Computer Science and Systems Analysis

Computer Science and Systems Analysis

Technical Reports

Miami University *Year* 1995

"State-based Control Language a State-based, Interrupt-driven, Concurrent Language with Error Detection and Recovery"

George Hellstern
Miami University, commons-admin@lib.muohio.edu

MIAMI UNIVERSITY

DEPARTMENT OF COMPUTER SCIENCE & SYSTEMS ANALYSIS

TECHNICAL REPORT: MU-SEAS-CSA-1995-003

State-based Control Language a State-based, Interrupt-driven, Concurrent Language with Error Detection and Recovery

George Hellstern

School of Engineering & Applied Science | Oxford, Ohio 45056 | 513-529-5928

STATE-BASED CONTROL LANGUAGE A STATE-BASED, INTERRUPT-DRIVEN, CONCURRENT LANGUAGE WITH ERROR DETECTION AND RECOVERY

by

George Hellstern
Systems Analysis Department
Miami University
Oxford, Ohio 45056

Working Paper #95-003 June, 1995

ABSTRACT

STATE-BASED CONTROL LANGUAGE A STATE-BASED, INTERRUPT-DRIVEN, CONCURRENT LANGUAGE WITH ERROR DETECTION AND RECOVERY

by George Hellstern

A programming environment to support interactive, concurrent programming for Miami University's Flexible Manufacturing System is described. The environment is intended to replace the existing extended cell programming language (ECPL), which was sequential in nature, to a state-driven one in order to support concurrency. The system has been altered from being sequential to reactive and is interrupt driven. This also enhances error detection and recovery capabilities. This paper will address the topics of Flexible Manufacturing Systems (FMS) and programming languages for manufacturing control, and will then develop a criteria for analyzing a FMS programming language. Based on that criteria, the former ECPL language will be discussed and analyzed to address problems in ECPL. The paper will conclude with an analysis of the new system outlining areas for further investigation and improvement.

State-Based Control Language

A State-Based, Interrupt-Driven,
Concurrent Language with Error Detection and Recovery

A Thesis

Submitted to the
Faculty of Miami University
in partial fulfillment of
the requirements for the degree of
Masters of Science
Department of Systems Analysis

By
George Hellstern
Miami University
Oxford, Ohio
1995

Advisor Douglas A Troy
Reader Jessica Parsons
Reader Alton F Sanders

1. Introduction

A Flexible Manufacturing System (FMS) defined by Buzacott, (1980) is, "a set of machines . . . linked by a material handling system all under central computer control." A typical FMS consists of one or more numerically controlled (NC) machines linked by an automatic guided vehicle or a conveyor belt. Other components typical to a FMS environment are as follows: machine queues, common storage areas, bar coding devices, tool management systems, and database/knowledge base and networking systems (Rau, 1993). FMSs have been designed as a means to automate manufacturing facilities targeted for manufacture of a large varieties of parts such as a job shop facility, operating in the medium to small batch size range.

As technology continues to advance in the areas of NC machine tools and robotics, FMSs are becoming more common in manufacturing. There are five specific advantages a FMS can provide to the manufacturing environment (Groover, 1980):

1. Higher machine utilization
2. Reduced work-in-process
3. Lower manufacturing lead times
4. Greater flexibility in production scheduling
5. Higher labor productivity

Each of these advantages translates into greater potential for satisfying customer demand at lower costs, making a businesses more profitable.

The key quality of a FMS is its ability to adapt to changes in production. The

FMS is able to be reprogrammed to allow for the creation of a new part, or for the addition (and subtraction) to the set of machines with which the system is operating. One major part of the FMS that must be reprogrammed is the factory floor workcells, using a workcell programming language. This programming requirement can become a disadvantage to the smaller manufacturing companies operating without a systems programmer. The simplest of workcell programs require an intimate knowledge of machine level coding and data communications lacking in most engineers. Also, most workcells will contain more that one machine, each machine operating with a different command language, making programming a workcell a complex task. To add to the complexity, command languages for manufacturing machines are usually low-level languages slowing down the process of generating and debugging workcell programs even further. Due to these several considerations, easy to use workcell programming languages are needed for the typical FMS environment. Such a language should provide the workcell programmer with a high level of abstraction to hide the specifics of the machine-level coding and enable fast generation of readable programs.

The purpose of this research is to investigate and extend a workcell programming language called CPL created for controlling the Flexible Manufacturing System in Miami University's CIM lab. First, a criterion upon which to base an analysis of workcell programming languages will be developed in order to more objectively determine the functionality and usability of a workcell language. The history of the CPL system will then be discussed, after which the latest version of CPL will be analyzed for strengths and weaknesses. The paper will then conclude with the presentation of a new language

developed as a result of this research.

2. Design Criteria for Workcell Languages

This section presents the requirements for a workcell programming language. These requirements are based upon the work of Rau (Rau, 1993):

1) A FMS workcell must be able to distinguish between differing types of manufacturing devices. For example the language must distinguish between input devices such as a limit switch or a photocell, and output devices such as a robot or a conveyor belt. An "object construct" must be developed in order to account for the differing objects attached to the control computer.

2) The language must be able to send messages to FMS workcell components that are controllable, and be able to allow message passing between components if necessary. A message data structure would enable control as well as error detection and recovery.

3) The language must allow the programmer to specify time-based actions. For example a certain function in the workcell may require a pause for a certain period of time, or an operation may be expected to be processed within a given time interval.

4) The language must process variable type information. The workcell may be processing more than one part at a time, and need a variable associated with the differing part types, or run times may require some user input, which would require a variable to store and process such information.

5) The language must incorporate flow-of-control constructs such as conditional execution (if. . .then) and iterations (loops).

6) Ease of use is another important criteria. The language should be easy to use as well as complete; an appropriate interface should be created. A complex, unwieldy language defeats the purpose of creating a workcell language.

7) The language should support concurrency. In a manufacturing environment, efficiency is essential to success; building a system which will support concurrency is needed to minimize the makespan of production (the time from start to finish to process a part). By establishing concurrency, a lengthy task can be completed on one component while another task can be under way at the same time on a different component.

The functionality and usability of a workcell programming language is greatly determined by the degree in which that language fulfills these seven specific functions. These functions are based upon the general characteristics inherent to all FMS systems. The most complete workcell language will meet all seven criteria.

It is to be emphasized that it is the functions of the above criteria that are the important "requirements" for a FMS workcell programming language and not those specific language constructs. The language components themselves merely represent the functionality a given workcell is inherently capable of possessing. For example: a language consisting of state diagrams will not have a literal 'if' statement in it; however, that language will posses the capability of making 'if . . . then' types of decisions.

The following sections provide a history of the CPL programming language, and examine the former two designs of CPL, analyzing them for their strengths and

weaknesses.

3. The CPL System

3.1 Miami University's FMS Workcell

The Miami University CIM lab was developed for the purpose of instructing students in the new technologies of computer integrated manufacturing (CIM). Students learn about computer numerically controlled machines (CNC), the use of Autocad for Computer Aided Design (CAD) and Computer Aided Manufacturing (CAM). The CIM lab also includes a FMS workcell that was designed and constructed by Miami University faculty and contributing companies. The workcell is used to introduce the students to an automated manufacturing environment which they may encounter in a job shop or batch production facility. The students are required to program the workcell as part of the lab curriculum.

The Miami University's FMS workcell currently has two robotic arms, an Automatic Storage and Retrieval system (AS/RS), and a CNC Mill and Lathe all placed around a material handling conveyor-belt. The workcell is designed to simulate getting a raw part out of stock, cutting it, and putting it back into stock. Around the conveyor-belt are stops used to control the flow of a pallet within the system. Limit switches and one photocell provide feedback involved in detecting pallet arrival at various stations. See Figure 3.1.

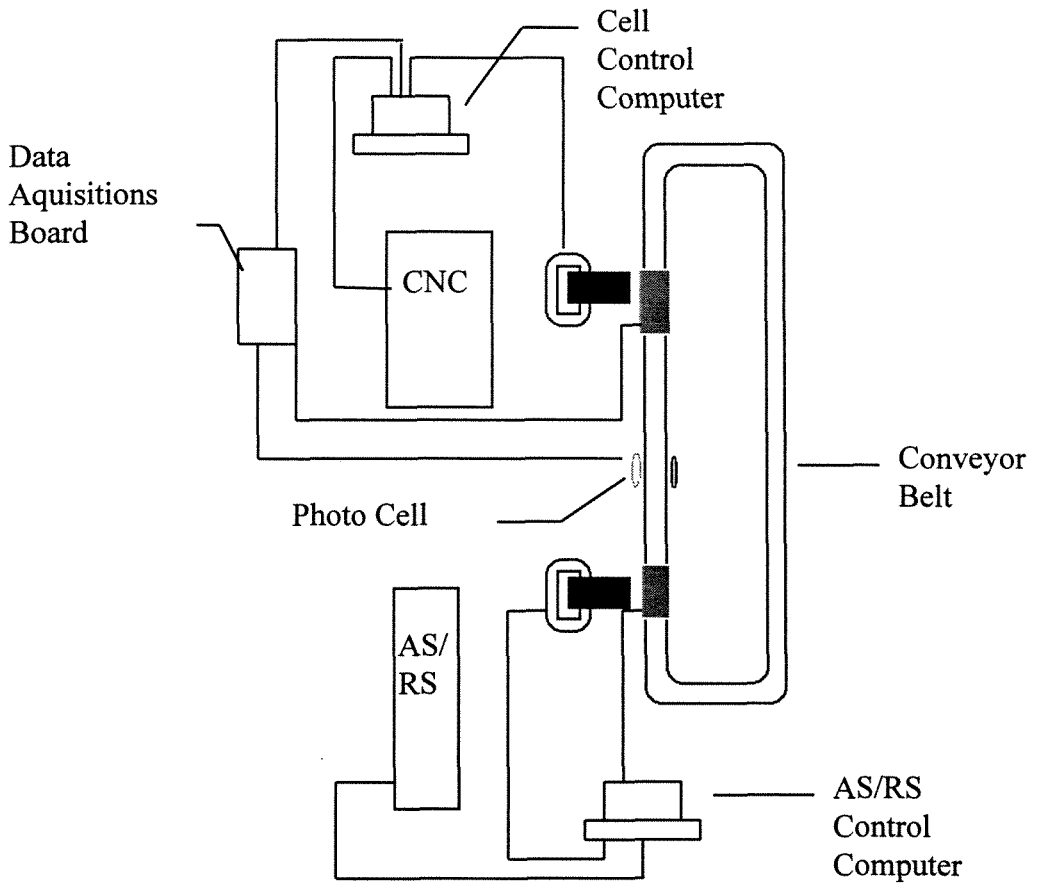


Figure 3.1 - Miami University FMS Workcell

The two robotic arms are currently operated using two separate control computers. The first control computer is responsible for operating the AS/RS, and the second control computer is responsible for the rest of the cell. The two computers function independently of each other as two individual units; one controls loading and storage of a part, while the other controls the manufacture of finished parts. The computer responsible for the AS/RS is able to control the conveyor-belt and pallet stops through a robot controller connected to an LPT port. The computer responsible for part production is able to control all aspects of the conveyor system (pallet stops, photocell, limit switches, and conveyor belt) through a data acquisition board connected to the bus.

3.2 History of CPL

Before a cell programming language was utilized for the workcell, students were responsible for programming the system using assembly language and the robot controller. Few students were able to learn the language well enough to write workable programs due to the limited lab time. The first cell programming language, the original CPL, was created by Meghamala (1992) and Farooq (1992) in 1992 and was later updated in 1994 into a version called Extended CPL by Wang (1994). Since the introduction of Cell Programming Language, all students have been able to write one or more programs for the workcell, there has been more time available for the students to learn key principles of data communication, and students have had greater opportunities to develop their own projects using the FMS workcell.

The following three sections describe the original CPL version and the extended version and conclude with an analysis of extended CPL's strengths and weaknesses.

4. The Original CPL

The first version of CPL consists of three sections of programming code: the ports declaration section, the device declaration section, and the procedure declaration section. It is described in detail in Meghamala (1992) and Farooq (1992).

4.1 Ports Declaration

The ports declaration section is used to identify the input/output ports being used by the control computer to communicate to the workcell's components. This section allows for any number of COM ports, LPT ports, or addresses on the data acquisition board to be declared. The declaration requires the name of the port (user defined), the address of the port, and an indicator as to whether the port is designated for input or for output. A sample is shown in Figure 4.1:

```
Ports
    PortA  640     Input;
    PortB  641     Output;
    PortC  642     Output;
    Com1   COM1 2400 7 1 2;
End
```

Figure 4.1 - Ports Declaration Section

Note that for COM ports and LPT ports, the baud rate, data bits, parity, and stop bits are specified here in the Ports declaration section.

4.2 Device Declaration

The device declaration section of the original CPL is similar in purpose to a Pascal *var* section, where all of the program's user-defined variables are to be declared. The variables defined in this section are representative of the devices connected to the control computer (not to be confused with a Pascal data type). There are five possible data types a CPL programmer can use to declare device variables, each corresponding to

a generic type of workcell object. They are as follows:

- Coil - A device which will either be energized or not energized.
- Sensor - A device which will supply input to the computer upon sensing an activity. An example of this type of device is a limit switch or a photocell.
- Pulse - A device which will change its state when it receives a short on/off signal (or pulse).
- Programmable - As the name suggests, this device will receive a list of code which will in effect program it to do something. An example of this is the robotic arm.
- Wait - A time oriented data type for creating pauses between operations.

Each device data type has associated with it a set of commands (or methods) that it is capable of processing. These are the only functions the given data type is capable of performing. A command statement consists of the concatenation of the device name, a period, and the command word. The commands are shown in Table 4.1:

Device	Function
Coil	On/Off
Sensor	WaitOn/WaitOff
Pulse	Strobe
Programmable	Send/Do
Wait	Milliseconds

Table 4.1 - Devices and Commands

In order to declare a device, the user must first define the name of the device, give

the device type, then the port assignment, and the data acquisition bit associated with the device being declared (devices connected to COM or LPT ports do not have any data acquisition bits associated with them). All CPL statements end with a semicolon. Many such examples are shown in Section 4.4.

4.3 Procedure Declaration

The final section of an original CPL program is the logic section, called the *procedure declaration* section. Using the declared objects representing the connected devices, a series of commands invoking the functions available on these objects are coded, for example:

```
Robot.Do(LoadLathe);
```

Notice the structure for a command is simply the name of the cell object, followed by a period, followed by the function, its optional parameter, and a semicolon.

4.4 An Example of an Original CPL program

Figure 4.2 shows a simple program written in original CPL. This program is designed to send the nest command ("NT") to the device named "Robot" which is connected to the PC Com1 port. The COM port is declared in the Ports section and the "Robot" is specified in the Devices section. The program then turns on the conveyor belt (also specified in the Devices section), and waits for the photocell to be triggered. This operation has a 5000 millisecond time-out parameter. Upon activation of the photocell,

the conveyor is turned off and the robot is sent the sequence of commands needed to move a part. The robot's commands are in the file named StorePart.CMD.

```
Ports
    PortA  640     Input;
    PortB  641     Output;
    Com1   COM1 300 7 2 0;
End

Devices
    Photocell      Sensor PortA  7;
    Conveyor       Coil   PortB  5;
    Robot          Programmable  Com1;
End

Procedure
    Robot.Send("NT");
    Conveyor.On;
    Photocell.WaitOn(5000);
    Conveyor.Off;
    Robot.Do(StorePart);
End
```

Figure 4.2 - Typical, Simple CPL Program

Notice that a CPL program has the following limitations:

1.) Execution is strictly sequential with no iterations.
2.) Execution of operations in a procedure occur serially from top to bottom, i.e. there is no concurrency.
3.) There is no error recovery associated with time-outs.
4.) There are no sub-procedures.
5.) There is no operator interfaces for display of messages or input from the operator.

4.5 Development and Execution of the Original CPL

To write and execute a CPL program the user has to first write the program using a text editor, compile it using the CPL compiler, and then use the CPL interpreter to

execute the code. The compiler transforms the source code into an intermediate form called p-code, which consists of the original textual code interleaved with the numeric op-codes for the interpreter. See Figure 4.3.

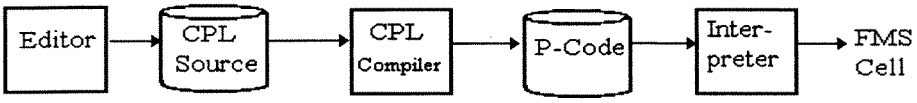


Figure 4.3 - Execution of a CPL Program

Due to the limitations listed in section 4.4, the original version of CPL was extended. The extended version, called "Extended CPL," is described in the next section.

5. Extended CPL

The extended version of CPL (ECPL) was created in an effort to develop error recovery, flow control, an operator interface, and sub-procedures. Each of these features are reviewed below, and described in detail in Wang (1994).

5.1 Procedures

Programs written in ECPL retain the organization of having a *ports declaration* section, a *device declaration* section, and a *procedure declaration* section; however, the *procedures declaration* section has been renamed to *program declaration* section and an option of creating sub-procedures (or subroutines), called *procedures,* has been developed. In the same manner that Pascal requires procedures and functions to appear

above the main program body, the subroutines (*procedure sections*) of an ECPL program must appear above the p*rogram declaration* section. The structure of an ECPL program is shown in Figure 5.1.

```
Ports
. . .
End

Devices
. . .
End

Procedure A
. . .
End

Procedure B
. . .
End
. . .

Program
. . .
End
```

Figure 5.1 - Structure of an ECPL Program

5.2 Error Detection and Recovery

The original CPL lacks error detection or recovery. For example, it does not allow for recovery from time-out error detection on a sensing device. The system will wait forever for a signal that may never arrive. The user is therefore left to guess which function the system is waiting for, if he or she had not been watching at the moment the

error occurred. Extended CPL allows for time-out error detection of sensing devices, and has the option of specifying an error routine to take control of the system if such a situation were to occur. This adds error recovery to simple error detection. An example of such a statement that will send control to a sub-procedure named "ErrorHandler" upon the timing out of the sensing device "PalletArrived" is as follows:

PalletArrived.WaitOn(5000):ErrorHandler;

See section 5.5 for an extended example of how this statement might be used.

5.3 Flow Control

One criteria discussed in Section 2 of this paper is the need for decision constructs in a workcell language in order to allow for the functionality of flow control. Looping constructs also are mentioned as a need in a workcell programming language to allow for repetitive processing of manufacturing routines. ECPL includes both decision constructs and looping constructs. The language components: 'If', 'While', 'Until', and 'For' have all been included with ECPL, allowing for flow control not possible in the older version. Section 5.5 gives an extended example of how these statements work together.

5.4 Operator Interface

Two new object types were added in ECPL to support an operator interface: IO and String.

The IO device, which can be thought of as an external device attached to the

central computer, is used for a textual display and for input from the operator. IO devices use the monitor as an output device and the keyboard as an input device. Operations on IO devices include "Get," "Put," and "Poll."

The String type is similar to a Pascal string data type. It allows for the declaration of a variable to receive a string of information from the user, through the keyboard.

The commands associated with these two devices are shown in Table 5.1.

Device	Function
IO	Get/Put/Poll
String	Conditional operations on strings: EQ/NE/GT/LT/GE/LE

Table 5.1 - Devices and Commands for IO and String Data Types

Strings and IO's are both declared in the device declaration of the program. Example declarations of these objects and their usage are as follows:

```
Devices
    InputA      String;
    Terminal    IO;
End

Procedure ProcedureA
. . .
End

Procedure ProcedureB
. . .
End

Program
    Terminal.Put("Welcome to Miami University's FMS");
```

```
    Terminal.Put("Please Enter Part to processes. --->");
    Terminal.Get(InputA);
    ProcedureA.Run(If InputA.EQ("001"));
    ProcedureB.Run(If InputA.EQ("002"));
End
```

This program exhibits the usage of IO's, Strings, and decision constructs, demonstrating one example of where these constructs might work together for flow control in a typical FMS workcell.

5.5 An Example of an Extended CPL program

Figure 5.2 gives a thorough example of an ECPL program designed to utilize many of the new design criteria emphasized in ECPL. This program processes two individual parts "001" and "002." The user is prompted for the part he or she wants to process, and how many times he or she would like to process it. Then the robot is told to process the part by the command file in Part001.cmd or Part002.cmd. (Please note that all files being sent to a programmable device must be stored with a file with ".cmd" extension, although as a parameter the ".cmd" extension is not included explicitly. See Wang (1994) for more specific details.) In the case of an error, control is sent to ErrorHandler where processing is stopped until a key is pressed:

```
Ports
    PortA           640     Input;
    PortB           641     Output;
    PortC           642     Output;
End

Devices
    Conveyor        Coil    PortC   5;
    PalletArrived   Sensor  PortA   6;
    PalletLiftUp    Pulse   PortC   4;
    PalletLiftDown  Pulse   PortC   6;
```

```
    Robot           Programmable    COM1;
    InputA          String;
    InputB          String;
    Terminal        IO;
End

Procedure ErrorHandler
    Conveyor.Off;
    Terminal.Put("ERROR, Pallet has not arrived!");
    Terminal.Put("Press Any Key to Continue");
    Wait(Until Consol.Poll);
    Conveyor.On;
End

Procedure ProcedureA
    PalletArrived.WaitOn(5000):ErrorHandler;
    PalletLiftUp.Strobe;
    Robot.Send(Part001);
    PalletLiftDown.Strobe;
End

Procedure ProcedureB
    PalletArrived.WaitOn(5000):ErrorHandler;
    PalletLiftUp.Strobe;
    Robot.Send(Part002);
    PalletLiftDown.Strobe;
End

Procedure Interface1
    Terminal.Put("Welcome to Miami University's FMS");
    Terminal.Put("Please Enter Part to processes. --->");
    Terminal.Put("Press 'Q' To Quit");
    Terminal.Get(InputA);
End

Procedure Interface2
    Terminal.Put("How many parts to processes. --->");
    Terminal.Get(InputB);
End

Procedure SubControl
    ProcedureA.Run(If InputA.EQ("001"));
    ProcedureB.Run(If InputA.EQ("002"));
End

Procedure Control
    Interface1.Run;
    Interface2.Run;
    SubControl.Run(For InputB);
End

Program
```

```
    Conveyor.On;
    Control.Run(While InputA.NE("Q"));
    Conveyor.Off;
End
```

Figure 5.1 - Example of workcell program written in ECPL

6. Analysis of Extended CPL

This section will examine ECPL to identify its strong and weak points. The analysis will first use the design criteria developed in Section 2 of the paper and apply it to ECPL, then a pragmatic summary of ECPL's strengths and weaknesses is presented.

6.1 Analysis Based upon Workcell Criteria

1) Ability to distinguish between differing devices and their types:

ECPL is able to recognize the various machines connected to the control computer in the workcell. The Devices section includes the possibility of creating seven data types, five of which apply to machines devices connected to the control computer. There is the ability to create unbounded numbers of these devices, allowing for the workcell to grow and expand at any time.

2) The language must be able to send messages to workcell components that are controllable, and be able to allow message passing between components if necessary:

ECPL allows for simple message passing to programmable device types.

3) The language must allow the programmer to specify time-based actions.

The time component of the workcell is handled both by the sensing devices and the wait device type. The wait device type allows for pauses during operation. The sensing device allows for error detection by a "timing-out" of the sensor.

4) The language must process variable type information.

ECPL supports variables, allowing for the processing of multiple parts at the same time. Input can also be received from the user during run time into variables, for processing.

5) The language must incorporate flow-of-control constructs such as conditional execution (if. . .then) and iterations (loops).

Decision constructs necessary to process the string data type have also been included in ECPL. The 'If,' 'While,' and 'Until' statements allow for comparisons of values and for control of the program to be sent to subroutines. ECPL is able to control the flow of a part within the entire system (macro), as well as through individual components of the system (micro). The macro control is accomplished through the program section as well as through the subroutines. The micro control is accomplished through the use of programmable data types, which allows for entire files of commands to be sent to a device. Looping, for sequential iterations of a given manufacturing process is supported in ECPL in three ways: the 'While' statement, the 'Until' statement, and the

'For' statement.

6) Ease of use is another important criteria.

The language presentation style is a simple text editor. Run time interfacing is performed in the DOS environment. With minimal compiler diagnostics, the language can be difficult to write and compile successfully.

7) The language must support concurrency.

ECPL executes statements strictly in sequence. Internally the interpreter uses polling to monitor the state of each external input device. Thus, the interpreter can react to only one event at a time -- the one it is waiting for. If other events occur, for example because of an error or because of concurrent operation of another device in the workcell, that event is simply ignored. That makes ECPL incapable of supporting concurrent control of more than a single operation.

6.2 Summary of Extended CPL's Strengths and Weaknesses

Extended CPL has brought the original CPL from being a simple programming tool to being a language that enables modular programming. The original CPL served to save the student from having to learn the lower level coding required in Assembly language, and provided an interface to the cryptic numeric languages with which most FMS workcell components operate. While the original CPL is a useful tool to enable fast

generation of readable programs, on the other hand it lacks important functionality such as subroutines, user inputs, looping, etc. With ECPL some of that functionality has been replaced, bringing CPL back up to a level near a high level language.

The language's strong points are as follows:

- extensive flow control with looping constructs, conditional branches, and subprograms.
- operator interface
- an unbounded number of devices
- range of computer communications
- good timing elements

The biggest weakness in ECPL is its inability to support concurrent processing. Processing is strictly sequential disallowing for multiple actions to occur at the same time. This detracts greatly from the overall efficiency of the workcell being programmed. For instance the control computer is unable to react to two input devices simultaneously. If it has been programmed to wait for the pallet to arrive at a certain station, it is unable to hear the pallet arriving at a wholly different station. Consequently if the control computer were to get out of sync with the system, it would perhaps never recover. If two processes were to be performed at the same time, the control computer would only be able to react to one of the processes, and would be forced to ignore the other.

Another weakness in ECPL is that it disallows the detection of spontaneous or unexpected errors. The only form of error detection in ECPL is "timing-out" error

detection. The control computer continually polls a sensing device waiting for a signal; if the signal never comes, then an error routine can be given control of processing. For a larger system this is an inadequate procedure due to the fact that the amount of unexpected errors are increased as well as the potential for hazardous situations to arise.

Compiler diagnostics are also a major weakness of ECPL. There are few diagnostics generated, all of which lack a definitive description of the error.

In the next sections, the design of State Based Control Language (SBCL), the latest upgrade of the CPL system, is discussed.

7. Concurrent Systems and SBCL

In the Miami University's CIM lab there is a need for a concurrent system able to simultaneously control both robotic arms as well as the CNC and AS/RS. This concurrent system would enable the entire workcell to be integrated and placed under the control of one single control computer.

In addition, in a production FMS workcell, concurrency is needed to minimize time from start to finish for the system as a whole. By establishing concurrency, a lengthy task can be completed on one component, while another task can be under way at the same time on another component. A workcell that is able to react to all activities occurring in its components enhances error detection and recovery, and increases performance.

A polling system, such as that used in ECPL, is only able to hear one device at a

time and is clearly limited in these areas. In a polling system, input can only be fed back to the control computer during a specific time window. This rules out both concurrency and unexpected events, including errors.

A truly concurrent system is always listening to the system, and responding to the events as they happen. This is called a *reactive system*. This type of system bases its actions upon the state in which the system currently is in, and the events tapping it on the shoulder. Error checking and recovery from errors are significantly improved in a concurrent system due to the fact that it can respond to any event.

In an effort to make CPL support concurrency, the language was redesigned and renamed to SBCL. SBCL, which stands for State-Based Control Language, aims at making CPL concurrent and state driven.

8. State Driven Workcell Programming Packages

The concept of describing FMS workcells using state diagrams has been implemented before. There are several state transition packages available for the purpose of designing and programming a workcell. This section gives a brief overview of some of those packages.

8.1 Ladder Logic

Traditionally the design of a sequential control system such as a FMS workcell was expressed in the language of relay ladder logic. Relay ladder logic was used in

conjunction with a Programmable Logic Controller (PLC). The PLC is a micro computer designed to accept input from sensing devices in a workcell, such as a photocell or a limit switch, perform some logic on that input, and generate outputs to control other external devices, such as motors or valves. The PLC is programmed using relay ladder logic to act as the controller, opening and closing switches in response to the sensing devices connected to it. Relay ladder logic is a graphical language consisting of a series of rungs, contacts, and coils. The solid lines making up the rungs represent the path the current will take, the contacts represent the sensors, and the coils represent the devices being energized. For example:

--------| |----------------------( )---------

Light switch Light bulb

Where ---- is a rung, --||-- is a contact (input), and --( )-- is a coil (Output). When the light switch is closed, the light bulb receives the current and turns on.

Programming with ladder logic is nonsystematic and for complex systems it can become quite convoluted and often times erroneous (Devanathan, 1991). An example of a more complex ladder logic program can be seen in Figure 8.1.

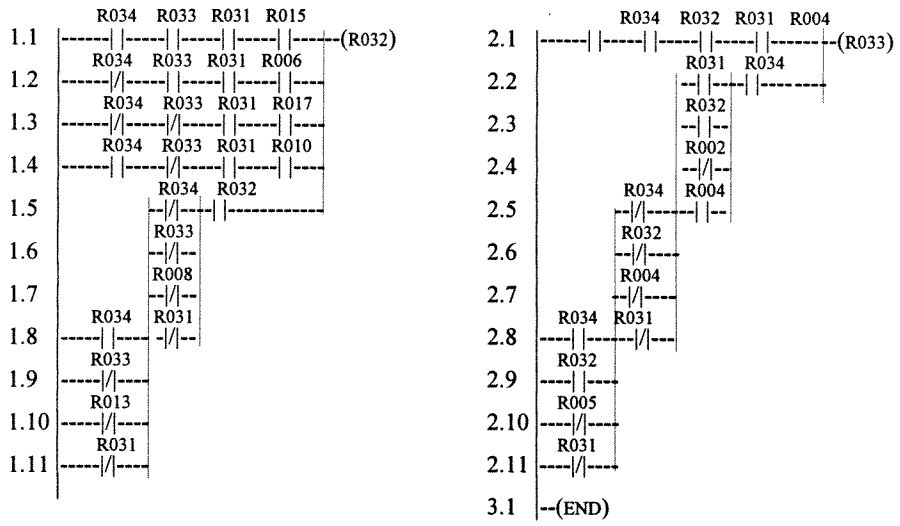


Figure 8.1 - An example of Ladder Logic (Taken from (Devanathan, 1991))

8.2 State Transition Diagrams

Another programming method used in programming FMS workcells is to use Finite State Machines (FSMs - not to be confused with FMSs) which are expressed using state transition diagrams (STDs). This method became popular in the eighties after being used previously by circuit designers in the logical design of circuits. The advantages that STDs provide to system design are in their ease of automation, ease of maintenance, and ease of use for troubleshooting (Devanathan, 1991).

State Diagrams are similar in nature to finite automaton diagrams. They are composed of nodes and arcs with associated labels. The nodes of an STD represent the possible states the system can be in. States represent a particular condition of the system, where the current state is a result of what has happened in the past. For example, an error state would be the resulting state if the system encountered an error in production. The

lines of an STD represent inter-state transitions that occur within the workcell, the directions of the transitions are denoted by arrows. Actions to be taken during a given state transition (as well as conditions that must be met for that transition) are labeled on the inter-state transition lines. See Figure 8.2 for an example of a STD.

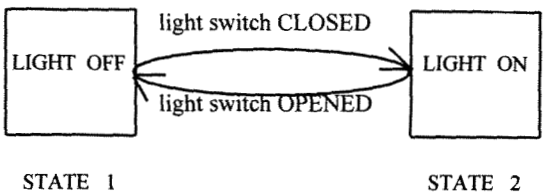


Figure 8.2 Example of a STD

One example of a state-based control language are the two software packages that work as a unit with state diagrams called PROLOC and AILISTD (Devanathan, 1991). PROLOC takes a STD and transforms it into ladder logic for use on a PLC. AILISTD is an English text based language that takes cause-effect sequence inputs from the user and converts them into State Diagrams (therefore, AILISTD can be circumvented if the state diagram has already been developed). An example of an AILISTD statement:

"**When** *device_name_I* **is**{active, not_active}, *device_name_II* **is** {active, not_active}"

A restricted grammar input format is used, parsing from left to right. Key words such as: "when" and "is" are used in determining the meaning of the statement. Logical operators are also included to enable the specification for multiple conditions to be met. Each entity of the system is assigned a variable for reference within the program.

8.3 Statemate

As STD diagrams became more popular, certain restrictions were discovered in their implementation. In an attempt to make the state diagram more useable, a new approach was investigated and discussed in (Harel, 1987) called "State-charts." State-charts were designed to deal with the problem of STDs being too:

- hard to read - need hierarchical decomposition
- difficult to draw
- unusable for stepwise refinement
- non-decomposable to usable code
- non-user friendly

State-charts combined the STD with the three concepts, "hierarchy, concurrency, and communication," which worked to, "Transform the language of state diagrams into a highly structured economical description language" (Harel, 1987).

What State-charts do in summary is: 1) Reduce the number of arrows required in designing a complex system by using clustering of nodes, 2) Represent concurrent systems by using orthogonal representations of clusters, 3) Systemize entry into clustered nodes to reduce the number of arrows, and 4) Provide hierarchical decomposition of the STD by adding zooming. These refinements reduce the confusion involved in the use of STDs, and systemize their creation, making the language of state diagrams more descriptive and precise. See Figure 8.3 for an example of a State-chart. State-charts are discussed in detail in (Harel, 1987).

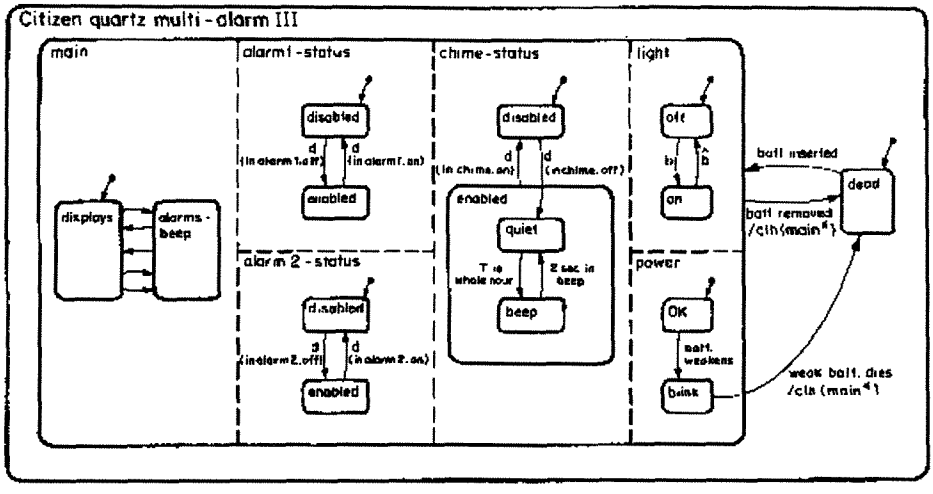


Figure 8.3 Example of a State-Chart diagram. Taken from (Harel, 1987).

Figure 8.3 is a State-chart representation for a Citizen quartz watch, where each square represents a state the system can be in. The encompassing squares show aggregation of states, a feature of State-charts. The dotted lines show orthogonality, and represent states and actions which can occur concurrently. The largest square is the state of "being" for the watch and inside of that state, the watch is either in the state of being dead or being alive, represented by the two sub-squares immediately within the outer square.

In the software package Statemate, built upon the concept of State-charts, three aspects of the system are analyzed by the programmer, and are graphically represented in State-chart form through the tools in Statemate: the structural view, the functional view, and the behavioral view (Chaar, 1990). The first view defines what components make up the system, the second view describes the hierarchy of activities the system will perform, and the third view describes how the various activities and modules of the system interact

with one another.

8.4 Petri Nets

A Petri net, named after Carl A. Petri and invented in 1962, is a graphical flow-chart-like representation used to "specify, analyze, simulate, and evaluate" the changing behavior of a given system. Petri nets allow for the modeling of concurrency, by giving a separate representation for the hierarchies of possible events that may take place and the corresponding states that may occur within each specific time frame. PN's are particularly powerful in visually presenting synchronization between events, and in detecting possible deadlocks or inadvertent exclusions of a machine or a module during the development of a FMS workcell (Chaar, 1990). Since their creation, Petri nets have been elaborated many times and many new versions have resulted: Modified PN's, Augmented PN's, Timed PN's, and Colored PN's. PN's are best for a smaller system because they tend to become convoluted as the system grows larger and more complex.

As a language, PN's are missing the required data structures to create a representation for the objects that may be connected to the system, as well as any data structures at all, and consequently are only used as a design tool rather than an inplementational tool for a FMS workcell (Rau, 1993). As a design language, they are however, especially useful and are one of the few languages that represent concurrency.

8.5 A Comparison

Relay ladder logic, as a language, is lacking in its presentation style. As the programs grow, so the complexity of the ladder grows, making a quick interpretation of the design difficult. Relay ladder logic programming is often done in a non-systematic way, and consequently is often confusing or erroneous.

The presentation style of STDs used in PROLOC and AILISTD, while an improvement over relay ladder logic, become confusing as the system gets larger. On smaller systems STDs are easy to create, easy to maintain, and easy to troubleshoot; however on larger systems an alternative is needed. One other feature STDs lack is a way to model concurrency within the system. In a STD it is the operational features of the system that are emphasized. Statemate and Petri-nets provide an alternative solution to the problems encountered with STDs, and both offer concurrency.

PN's neglect the representation of actions occurring within the system as a whole, while Statemate makes special exception to include them. Both allow for the representation of actions occurring inside the system, providing a local view of process flows. As of 1993 only Statemate included timing constructs, while PN's had submitted proposals to include them (Rau, 1993). Data structures, programming statements, decision making based on user-defined variables (such as "if . . . then" statements), and looping constructs are all included in Statemate, while PN's only include the looping constructs.

The specific advantages associated with each package are as follows:

- Relay Ladder Logic- the traditional approach
- PROLOC and AILISTD- the English textual interface, and state variable approach
- Statemate- exhaustive, graphic approach (Rau, 1993).
- Petri nets- provide for analysis capability, such as deadlock detection (Rau, 1993).

From this comparison it can be seen that a successful workcell programming language will include concurrency, simplicity of design, and the various data constructs outlined in the second section of this paper.

9. Implementation of SBCL

This section will describe the various design criteria used in the creation of SBCL. The paper will then conclude with a detailed look at the strengths and weaknesses of SBCL, analyzing them based upon the criteria developed in Section 2, as well as from an overall standpoint of functionality and usability.

9.1 Review of Problems in Old CPL

The problem that had been plaguing CPL since its debut in 1992 was the error recovery problem. The first two versions of CPL limit the control computer's ability to detect errors because it does not allow input signals unless it is ready to receive them. This implies that a programmer using the first two version of CPL needs to be able to anticipate when an error will occur, set the computer in a listen status before it occurs,

and wait for the error. Not only does the programmer have to anticipate *when* the error will occur, he or she has to anticipate *where* the error will occur in order for the control computer to listen to that specific device to determine if it is indeed occurring. While the control computer is listening for the error, no other processing can be accomplished, because the first two versions of CPL don't support concurrency. This makes effective error checking and recovery an impossibility.

A workcell language should be able to detect any and all activity coming from every component at any given time, while the system remains in total control of product flow. For example, if a signal from the pallet lift center were to come into the control computer, the system must be able to know, given the state the system is currently in, whether that signal is an appropriate one and, if not, what to do about it. A state-driven approach would allow CPL to retain a record of the past occurrences in order to properly respond to the present circumstances. A system able to accomplish this level of performance would have to be a reactive, state-driven system.

9.2 A Reactive System Based on Interrupts

In order to make CPL a reactive control system, the interpreter was changed to use an interrupt driven approach rather than the previously used polling approach. Interrupts were generated by utilizing the data acquisition board's capability of generating them in order to detect external events as they occurred. If, while in the middle of an action, the control computer detects an interrupt (an event), the interpreter

will generate a corresponding event and put the event into a queue, finish the action it is currently performing, refer to the queue, and process any pending events.

9.3 The Finite State Machine Approach

In order to make CPL retain its record of past history, FSMs were used. By using a state-based design, such as a FSM, a system can be programmed to carry out different actions for each particular event based upon its current state. The first two versions of CPL, Original CPL and Extended CPL, were sequential, i.e. a series of actions were sent in a 'list of things to do' to the control computer, which then drove the workcell. The control programs acted like a recipe, telling the central computer what to do and when. The state driven system gives the control computer a series of state/event combinations with corresponding action/new state assignments. With a state-driven system the computer is no longer forced to ignore an unexpected event, nor is it left guessing what should be done in the case of that event. As soon as the event occurs, the computer looks up in a table what corresponding action needs to be performed based upon what state the system is in. From this point it can be programmed to respond accordingly. After processing the action to be done, the computer adjusts its state by proceeding to the next state it has been programmed to proceed to. The new state would then contain its separate list of processes to do given an event, as well as the corresponding state transitions. The states, therefore, provide the record of what has happened within the system. The only way a given FSM can be in a particular state is if the computer has first

completed an action and then put the FSM into that state (see Figure 9.1).

As an example of how the *event / state - action / new state* cycle would proceed, here is a scenario that a simple workcell may encounter. In this scenario the conveyor belt is turned on by a starting event, Start_Key pressed. Then, the system waits for a pallet to pass a photocell. Given that event, the pallet stops are turned on and the system waits for the pallet to arrive at the station. When the pallet arrives at the station, it is lifted and a part is loaded. Once it has been loaded the pallet is lowered and it returns to the initial state to wait for another pallet to arrive. Figure 9.1 shows the FSM diagram as a STD given the processing cycle just described.

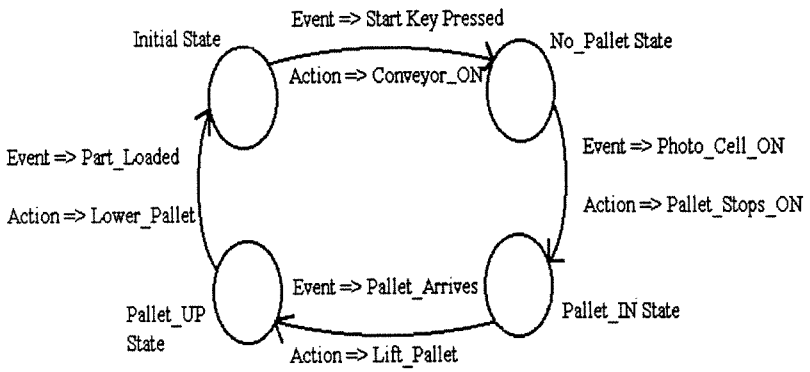


Figure 9.1 - FSM using a STD

As this workcell is in the "Pallet_IN" and is waiting for the pallet to arrive, suppose the photo cell is triggered again to signify that another pallet is about to arrive. This could be an error. The activation of the photocell will produce an event, and the control computer will then search through its table to find out what it should do, given the state "Pallet_IN." With the finite state machine in Figure 9.2, the control computer would find

that if it is in the state of having a pallet in the system, the photocell is activated, then an error event is to be generated. This error event would also have its action and appropriate new state (see Figure 9.2).

	Initial State	No_Pallet	Pallet_IN	Pallet_UP
StartKey_pressed	Act.-> Conv_ON New_st-> No_Pallet			
Photo Cell Activated		Act.-> Pal_St_ON New_st-> Pallet_IN	**Act-> Gen_Error** **New_st->Pallet_IN**	
Pallet_arrived			Act.-> Lift_Pallet New_st-> Pallet_UP	
Part_loaded				Act.-> Lower_Pall New_st-> Init_State
Error	Act.-> New_st->	Act.-> New_st->	Act.-> New_st->	Act.-> New_st->

Figure 9.2 - FSM using a State Table

9.4 State-Based, Interrupt-Driven, and Concurrent Interpreter

Given the state/interrupt-driven nature of the system, concurrency is now possible. The polling-based implementation of the previous two versions of CPL had the problem of focusing the resources of the control computers on one particular action to be processed. With the FSM approach and the use of hardware interrupts, SBCL addresses this problem by allowing the system to be reactive.

The design approach for SBCL is as follows. In SBCL the control system is

modelled using separate machines for each autonomous manufacturing operations, one for each concurrent activity. Then, when an interrupt occurs that indicates an external event, the corresponding event code associated with that interrupt is placed in an event queue. As soon as the SBCL has completed any currently in-progress action, it checks the events queue; if there is a pending event, it will then examine *each* state machine (like the one in the above scenario) in order of their creation, find the appropriate *action / new state* to be performed for that *state / event* combination, perform the action, and change the state of that machine. This method effectively creates a concurrent processing environment. Figure 9.3 sumarizes this process.

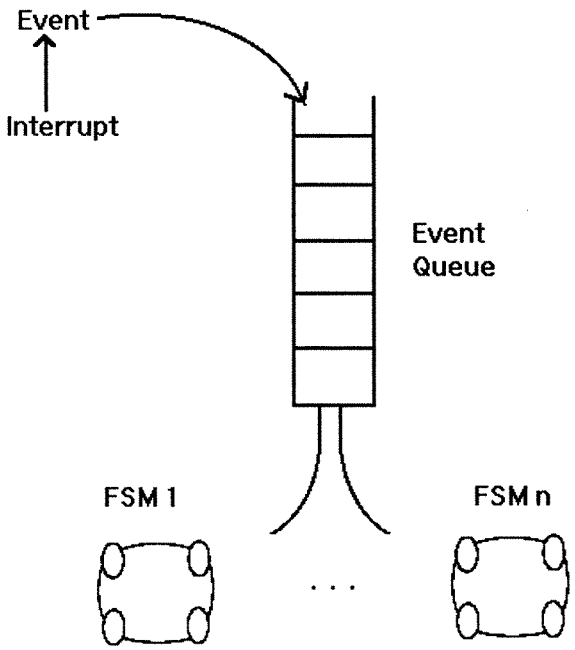


Figure 9.3 - Illustration of FSM/Interrupt Concurrent Processing in SBCL

10. Verification of SBCL

In order to verify SBCL, the interpreter and interface were built and tested on the FMS workcell in Kreger Hall, at Miami University. A description of the actual implementation is as follows.

10.1 Implementation

SBCL uses object oriented techniques, The user-interface is written in Visual Basic and the interpreter is written in C++ on an IBM-compatible PC. It consists of a user interface for accepting the description of a FSM and an interrupt driven interpreter to execute the p-code created by the interface. Interrupts are generated using the technology built into the Omega Data Acquisitions Board, and a companion program that came with the board, written for the purpose of redirecting hardware interrupts to user defined procedures. The details about the design of the interpreter as well as a description of the graphical user interface are as follows.

10.2 Interpreter Construction

SBCL was designed with four main objects used for control of flow within the program. Input to the SBCL interpreter consists of a series of numbers. The first four numbers tell the interpreter how many lines to read, as well as how many objects it is going to create for control of the flow in the workcell. The objects used by the interpreter

are as follows: 1) A double array called the event list, 2) A single array called the action list, and 3) A class called a "state machine." They are described below.

10.2.1 The Event List

The first object, a fixed length double array, is used for the purpose of storing which event to place in the event queue in response to a hardware interrupt. Each field of the array contains a number that corresponds to the corresponding event code to be placed in the event queue. The array is initialized to zero, this is done so that the programmer is not required to assign an event code to every possible interrupt, but only to those in which he or she is interested. When an interrupt occurs, the registers AX and BX are loaded with the bit number on the DAB where the interrupt occurred, as well as a logical One or Zero to show whether the signal was from an activation or deactivation of a sensor. These two values become the indices to the event list.

10.2.2 The Action List

The second object, the action list, is a list of numbered actions to be taken given an event. The action list array is made up of pointers to objects derived from the same base class, called action( ). The common function all action objects share is the execute() function for executing the designated action. When an action needs to be performed, based on an event/state combination, the field containing the pointer to the appropriate

action object is called by the state machine and told to execute( ).

10.2.3 The State Machine

The third object, the finite state machine (FSM), is an object described using a transition matrix of events and states. There can be as many FSMs created as needed for a workcell (see Figure 9.3). Each coordinate in the state chart contains two values, the first being the appropriate action to be called and the second being the appropriate new state. The current state of the machine is stored in a variable and whenever an event is in the event queue, the state/event coordinate is checked for each state machine, and if there is an action associated with that coordinate, it is executed. See Figure 10.1 for an illustration of this objects and its functions.

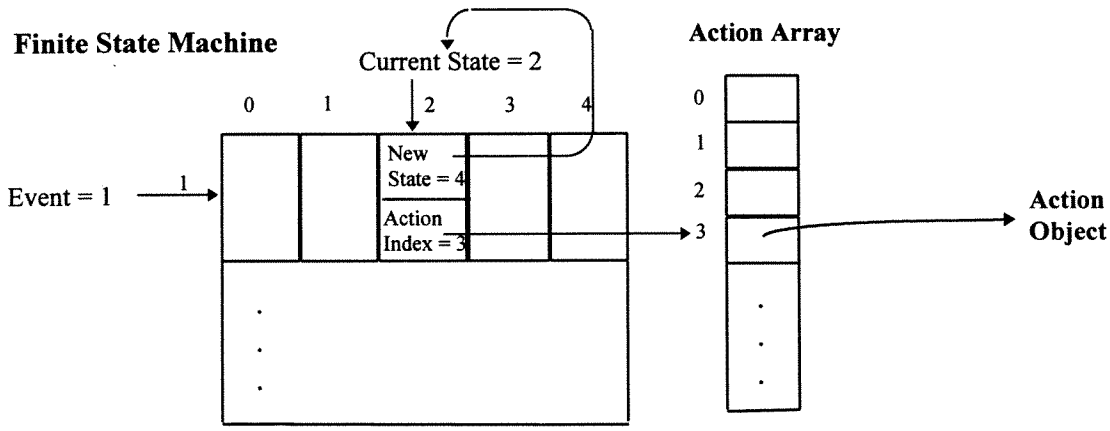


Figure 10.1 - The FSM structure and its interaction with the Action Array

10.3 Functionality of SBCL

SBCL has eight operations that can be performed by an action object. These eight operations are each assigned a unique op-code. Each op-code has associated with it certain parameters required for the action to be carried out. A list of the various operations, their op-codes, and associated parameters are shown in Table 10.1.

Op-Code	Function	Parameters
0	Do-Nothing	None
1	Send a string	Com Port, String
2	Strobe a Bit	Port #, Bit #
3	Set Bit On	Port #, Bit #
4	Set Bit Off	Port #, Bit #
5	Send an Error Message	Message to Send
6	Generate Event/Do Action	Event #, Op-Code, Operation parameters
7	Timer	Amount of time to pause (in Milliseconds)
8	Send a File	Com Port, File Name

Table 10.1 - Op-Codes, Function, and Parameters for SBCL's Action Objects

10.4 Example of SBCL P-Code

Figure 10.2 gives an example of p-code used to run the SBCL interpreter. This example includes comments for a better understanding of the numeric code; however, the SBCL interpreter has not been constructed to allow for comments.

```
5          -- The # of events to be read into the event list
6          -- How big to make the Event Q
5          -- The # of actions to be read into the action list
1          -- The number of state charts

1 7 1      -- The event number followed by the interrupt #
```

```
2 7 0                       --  note: that the interrupt # consists of the
3 6 1                       --        interrupt bit on the DAB as well as the
4 6 0                       --        logical operator 1 or 0 (for ON or OFF)
5 21 1

1 1 640 5                   -- The action number followed by the op-code,
2 5 0 Error in Pallet lift  --       followed by the parameters
3 2 640 7
4 4 640 0
5 3 640 2

3                           -- The number of states for state chart #1
5                           -- The number of event/state comb. to be input
                            --     note: all other permutations will be given the
                            --           initialized values
0 1 2 1                     -- The State, the Event, the Action, and New State
1 3 3 1
1 4 2 2
2 2 1 0
2 4 5 2
```

Figure 10.2 - Example P-code for the SBCL Interpreter

10.4 The GUI Interface

The examples in the following sections are taken from the Miami University FMS workcell located in Kreger Hall, and demonstrate the SBCL interface.

10.4.1 Declaring the Device Objects for a Workcell Using SBCL

The user is required to first build a list of the devices that are connected to the control computer. The purpose of this window is similar to the device declaration section of the two older versions of CPL. The user names the device, chooses the object type for the device (Coil, Sensor, Pulse, Programmable, and a new device type- FSM), specifies

its appropriate port, and chooses the DAB bit number if applicable. Once the appropriate details have been selected, the user then presses the Add button. The object is then added to the list on this screen as well as to a drop-down list box in the action window (see Figure 10.4). Any devices declared as sensor are added to the finite state machine as events in the state/event chart. For each FSM device declared, a corresponding FSM machine will exist for the user to fill-out with the events already present. Multiple FSMs must be declared in order for concurrent processing to exist. At least one FSM must be specified and added to the list. Figure 10.3 shows an example of a completed list of objects.

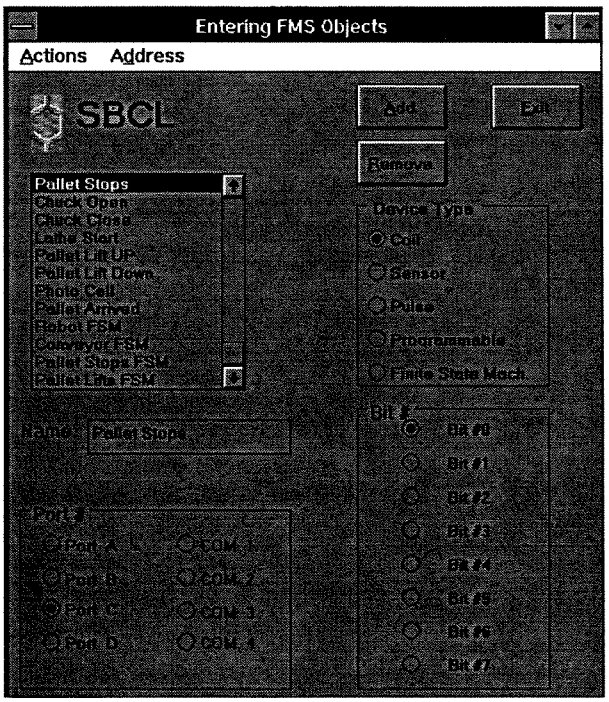


Figure 10.3 - The Object Window

10.4.2 Declaring Action Objects for a Workcell Using SBCL

The user is required to create a list of actions for the workcell to perform. The available actions correspond to the op-codes previously listed (see Table 10.1). In this window an object is to be selected from the drop-down object's list box (created by the previous window) and then an appropriate action is selected from the action list and given a name. The user then presses the Add button and the action is added to the list on this window as well as to a drop-down list box in the FSM window (see Figure 10.4 and 10.5). Parameter frames appear for the user to enter required parameters. For example, the amount of time to wait during a timer action, or the string to be sent to a programmable object. Available action choices are limited once an object is selected due to the fact that the objects only have certain functions that can be performed (see Table 4.1). Figure 10.4 gives an example of a completed action list.

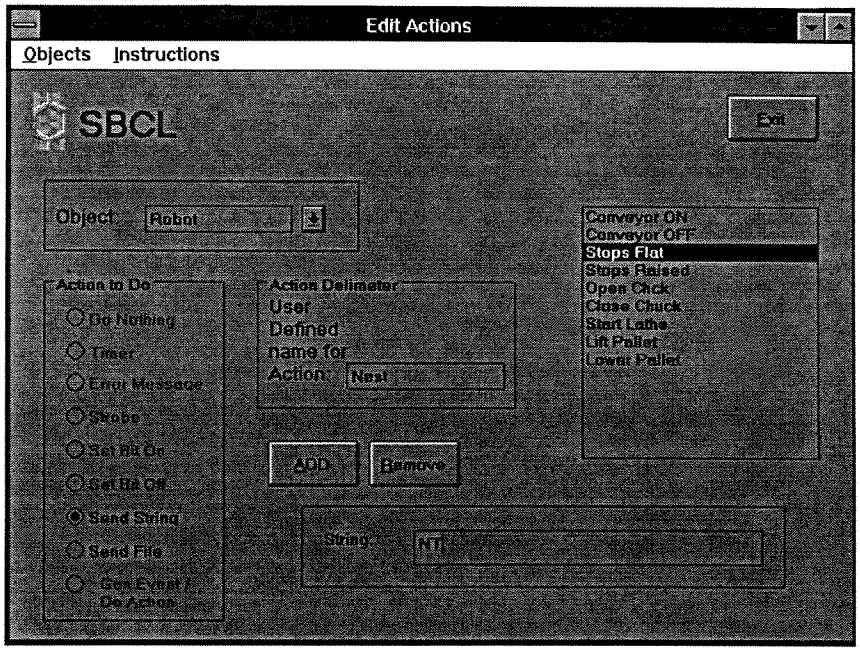


Figure 10.4 - The Action Window

10.4.3 Creating a FSM in SBCL

The user is required to build the various FSMs that have been declared in the Device Object window. From the drop-down list box of FSMs created in the device objects window on the Main screen, the name of the FSM to be specified is selected (see Figure 10.6) and the "Build a FSM" button is selected. The name of the FSM appears in the upper left-hand corner of the screen and the appropriate FSM matrix appears. At this point the matrix will be empty except for the events column, which was created previously while sensors were being declared in the device object window. States must be named and declared for the FSM, as they are added they appear in the FSM sequentially as columns. Once the appropriate states have been declared, the FSM is ready to be programmed.

Programming of the FSM is done by specifying the event - state coordinates, and selecting an appropriate new state and action for that coordinate. Once the appropriate coordinates and action/new states have been declared, the user must press the add/edit button to add it to the matrix. Figure 10.5 gives an example FSM under construction.

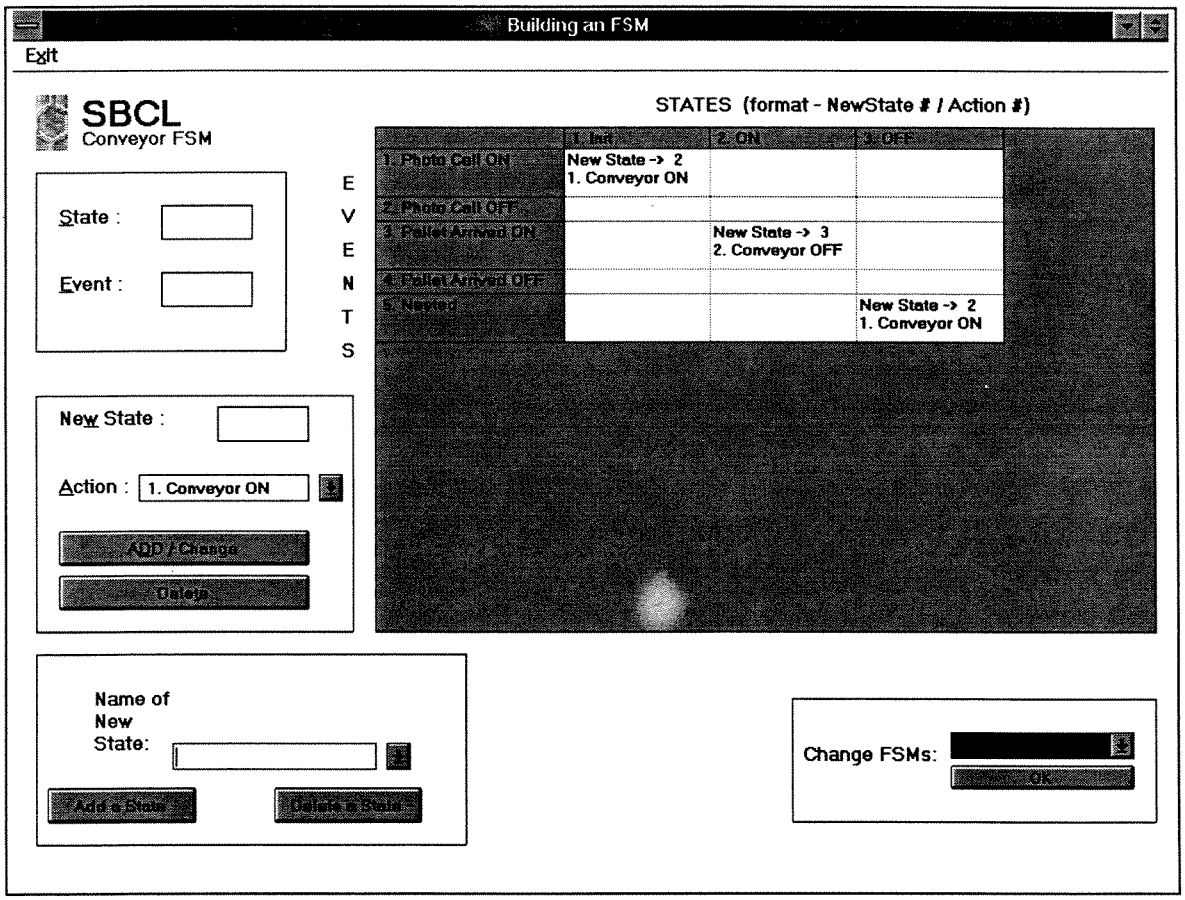


Figure 10.5 - The FSM Window

10.4.4 The Main Window

In the Main window, the user is presented with five buttons and a list of FSMs that have been created in the device objects window (at start up the list is empty). The user must navigate through the interface using this screen. The first two buttons get the user to the windows that build the device objects list and the action objects list. The third button gets the user to the FSM window, and the last two are used to generate the p-code and to run the interpreter. At the present moment the run button is not implemented. SBCL programs, like the ECPL version, must be run through an external interpreter after the p-code has been generated. The interpreter is activated by running the executable file "SBCL.EXE" and specifying the name of the p-code file. For example, typing the command: SBCL DEMO.OUT at the command prompt in DOS will execute the SBCL p-code in the file DEMO.OUT (please note that all generated SBCL code must have the ".out" extension). Figure 10.6 shows the main window for SBCL.

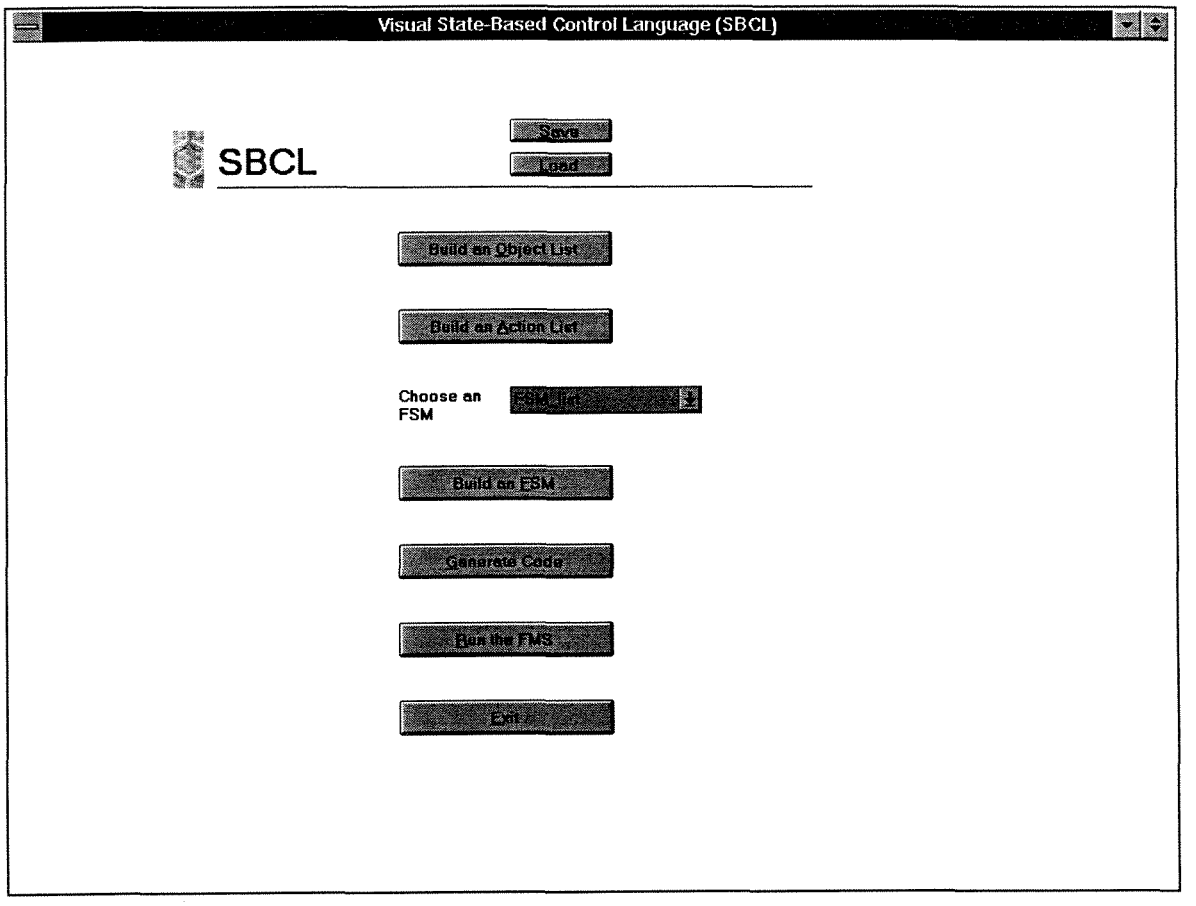


Figure 10.6 - Main Window for SBCL

10.5 Testing of SBCL

The SBCL interpreter and GUI interface have been run several times successfully in the Miami University FMS workcell located in Kreger Hall. Error routines were tested, as well as concurrent procedures. An example of a working demo for the Miami University FMS workcell is shown in Appendix A.

11. Analysis of SBCL

This section will look at SBCL from the perspective of the design criteria laid out in section two of this paper. Then SBCL will be analyzed from a usage standpoint, with an elaboration of its strengths and weaknesses.

11.1 Analysis Based on Workcell Design Criteria

1) Ability to distinguish between differing devices and their types:

In SBCL, the control computer is able to recognize an unbounded number of devices connected to it. Devices can be connected through COM ports, LPT ports, and internal ports, allowing for the addition of new objects as the workcell grows. All actions are specified and associated with an external port and a specific signal for the purpose of communicating with the various devices. The COM port specifications are made once in a separate file, comsetup.ini, just as in the extended version of CPL. In this manner communication parameters are specified once only, saving the CPL programmer from added complexity and confusion. All other communication is accomplished through the DAB.

2) The language must be able to send messages to workcell components that are controllable, and be able to allow message passing between components if necessary:

Communication between devices is accomplished through the control computer.

Strings and files with the ".cmd" extension (note: while programming an action to send a file, the ".cmd" extension does not have to be specified. For example to send the file LOADPART.CMD, the user need only write LOADPART, it is assumed that the file has been stored with a ".cmd" extension) can be sent to programmable devices connected to COM ports.

3) The language must allow the programmer to specify time-based actions.

There is a timing construct in SBCL, which will allow a pause for a specified period of time. This timing construct can be used in situations such as an inspection station where some sort of visual processing can be accomplished.

4) The language must process variable type information.

There is no explicit variable representation within SBCL.

5) The language must incorporate flow-of-control constructs such as conditional execution (if. . .then) and iterations (loops).

Conditional execution and looping are incorporated within the language of FSMs.

6) Ease of use is another important criteria.

With the graphical user interface, SBCL has become an extremely easy programming language to use.

7) The language must support concurrency.

SBCL was designed for the purpose of establishing concurrency. See Section 9 for more details.

11.2 Strengths and Weaknesses of SBCL

11.2.1 Strengths

SBCL has the following strengths:

- Graphical User Interface
- Concurrency
- State-Driven Processing
- Error Recovery
- Error Detection
- Simplicity of program design

As mentioned before SBCL is able to process two independently running sections of a workcell concurrently. This greatly increases the efficiency of a FMS.

SBCL is reactive in nature, enabling the control computer to be aware of any activity occurring within the workcell. This enhances error detection and includes the added dimension of recovering from errors, a valuable characteristic in a FMS workcell.

The programming of SBCL has been reduced to the development of a state machine. Due to the simplicity of state diagram, programming a workcell has been simplified. This saves time by removing the need for proofreading, or debugging a long

textual program (with syntax and semantic rules).

SBCL also includes the feature of allowing the user to use the keyboard to enter an event into the event Queue, which enables interaction with the user during run time.

11.2.2 Weaknesses

SBCL has lost the previous version's ability to time out while waiting for a given device to register an interrupt, and in certain situations this may be a desirable feature.

At the present time SBCL has no way to prioritize events, they are simply processed in the order in which they occur. The only event which has priority is the ESC key being pressed, which will reinitialize the system and exit the program. A priority system for differing events would further decrease the makespan of a part in high demand. For example, by prioritizing the event of a high priority part's arrival, the higher priority part could then be processed ahead of lower priority parts.

With the redesign of the compiler and interpreter, SBCL lost the variable constructs. Although events can be considered variable constructs of a sort, they do not store any data to be used in future processing. True variable constructs in SBCL could be used to implement user input, a vision system, or a bar coding device.

12. Conclusion and Areas for Future Research

By virtue of its graphical user interface, workcell programming at Miami has become a more user friendly process. By making the interpreter both interrupt and state driven, SBCL is now able to support concurrency, as well as able to recover from errors and to be more sensitive in detecting errors in comparison to CPL.

SBCL can be further validated and tested by using it to integrate the AS/RS into the Miami workcell.

Some areas of further research for SBCL are:

1. Processing of Variable constructs and the implementation of a bar coding device, a vision system, or a user interface during run time.
2. Incorporate a way to prioritize events in the system.
3. Incorporate timing-out error detection on sensor devices.
4. Develop a graphical representation of the state transition diagrams for the development of programs.
5. Incorporate Printed reports of the state charts, programmed using the compiler, or provide reports for cross referencing of events between finite state machines.
6. Incorporate simulations of the user created FSM cycles that will display what will happen during run time in the FMS workcell.

In conclusion, SBCL provides superior error detection and recovery, supports concurrency, and is much easier to use than CPL.

REFERENCES:

Apt, Krysztof R., *Logics and Models of Concurrent Systems*, Springer-Verlag, Berlin, 1985.

Brown, Frank M. (1991), "An Improved State Diagram," *Transactions on Education* vol 24, No 2, May, pp. 199-203.

Chaar, Jarir Kamel, *A Methodology for Developing Real-Time Control Software for Efficient and Dependable Manufacturing Systems*, Doctoral Dissertation for PHD in The University of Michigan Dept of Computer Science and Engineering, 1990.

Devanathan, R. (1991), "Computer Aided Design of Sequential Control Systems Using State Diagrams and AI Techniques," *Journal of Electrical and Electronics Engineering* vol 11, No 4, December, pp. 227-232.

Eilenberg Samuel, *Automata, Languages, and Machines vA&B*, Academic Press, NewYork, 1974.

Farooq, Shabi, *Software Development for Manufacturing Systems - Language and Networking Issues*, Working Paper #92-013, Systems Analysis Dept., Miami Univ., Oct., 1992.

Goos, G., Hartmanis, J., *Lecture Notes in Computer Science v224 - Current Trends in Concurrency*, Springer-Verlag, Berlin, 1986.

Groover, Mikell P., *Automation, Production Systems, and Computer Integrated Manufacturing,* Prentice-Hall, Inc., Englewood Cliffs, N.J., 1980.

Harel, David (1987), "Statecharts: A Visual Formalism for Complex Systems," *Science of Computer programming* vol 8, pp. 231-274.

Hatley, Derek J., Pirbhai, Imtiaz A., *Strategies for Real-Time System Specification*, Dorset House Publishing, New York, NY, 1987.

Harel, David, et al (1990), "STATEMATE: A Working Environment for the development of Complex Reactive Systems," *Transactions on Software Engineering* vol 16, No 4, April, pp. 403-413.

Meghamala, N., *Development of an Object-Oriented High-Level Language and Construction of an Associated Object-Oriented Compiler*, Working Paper #92-015, Systems Analysis Dept., Miami Univ., Dec, 1992.

Rau, Hsin, *Modeling Automated Manufacturing systems from Specification to Simulation*, Doctoral Dessertation for PHD in The University of California Los Angeles Dept of Mechanical Engineering, 1993.

Silver, Edward A., Peterson, Rein, *Decision Systems for Inventory Management and Production Planning*, second edition, John Wiley and Sons, New York, 1985.

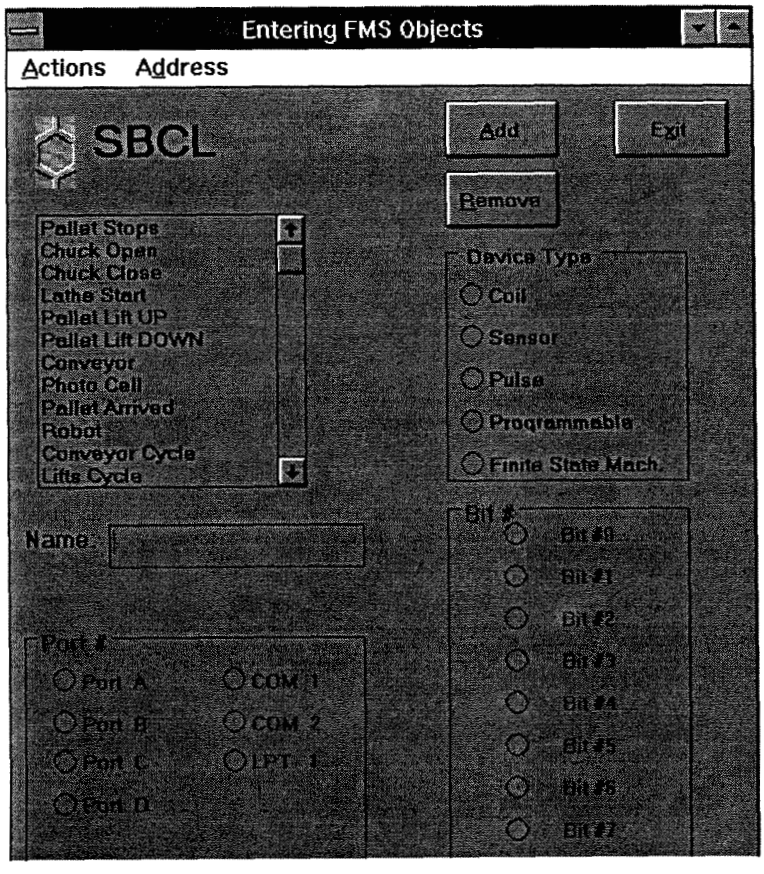
Wang, Zhuming, *Cell Programming Language: Investigation of extension for flow control and error recovery in the language*, Working Paper #94-0, Systems Analysis Dept., Miami University, November, 1994.

Ward, Paul T., Mellor Stephen J., *Structured Development of Real-Time Systems* vol 1, Yourdon Press, New York, NY, 1985.

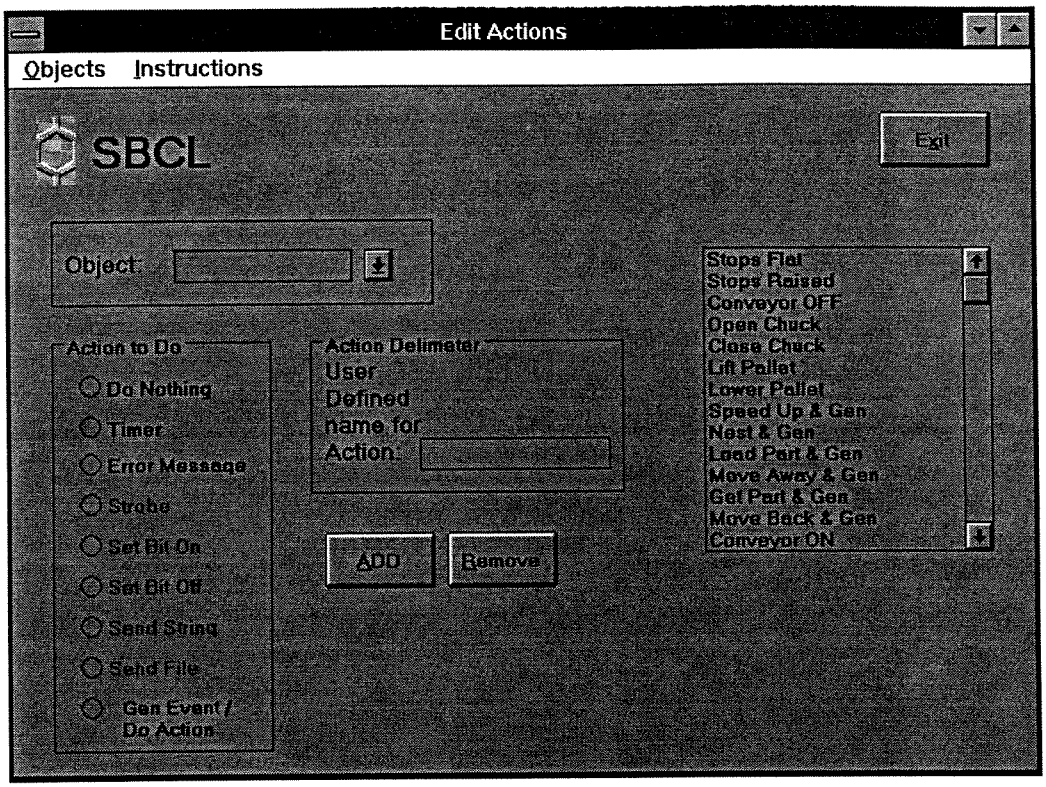
Wright, Paul Kenneth, Bourne, David Alan, *Manufacturing Intelligence*, Addison-Wesley Publ. Co., Inc., Reading, Mass., 1988.

APPENDIX A

Working Demo for Miami Universities CIM Lab FMS Workcell
Device Objects, Action Objects, and Finite State Machines



Device Objects



Action Objects