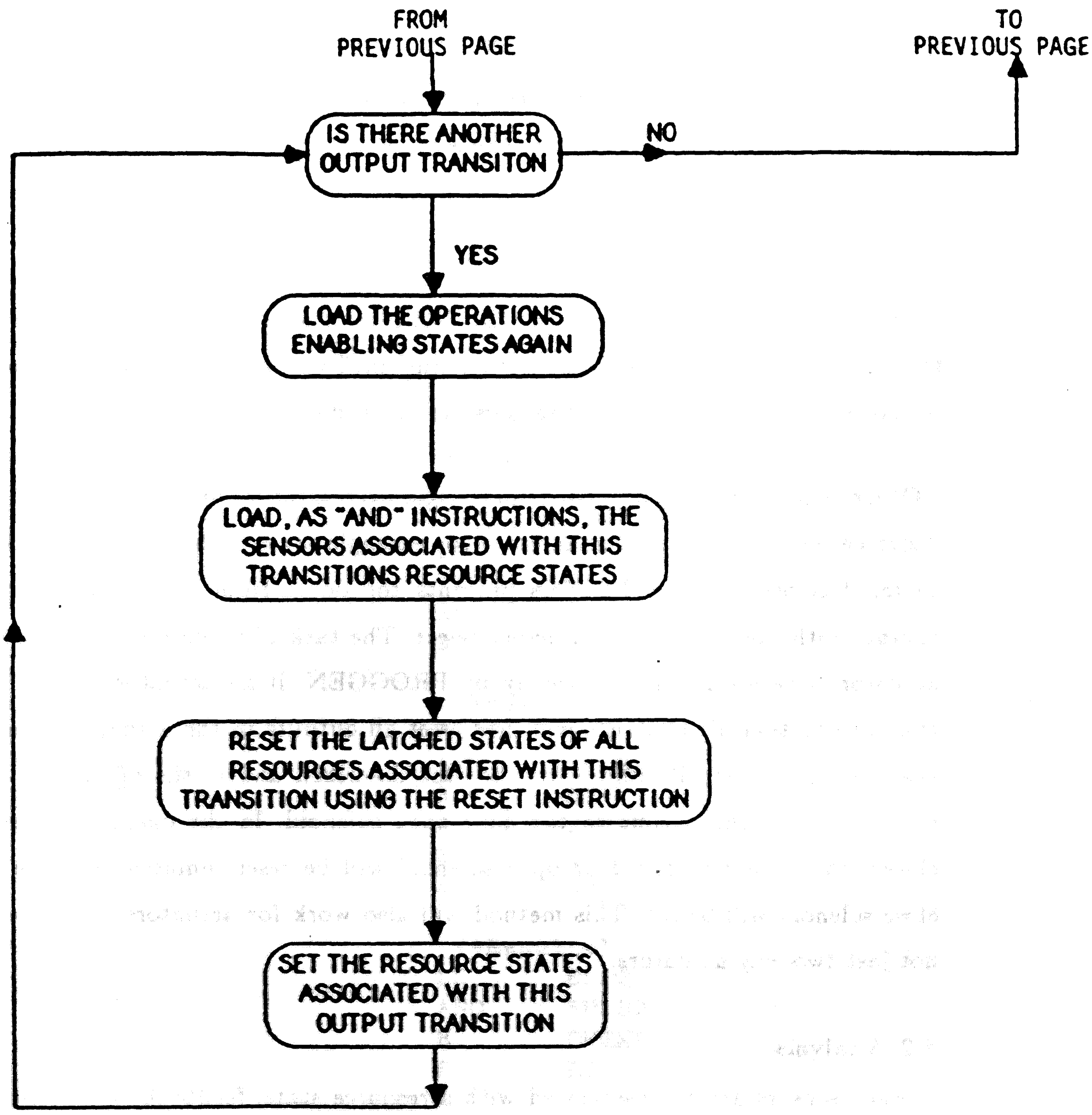

Figure FLOW (continued)

In IL would be represented as follows:

```
LD      LS1
ANDN    LS2
OR(     LS1
AND     LS3)
ST      S1
```

To simply enable the actuator when the input resource state conditions are satisfied is not sufficient. Actuators vary in types; some are required to remain enabled for the duration of the operation while others are required to remain enabled until another motion of the same actuator is needed.

Enabling the actuator output for the duration of an operation is established by the fact that the input states to the operation remain true until an output transition becomes true, as defined by the associated resource state sensors, and new states are defined.

Other types of actuators must remain rigid even after its motion is complete. For example, the doors between the booths in the conveyor example must be held open after the door open limit switch has been activated. This prevents the doors from drifting shut and possibly making contact with the van, causing a paint defect. The task of maintaining the output to the specified actuator is performed automatically by PROGGEN. If an actuator has in its description more than one motion, PROGGEN will first reset all outputs to the actuators then set the output for the desired motion. Therefore for the case described above, the operation that opens the door will set (latch) the output to the door open solenoid. In the operation that the door is to be closed, the output to the door open solenoid will be reset (unlatch) and the output to the door close solenoid will be set. This method will also work for actuators with more than one motion, not just two-way actuators.

4.2 Analysis

When sensors are not associated with a resource state, feedback words are needed to maintain the control logic. Feedback words are words that are stored in memory and are used to remember if a resource is in a given state. For example, the state of the base booth in the conveyor example is not explicitly defined by sensors. Therefore when its state is changed it is set with the "S" instruction (latched) and a location within its memory structure in DBBUILD is updated with its latched state. If an old state is still latched when a new state is to be latched, PROGGEN will unlatch the old state and latch the new state. This operation follows from the fact that a resource cannot be in more than one state at any given time.

Creating feedback words only for those states that are not defined by sensors does not provide sufficient information on the system state to enable the proper outputs. In the current version of PROGGEN, feedback words are created for *all* resource states. Storing all resource states provides the required information for proper sequencing, but leads to inefficient IL code.

To clarify the need for the storage of all resource state information, consider operations 2 and 5 in the conveyor example (move the van into the painting position and move the van out of the painting position). The resulting IL code for only remembering those states that are not defined by sensors is as follows: (Note: enabling conditions are now the sensors for those resource states that are defined by sensors:)

```
OPERATION 2

(*Enabling*)
LD      BLS1
AND     BPLS1
S       BPEXT

(*Res\ilti*)
LD      BLS2
AND     BPLS2
R       BPEXT

(*Result*)
LD      BLS1
AND     BPLS2
R       BPEXT
S       El
```

```
OPERATION 5

(•Enabling*)
LD      CBC
AND     BPLS1
AND     BLS2
S       BPEXT

(•Result*)
LD      BLS3
AND     BPLS2
R       BBF
R       BPEXT
S       BBC
S       CBF

(*Reisuit*)
LD      BLS2
AND     BPLS2
R       BPEXT
S       E2
```

We see that when both BLS2 and BPLS2 are high, following completion of operation 2_f E2 from operation S will be set, which is not what we wanled. To prevent this type of sequencing problem all resource states, whether defined by sensors or not? are used as feedback words. This change produces the correct code as shown below.

```
OPERATION  2

(*Enabling*)
LD      V1
AND     BP1
ST      BPEXT

(*Result*)
LD      V1
AND     BP1
AND     BPL52
AND     BLS1
R       V1
R       BP1
S       E1

(*Result*)
LD      V1
AND     BP1
AND     BLS2
AND     BPLS2
R       V1
S       V2
S       BP2
```

```
OPERATION  5

(*Enabling*)
LD      CBC
AND     BP3
AND     V3
ST      BPEXT

(*Result*)
LD      CBC
AND     BP3
AND     V3
AND     BPLS2
AND     BLS2
R       CBC
R       BP3
S       E2

(*Result*)
LD      CBC
AND     BP3
AND     V3
AND     BPLS2
R       CBC
R       BP3
S       B3C
S       V4
S       BP4
```

The inefficiency of using this method to maintain correct sequencing stems from the fact that many times feedback words are generated which are not required to maintain correctness. For example, the state V1 (van entered base booth) is explicitly defined by BLS1. At no other time is BLS1 activated, nor will the state V1 exist if BLS1 is not activated.

Using the S (set) instruction is considered poor programming style primarily because if a power failure occurs the set or latched states will remain high, thus resetting the system logic becomes very difficult. Also, with set instructions there is possibility of logic errors by forgetting to reset the word; however, PROGGEN removes this problem because it maintains the states of the latched words.

5 Additional Utilities

The prototype versions of DBBUILD and PROGGEN presented in this report have been developed to support automatic generation of controller code for systems with binary sensors and actuators. Further work is required to implement the required software to support timers, counters, external functions (add, subtract, logical comparison, etc.), and non-binary inputs and outputs. Some ideas for possible implementations of these control structures are presented in this section.

5.1 TIMERS

Timers are often used to monitor the sequencing of a system. A timer can be viewed as a function within an operation that is initiated when the operation is enabled. We propose to have operations that can be specified as timed operations for which DBBUILD will prompt the user for the pre-set timer duration. During controller code compilation PROGGEN will allocate a timer to that operation internally and will attach to the variable state TIMER the address of the timer completed status word (bit 15 of the timer address [5]). The use of the variable TIMER allows the user to specify those output transitions that are dependent on the timer. If the operation reaches an acceptable output transition the timer is automatically reset.

5.2 COUNTERS

Counters are often required to remember how many times an operation has been executed and based on the accumulated value of the counter, initiate another operation. For example, in an automated paint shop the paint gun requires cleaning if the same color has been used N times (If a different color is used a purge operation is performed which includes cleaning the gun). We therefore want to count the number of consecutive times the same color has been used. It is proposed to view the counter as a type of actuator. The counter name would act as the label to the counter address within the controller code. The state of the counter is then defined by two associated feedback words representing counting and finished states. These states can be defined by the counter address bits 16 and 15 respectively [5]. To allow the user to use the counter feedback words in other operations we define feedback words **label.ent** and label.done as follows:

```
for countervails < M    label.cnt = 1; label.done = 0
for countervalue = M    label.cnt = 0; label,done = 1
for countervalue > K    reset cou&tervalu*; countervalue = 1;
```

where label is the counter name as defined by the system designer. For example, samecolor.cnt would be the variable attached to bit 16 of the samecolor counter.

5.3 EXTERNAL FUNCTIONS

External functions are required to perform a series of operations that do not belong at the level of the system state description. For example, comparing the value of a sensor to some set point. It is proposed to have the user define an external function label in the associated actuator list in an operation and it will remain his responsibility to generate code for that label. Simple routines are easy to write in the Structured Text Language [5] and are easily accessible by the Instruction List code using the JMP instruction. All variables will be the same names as those used in the system description level.

6 Conclusion

This report presents some initial work in the area of automatic programming of programmable controllers from high level descriptions. The software developed illustrates the ability to interpret a data base that contains the system operation information, and from it generate executable controller code.

Additional work is required in the area of simulation and analysis of the generated control logic. The data base generated by DBBUILD is structured identically to the informatioE contained within a PN model of the system. This structure allows existing Petri net theories to be used to determine if deadlocks are present. The program that performs the net analysis may be a simulation program that can simulate the nets operation given an initial marking, or placing of the tokens.

Ultimately to allow the generated code to be used in a production environment, an interface such as Ladder Diagram needs to be presented to the technician for use in on-line debugging of the system. One of the purposes of the DEC Language Specification is to provide consistency between controller codes. This consistency should allow the development of linking programs that can change the controller code from IL to Structured Function Chart [5] to executable code, etc, and back again.

I. Sensors, Actuators, Resources, and Operations for Conveyor Example

The following two lists show the sensors and actuators used in the conveyor example:

SENSORS:

PLS1	PREP BOOTH LIMIT SWITCH 1
BLS1	BASE BOOTH LIMIT SWITCH 1
BLS2	BASE BOOTH LIMIT SWITCH 2
BLS3	BASE BOOTH LIMIT SWITCH 3
CLS1	CLEAR BOOTH LIMIT SWITCH 1
BPLS1	BASE PUSHER LIMIT SWITCH 1
BPLS2	BASE PUSHER LIMIT SWITCH 2
BLDO	BASE LEFT DOOR OPEN LIMIT SWITCH
BRDO	BASE RIGHT DOOR OPEN LIMIT SWITCH
BLDC	BASE LEFT DOOR CLOSED LIMIT SWITCH
BRDC	BASE RIGHT DOOR CLOSED LIMIT SWITCH

ACTUATORS:

PBSD	PREP BOOTH STOP DOWN
PBSU	PREP BOOTH STOP UP
BPEX	BASE PUSHER EXTEND
BPRET	BASE PUSHER RETRACT
RBDO	RIGHT BASE DOOR OPEN
LBDO	LEFT BASE DOOR OPEN
RBDC	RIGHT BASE DOOR OPEN
LBDC	LEFT BASE DOOR CLOSE

The following lists provide a brief description of the resource states and operations modeled by the PN in figure 3.

VAN RESOURCE CYCLE:	SENSORS REQUIRED:
V0 = Van at prep booth stop.	PLS1
V1 = Van arrived in base booth.	BLS1
V2 = Van in base booth painting position.	BLS2
V3 = Base coat applied to van.	NONE
V4 = Van at base booth doors.	BLS3
V5 = Van arrived in clear booth.	CLS1
VE1= Failed to move into paint position	BPLS2 and BLS1
VE2= Failed to move off grounding bars	BPLS2 and BLS2

BASE BOOTH PUSHER RESOURCE CYCLE:	SENSORS REQUIRED:
BP1 = Base pusher retracted and waiting for van to arrive	BPLS1
BP2 = Base pusher extended with van in the back dog (thus the van is in the painting position).	BPLS2
BP3 = Base pusher retracted while the van is in the painting position.	BPLS1
BP4 = Base pusher extended with van in the front dog (thus the van is pushed past the painting position).	BPLS2

BASE BOOTH DOORS RESOURCE CYCLE:	SENSORS REQUIRED:
BDO1 = Opened for van to pass through	BLDO and BRDO
BDO2 = Base doors open and van passed	BLDO, BRDO, CLS1
BDC1 = Base doors closed for painting	BLDC and BRDC
BDC2 = Base doors closed, painting complete	BLDC and BRDC
BDOE = Error base door open (the doors did not open)	BLS3 and NOT BLDO
BDCE = Error base doors close (the doors did not close)	BLS2 and NOT BLDC and NOT BRDC

BASE BOOTH RESOURCE CYCLE:	SENSORS REQUIRED:
BBF = Base booth clear (empty) and waiting for the next van.	NONE

CONVEYOR RESOURCE CYCLE:	SENSORS REQUIRED:
CS = Conveyor stopped.	NONE

OPERATIONS:	ACTUATORS REQUIRED
OP1 =Drop stop in prep booth and allow van to move into base booth.	PBSD
OP2 =Put van into base booth painting position by extending base pusher.	BPEXT
OP3 =Retract base pusher.	BPRET
OP4 =Apply base coat to van.	NONE
OP5 =Extend base pusher to push van past painting position.	BPRET
OP6 =Open base booth doors.	RBDO and LBDO
OP7 =Retract base pusher to accept new van arriving in base booth.	BPRET
OP8 =Stop conveyor to prevent van from hitting base doors.	NONE
OP9=Move van from base doors to clear booth pusher.	NONE
OP10=Close base booth doors.	RBDC and LBDC
OPE1=Manual reset of base pusher and van in paint position	NONE
OPE2=Manual reset of base pusher and van off grounding bars	NONE
OPE3=Manually open of base doors and restart conveyor.	NONE
OPE4=Manually close base doors	

II. DBBUILD User's Manual

ELI Introduction

0BBUHiD *m* iatetacthre prog^® used to obtain and store information concerning a naarfactarag ifsim[1]. The structure of DBBUILD emulates a Petri net model to s aaiJpii of lit *q/tim* lope using existing Petri net theories. The purpose of this append familiarise (ft* awr with DBBUILD's structures and menues. DBBUILD prompts the user lafbtrmftlibi thai is required and therefore an experienced programmer would fee] wttlbrtmbfe ariag BIJBUILD without first reading this manual. However, DBBUILD wi! far ufoivuUiM tfctt nay see© irrelevant; this manual tries to explain the need for these ^

EL3 Structure

Tit dkfca l»s» is eomplissd of four major record types: 1. operations, containing infon os iipit *mi* «MsMBt tnuBsitikHis, resource states, and actuators; 2. resources, cott *Uhnnlim m* ttt f^Mi^[1] .^ato aad the sensors data required to define each state; 3. *m* ttMiiyiaitg tit aAdras Ubd *dt* the sensor input port; and 4. actuators, containing the *m* ^W fe^ tl§ *mtomtom* osi|wt port Schematics of the records are shown in figures 4 throi **Tit** tepid ISieiiiii &i Afa prl of lie manual .are for use by those who have an undersfl rflrt^^^ hlgmgi. HnptitiTOa ^ff the material is not required to use DBBUIOii

II.3.1 Operation *WmmmiM*

Tit^ ^ ^ ^b As lop Iml *1bw£m h* ;ft# cferation record:

```
 ^W*    MHI ^JiliJ;        -operation name defined by user-
 i^S*   #i^ i^i£il^;       . operation description
 ti»    ^mjMjs§;           " iolis the number of input transit^
 int    num_assoc_act;     feolAt t&e number of output transl^
 struct operation_type     bolis number of associated actuat®H
 struct operation_type
                           tpreir;
 struct in_op *in_op_ptr;  pointer to list of input transitio
 struct out_op *out_op_ptr; pointer t/O list of output transitio
```

[1] The authors would like to thank Wayne Figurelle for developing the C code for DBBUILD.

```
     struct act_list *assoc_act_ptr; pointer to list of actuators
                                     affected by the operation
```

The following structure contains information on the associated actuators

```
typedef struct act_list {
     char     name [NAME_SIZE];       structure name defined by DBBUILD
     char     desc [DESC_SIZE];       not used
     char     act_name [NAME_SIZE];     name of the actuator
     char     assoc_op_name [NAME_SIZE];    not used
     char     act_cond [COND_SIZE];   the condition of the actuator
                                      defined by user
     struct   act_list              *next;
     struct   act_list              *prev;
```

The following structure holds information on the input transitions

```
typedef struct                      in_op  {
     char     name [NAME_SIZE];       DBBUILD name of the transition
     char     desc [DESC_SIZE];       not used
     int      num_in_op_AND;          number of resource states associated
                                      with the transition
     struct   in_op                 *next;
     struct   in_op                 *prev;

     struct in_op_AND *in_op_AND_ptr;  points to a list of the resource
                                    states associated with the transition
```

The following structure holds information on the output transitions:

```
typedef struct out_op  {
     char     name [NAME_SIZE];
     char     desc [DESC_SIZE];
     int      num_out_op_AND;
     struct   out_op                *next;
     struct   out_op                *prev;

     struct   out_op_AND            *out_op_AND_ptr;
```

The following structure holds the input transition's resource states;

```
typedef struct in_op_AND  {
     char     name [NAME_SIZE];       structure name defined by DBBUILD
     char     desc [DESC_SIZE];       not used
     char     res_name [NAME_SIZE];     the resource name
     char     state_name [NAME_SIZE];   the resource state name
     struct   in_op_AND             *next;
     struct   in_op_AND             *prev;
```

n. following .truclur. holds the output transition "Bource .«.«..

```
tjpedef        struct              out_op_AND   <
      char     name  [NAME_SIZE] ;
      cb^      desc  [DESCJSIZE];
      char     res_naine  [NAME_SIZE] ;
      char     statejiame  [HAMEJSIZE] ;
      fltruct  out_op_AND            *next;
      •tract   out_op_AND            *prev;
```

H9L2 E^nrce Records

The folfowlag » the resource record and ite components:

```
Tjrptdrf struct resource_type {
Oiwr B«MCIIA1IEMSIZE];                      Name of the resource
chix dMcCDESC^siZE];                        Description of the resource
Struct rt«ottrce_type *next;
Strmct resource_type *prev;
lot num_state;                              Holds the number of different
                                            states "the resource has

Struct state_type      *statejptr;          Points to tlie resource state
                                            structure
```

IHt following stntcfeurs contains information on the resource states:

```
Tjrptiff struct state type i                The resource state structure
Char MBmtWMiM^BtZBl"**                       Name of the state
Char 4ssc CDB8CJBZZE]                       Description of the state
Char latched                                Used for generating the IL
                                            code
Struct stat type *next
ttrvct ttattjbyp* *prev
lnt avm_i i '*                              Number of sensors used to
                                            determine the state

struct OR_typt        *01j>tr               Points to the series of
                                            sensors used t>o define state
```

The foUaviag tferttctws coatalna the name of the series of sensors used to *m* fine a *BpmitU4* r«»o«rc« state;

```
ffp§4%t struct OR_type
ouur name [NAME_SIZE]                       DBBUILD struc-fcure name
Char desc [DESC_SIZE]                       not used
ttntct OR_type *next
•tract OR_type *pr#T
Xat aua_AND                                 Number of sensors in series
```

Struct AND_type	*AND_ptr	Pointer to the sensors in the series

The following structure contains the sensor names for a specified series

Typedef struct AND_type	
Char name[NAME_SIZE]	DBBUILD structure name
Char desc[DESC_SIZE]	not used
Struct AND_type *next	
Struct AND_type *prev	
Char sensor_name[NAME_SIZE]	Sensor name
Char sensor_cond[COND_SIZE]	The state of the sensor - activated/not activated
Char assoc_res_name[NAME_SIZE]	not used

II.2.3 Actuator Records

The actuator record is defined as follows:

Typedef struct actuator	Actuator structure
Char name[NAME_SIZE]	Name of the actuator
Char desc[DESC_SIZE]	Actuates description
Struct motion_struct	Indicates different actuator/ motions
Int wire_num	Actual wire number
Struct actuator *next	
Struct actuator *prev	
Int num_assoc_op	Number of operation in which actuator is used
Struct assoc_op	Points to an operation

The following structure holds information on the operations in which the actuator is used:

Typedef struct assoc_op	
Char name[NAME_SIZE]	
Char desc[DESC_SIZE]	
Char op_name[NAME_SIZE]	Name of the operation
Char act_list[NAME_SIZE]	Not used
Struct assoc_op *next	
Struct assoc_op *prev	

II.2.4 Sensor Records

The sensor record is as follows:

```
Typedef struct sensor type
Char name[NAME_SIZE]                        sensor name
Int wire_num
Char desc[DESC_SIZE]                   description of the sensor (optional)
Int cond                               condition the sensor will be in when
                                          actuated

Struct sensor_type *next
Struct sensor_type *prev
Int num_assoc_res                           Number of resources for
                                            which this sensor is used

Struct assoc_res   *assoc_res_ptr    Pointer to associated resources
```

The following structure contains information on resource states in which the sensor is used:

```
Typedef struct assoc_res
Char name[NAME_SIZE]                        DBBUILD structure name
Char desc[DESC_SIZE]                        not used
Char res_name[NAME_SIZE]                    Resource name
Char state_name[NAME_SIZE]                  State name
Struct assoc_res *next
Struct assoc_res *prev
```

II.3 Menus

The menus used to prompt the user use terms used to describe elements of Petri nets. Most menu options are self explanatory; however, those options that are not will have a brief explanation following the menu listing.

The top level menu, and therefore the first one you see, allows you to choose which record you want to investigate. This menu is as follows:

```
S = For sensor data type
R = For resource data type
O = For operation data type
A = For actuator data type
Q = To quit this program

Which type do you want to alter or look at?
```

II.3.1 Operation Menu

If at the top level you decide to look at operations, the following menu will appear:

```
I-INSERT new operation
D-DELETE an operation
F-FIND an operation or some info about an operation
A-INSERT assoc. actuator for this operation
P-INSERT an out op cond OR header for this operation
C-INSERT an out op cond AND header for this operation
O-INSERT an in op cond OR header for this operation
H-INSERT an in op cond AND header for this operation
L-LIST all of the names present
Q-Quit, and look at another data base
?-List all of the commands availabel
```

"P" will generate the structure for an output transition and name that transition TRANS_(n); where n is a number DBBUILD maintains. Once the transition has been named; DBBUILD will ask if there are any resource states that you want to attach to this transition. Upon entering a state DBBUILD will generate a structure to hold the state name. DBBUILD will name this structure STATE_(n) much in the same way it names the transitions.

"C" can be used to add additional resource states to an existing output transition. DBBUILD will first ask for th output transition name (TRANS_1, TRANS_2, etc.) and then allow you to enter a resource state.

"O" and "H" perform the same as "P" and "C" respectively, but are used for input transitions rather than output transitions.

NOTE 1:

The words "OR" and "AND" used in the menus refer to transitions and resource states associated with that operation respectively. OR is used for transitions because they represent the different enabling or resulting sets of resource states. AND is used for resource states within a transition because all of the resource states must be satisfied for that transition to be enabled.

NOTE 2:

The labels TRANS_(n) and STATE_(n) are used by DBBUILD to search through the record.

See struct in_op_OR and struct in_op_AND in section 3 of this manual for more information.

"F" will cause DBBUILD to prompt the user for an operation name and will then display the next menu containing new options.

```
D-To see the description of the operation
A-To list all of the assoc. actuators with this operation
F-To find info about assoc. actuators with this operation
O-To list all of the out ops assoc. with this operation
N-To get info about the out ops assoc. with this operation
I-To list all of the in ops assoc. with this operation
G-To list all about the in ops assoc. with this operation
Q-To quit looking at this operation
?-To see these commands
```

"O" will list the names of this operations output transistions (TRANS_1, TRANS_2, etc.).

"N" will cause DBBUILD to ask for the output transition name and then present the resource states associated with that transition.

"I" and "G" will perform the same tasks as "O" and "N" respectively except they are used for input transitions.

The following menus are presented when the "N" and "G" options are chosen from the previous menu:

```
D-To see the description of the out_op
L-To list all of the ANDs present
R-To see the resource name and the state name of an AND
Q-You are done looking at this out_op
?-To see these commands
```

```
D-To see the description of the in-op
L-To list all of the ANDs present
R-To see the resource name and the state name of the AND
Q-You are done looking at this in_op
?-To see these commands
```

The following menu is presented when the "F" option is used in the previous menu:

```
D-To see the description of the assoc_act
C-To see the condition the sensor will be in after the op
```

```
L-to list all info about the assoc. actuator for this op
Q-You are done lookinga t this assoc_act
?to see these commands
```

II.3.2 Resource Menu

If from the top level you decide to work on the resource record, the following menu will be presented:

```
I-INSERT new resource
D-DELETE a resource
F-FIND a resource or some info about a resource
L-LIST the name and descriptions of the resources present
S-Insert a STATE to a resource
E-ELIMINATE a state from a resource
O-ADD a new SERIES of SENSORS to a given state
A-ADD a SENSOR to a given series of a given state
T-TRASH (delete) a SERIES of SENSORS from a given state
W-Delete a SENSOR to a given series of a given state
Q-Quit, and look at another data base
?-List all of the commands availabel
```

NOTE:

AS a resource cycles (or is cycled) through the systems operations, its state will change. These states may or may not be defined by sensors, and in addition some states may be defined by more than one set of sensors. For example, some arbitrary state may be defined by sensors 1 and 2 or by sensors 3 and 4. DBBUILD's terms for these sets of sensors is SERIES; i.e. sensors 1 and 2 would be listed in SERIES_1 and sensors 3 and 4 would be listed in SERIES_2. DBBUILD uses the word SERIES__(n) to label the structure that contains the pointer to each of the sensors. See struct OR__type in section 3 of this manual. Additionally DBBUILD uses SENSOR__(n) as the name of the structure that holds the actual sensor name. See struct AND__type in section 3 of this manual.

If the "F" option was chosen to find information about a resource, the following menu will appear:

```
D-To see the description of the state
S-To get info about a particular state
L-to list all of the states assoc. with this resource
Q-To QUIT looking at this resource
?-to see these commands
```

If at this level "S" is requested the following menu will appear:

```
D-To see the description of the state
L-To list the SERIES of SENSORS assoc with this state
O-To see info about a particular SERIES
Q-You are done looking at this state
?-To see these commands
```

If the "O" option is chosen the following menu will appear:

```
L-To list SENSORS assoc with this SERIES
S-To list all of the sensor names under this SERIES
  and their conditions
A-To see info about a particular associated sensor
Q-You are done looking at this SERIES
?-To see these commands
```

If at this level the "A" option is used DBBUILD will ask for the sensor name, SENSOR_1, SENSOR2, etc. This version of DBBUILD does not contain additional information on sensors beyond what the "S" option provides.

II.3.3 Actuator Menu

If at the top level you requested to enter the actuator record the following menu would appear:

```
I_INSERT new actuator
D-DELETE an actuator
F-FIND an actuator or some info about an actuator
L-LIST all of the names present
Q-Quit, and look at another data base
?-List all of the commands availabel
```

The find command invokes the following menu:

```
D-To see the description of the actuator
S-Get info about a particular assoc op
M-to list all of the motions this actuator has
L-To list all of the assoc op with this actuator
Q-To QUIT looking at this actuator
?-to see these commands
```

II.3.4 Sensor Menu

If at the top level you entered the sensor record, the following menu would appear:

```
I-INSERT new sensor
D-DELETE a sensor
F-FIND a sensor
L-LIST all of the sensors present
W-Change the WIRE number assoc with a sensor
Q-To quit and look at another data base
?-List all of the commands availabel
```

The find option will cause the following menu to appear:

```
D-To see DESCRIPTION of the sensor
L-To LIST all of the states that this sensor is used to define
W-To see the WIRE number of this sensor
Q-When you are done looking at this particular sensor
?-List these commands
```

References

1. J.L. Peterson, *Petri Net Theory and the Modeling of Systems,* Prentice-Hall, Inc., Englewood Cliffs, NJ., 1981.

2. C.L. Beck, "Modeling and Simulation of Flexible Control Structures for Automated Manufacturing Systems", Tech. report, Robotics Institute, Carnegie Mellon University, 1985.

3. C.L. Beck and B.H. Krogh, "Models for Simulation and Discrete Control of Manufacturing Systems'[1], *IEEE International Conference on Robotics and Automation,* San Francisco, April 1986.

4. B.H. Krogh and C.L. Beck, "Synthesis of Place/Transitions Nets for Simulation and Control of Manufacturing Systems", *Jth IFAC/IFORS Symposium Large Scale Systems,* International Federation of Automatic Control, Zurich, August 1986.

5. International Electrotechnical Commission, *Standard for Programmable Controllers, -Part 8: Programming Languages,* 1982, Technical Committee 65: Industrial Process Measurement and Control