Cell Programming Language Investigation of Extensions For Flow Control and Error Recovery in the Language

Zhuming Wang
Miami University, commons-admin@lib.muohio.edu

MIAMI UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE
& SYSTEMS ANALYSIS

TECHNICAL REPORT: MU-SEAS-CSA-1994-005

Cell Programming Language Investigation of Extensions For Flow Control and Error Recovery in the Language

Zhuming Wang

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

Cell Programming Language
Investigation of extensions for flow control and error recovery in the language

by

Zhuming Wang
Systems Analysis Department
Miami University
Oxford, Ohio 45056

Working Paper #94-005 Nov. 1994

Cell Programming Language

Investigation of extensions for flow control and error recovery in the language

by
Zhuming Wang

Systems Analysis Department
Miami University
Oxford, Ohio

November, 1994

Abstract

The Cell Programming Language (CPL) is a simple programming language which allows the students and faculty to easily program a workcell on a personal computer as the workcell controller. However, the original version of CPL lacks programming flexibility and power since it does not support subroutines, flow control, error recovery and operator interface to the execution of a CPL program. The objective of the project is to design and implement a new version of CPL to overcome the shortcomings of the original CPL by extending the language to include flow control, error recovery and operator interface to the execution of a CPL program. In this report CPL is introduced. The problems in the original CPL are addressed, the extensions to the original CPL are defined, and the implementation of the extensions is described. Some future development tasks are also discussed.

1. Introduction

Today's manufacturing industries are growing rapidly. The competition within each industry marketplace has resulted in an increased demand for higher quality and lower priced products. The way to maintain the competitive edge is by implementing state of the art computing technology within the manufacturing systems [Higgins91].

A flexible manufacturing system (FMS) is a reprogrammable manufacturing system capable of producing a variety of products automatically [Chang87] . It can be considered to be a set of workcells that operate and are scheduled independently of each other [Benhabib89]. Each individual workcell is composed of one or more machine tools linked by a common material handling system and under the control of a centralized workcell controller for the purpose of producing the given requirements of a family of parts [Martin89]. The workcell controller is programmed to coordinate the interoperation of the various devices in the workcell.

Flexible manufacturing systems can be applied to provide a number of benefits and advantages over alternative methods of production: (1) Higher machine utilization; (2) Reduced work-in-progress; (3) Lower manufacturing lead time; (4) Greater flexibility in production scheduling; (5) Higher labor productivity [Groover80].

This research is an investigation of a programming language, generically called a "workcell programming language" designed specially for programming an individual workcell controller by students. The goal of this research is to provide an easy-to-use programming language for writing software for individual manufacturing workcells by developing a special purpose high-level language, called the Cell Programming Language (CPL).

Individual workcell components and their operations can be integrated by programming the workcell as a single unit. This can be performed by writing a program in a high-level language such as BASIC or C, but it requires the user to know how to perform low-level interfacing to the workcell's devices. For example, the user needs to know how to set a particular bit on a particular port to turn a particular device on or off. Instead, CPL allows the user to program the workcell by referring to devices as objects and using commands such as On/Off to turn a device on/off, and the CPL system takes care of the low-level programming details.

2. Workcell Languages

2.1 Workcells

A typical FMS workcell may consist of robots, conveyors, CNC machines, pallet stops, pallet lifts, sensors, and other devices. Devices are connected to the workcell controller computer (PCs or programmable logic controllers) through some interfacing electronics and data acquisition boards. The interfacing electronics convert signals from the PC to appropriate signals for these devices. Some devices such as robots and CNC machines are controlled by programs written in the host command languages of these machines.

An example of a workcell is in the Manufacturing Engineering Department's CIM lab at Miami University shown in Figure 1. The inputs and outputs of the workcell devices, such as the conveyor and pallet lifts, are wired through external relay interfacing to a data acquisition board in a PC which is used as the workcell controller. The robot controller is connected to either the PC's serial or parallel communication port. The CNC machine is connected to the PC's serial communication port.

2.2 Overview of Existing Workcell Programming Languages

A survey of recent literature on workcell management reveals that most research and development efforts center on specific topics such as data base applications in workcell control, knowledge-based system/software for workcell monitoring and coordination purposes, and workcell communications. There are very few research projects that address the development of workcell programming languages [Benhabib89]. They are briefly discussed in this section.

An Integrated Manufacturing Work Cell Management System [Benhabib89] represents an approach to develop "a work cell management system that functions both as an interpreter to a task-level work cell programming language, and as a work cell supervisory capable of coordinating work cell activities and implementing error recovery." Its design is based on the expert system approach that makes the process of reprogramming and system upgrading more efficient and easier.

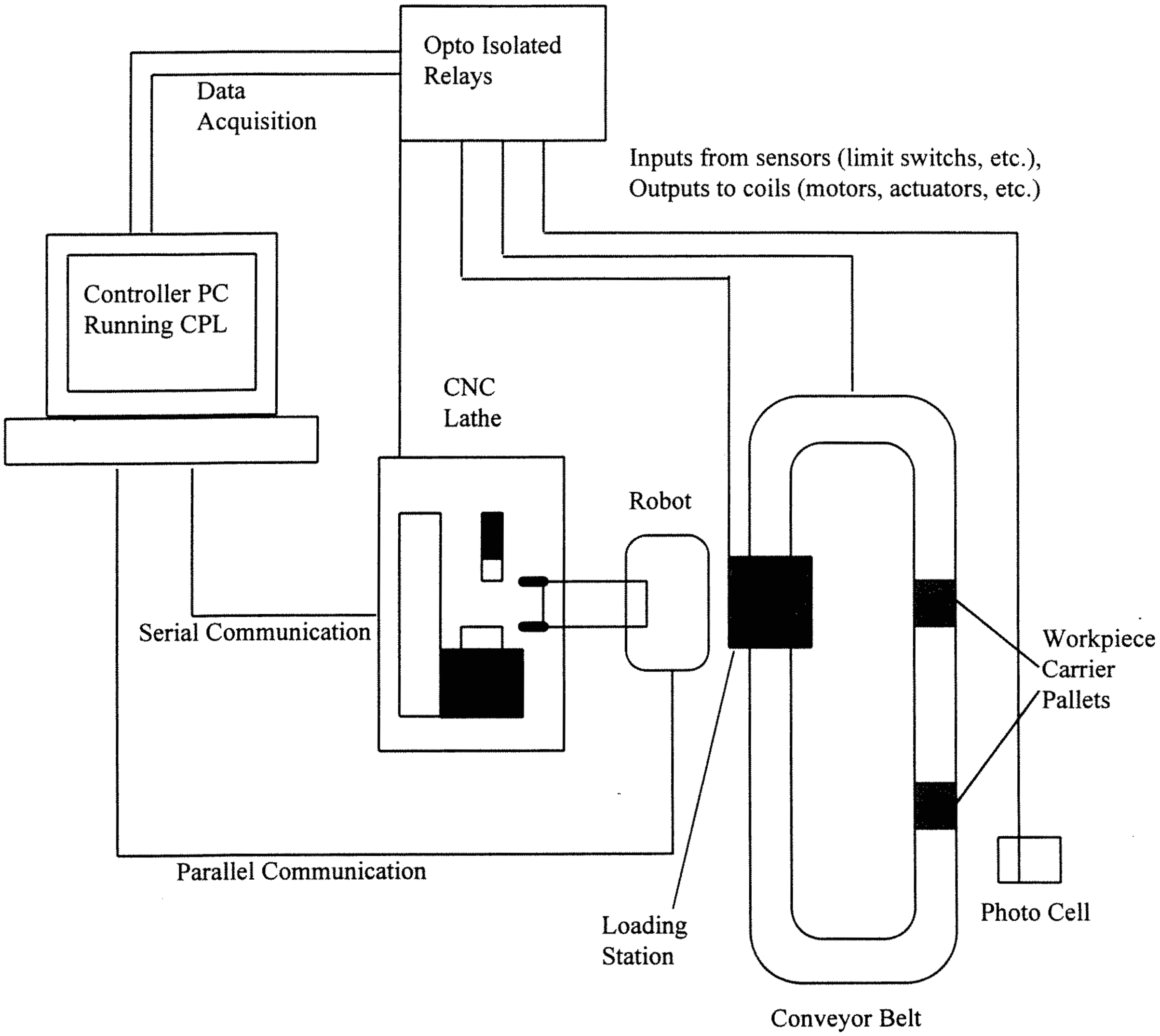
Opto Isolated Relays
Data Acquisition
Inputs from sensors (limit switchs, etc.), Outputs to coils (motors, actuators, etc.)
Controller PC Running CPL
CNC Lathe
Robot
Serial Communication
Workpiece Carrier Pallets
Parallel Communication
Photo Cell
Loading Station
Conveyor Belt

Figure 1. Diagram of the Flexible Manufacturing Cell in CIM Lab

The assembly and information management system (AIM) [Shimano88] represents an approach to robot programming which "simplifies the integration and operation of robotic assembly cells." Its design is based on the modular structure that permits customization, by modifying or enhancing the set of task-level statements, to meet different work cell functional requirements. Its decision-making capability is limited and its error recovery mechanism simple, since the system does not seek to acquire knowledge about work cell status in real time. Therefore, it cannot deal with error events without operator intervention.

The task description language (TDL) [Alder88] system follows an approach similar to that of AIM. It is developed to "enable accurate simulation of robotic work cells and the generation of programming codes to be loaded directly to equipment controllers on the factory floor." Compared with AIM, TDL is more versatile in the sense that it allows work cell programming rather than solely robot programming. TDL, as AIM, lacks the sophistication to handle error events in real time without operator assistance.

In [Benhabib89], the researchers concluded that a major weakness of existing workcell languages was a lack of error recovery. Most approaches deal with specific issues concerning workcell management, but do not encompass all three aspects of programming, coordination, and error recovery.

The Cell Programming Language (CPL) is an object-based workcell programming language developed at Miami University for use by students in the Manufacturing Engineering Department. This language is described in the next section.

3. History of CPL System

CPL was developed as a simple programming language that would allow a student to program a workcell by using a personal computer as the controller. Devices in the cell were connected to the PC using appropriate relays and a data acquisition board in the PC. To turn a device in the workcell on, software in the PC simply needs to set the appropriate bit in one of the data acquisition board's registers. To examine the state of a device, the software simply reads a register from the board and examines the state of the appropriate bit. Thus, each digital device in the FMS cell is assigned a unique bit in a register on the data acquisition board.

To control other devices, such as robots or CNC machines, the user writes a program in the language of the particular device and uploads that program, using the PC's serial or parallel port, to the device.

Before CPL was developed, students wrote BASIC programs or assembly language programs to control the workcell. Only a few students could learn the necessary low-level details needed for this kind of programming.

The first version of CPL system consisted of three parts: the CPL language, the compiler and the interpreter. The CPL language allows the user to describe the sequence of steps required to manufacture a part in a FMS workcell. The compiler translates the CPL source code into intermediate code which is called p-code. The interpreter takes p-code as input and executes it to control the operation of the devices in a FMS workcell by writing and reading binary data to/from a data acquisition board and the serial and parallel communication port in the PC. The CPL software architecture is shown in Figure 2 [Troy92]. The CPL language, the compiler, and the interpreter are described below.

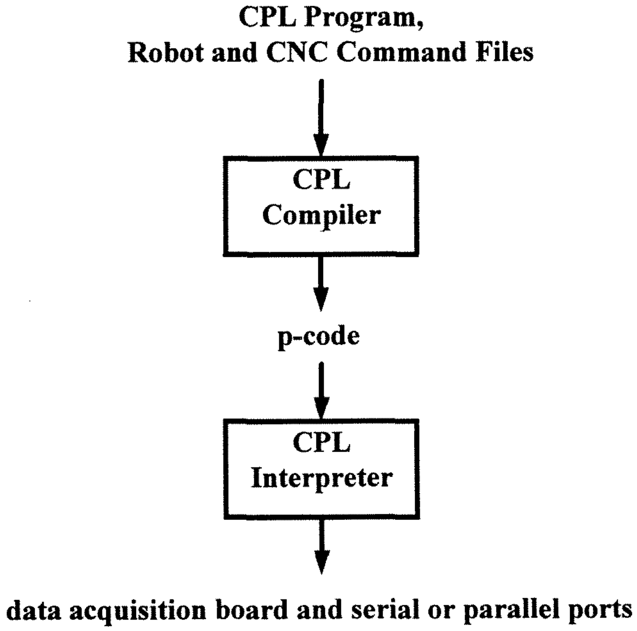
CPL Program,
Robot and CNC Command Files
CPL
Compiler
p-code
CPL
Interpreter
data acquisition board and serial or parallel ports

Figure 2. CPL Software Architecture

3.1 Description of Original CPL Language

In the original CPL, which is designed and implemented by Meghamala [Meghamala92] and Farooq [Farooq92], a program consists of three sections: port declaration, device declaration, and procedure declaration.

3.1.1 Port Declaration

The port declaration section is used to assign physical I/O port addresses to registers on the data acquisition board in the PC and to define data flow direction (input/output) of each register. In CPL these addresses are given port names for later reference. Ports could also be assigned to serial (COM) ports or parallel (LPT) ports. The declaration of ports are made within a Ports...End block.

The syntax for port declaration is as follows:

<port_identifier> <port_address> {<data_flow_direction>|<parameters>}

The port_identifier can be any user defined name consisting of alphabetic characters, digits and underscores up to a maximum of 30 characters. The port_address should be a physical port address in the data acquisition board and the data_flow_direction is either Input or Output depending on whether the port is used to send or receive signals. The parameters are used with COM ports. The syntax is <baud rate> <data bits> <stop bits> <parity>. An example of a port declaration section is as follows:

```
Ports
    PortA    640    Input;
    PortB    641    Output;
    PortC    642    Output;
    Com1Port    COM1  300 7 2 0;
End
```

3.1.2 Device Declaration

The device declaration section is used to declare device objects and associate a port and a bit number with each device object. The device types are predefined in the language. The device declarations are made within the Devices...End block. The following is the syntax for a device declaration:

<device_identifier> <device_type> [<port_identifier> <bit_number>]|[<programmable_port>]

The device_identifier is a user defined name and the device_type is a keyword in the CPL language. The device types are shown in the Table 1. A coil is a device type that can be turned on or off to control things like motors. A sensor is a device that capture signals and send signals to the controller, for example a limit switch or photocell. A pulse device type can be sent a short on/off signal to start it. A programmable device type is a device which can be programmed in its host language, such as robot and CNC machine. The port_identifier should be defined in the port declaration section as discussed previously, and the bit_number is a constant between 0 and 7 and corresponds to a bit on the data acquisition board. For the programmable device type, the port name LPT1 or the port identifier defined in the port declaration section for COM1 or COM2, would be specified depending on the communication port to which it is connected. Port identifiers are used for devices that interface through the data acquisition board, whereas programmable ports use the serial or parallel ports.

DEVICE TYPES	VALID FUNCTIONS
Coil	On / Off
Sensor	WaitOn / WaitOff
Pulse	Strobe
Programmable	Send / Do
Wait	Milliseconds

Table 1

3.1.3 Procedure Declaration

The last section in CPL is the procedure section. A procedure section consists of control statements. Each statement represents one device operation and directly corresponds to an actual operation of the real device in the FMS workcell. There is only one procedure section in the original CPL program and all statements are executed in sequence. There are no control constructs such as loops or conditions and no subroutines. The syntax of a procedure statement is as follows:

<device_identifier>.{<function>[(parameter)]}|<delay_time>

The device_identifier is an identifier previously declared in the device section. The device_function is a key word, shown in Table 1, and predefined in the language.

The procedure section is enclosed in the keywords Procedure...End block. An example of a procedure section is given below.

```
Procedure
    Conveyor.On;
    Robot.Send("NT");
    PhotoCell.WaitOn;
    ChuckOpen.Strobe;
    Delay.1000;
    Conveyor.Off;
End
```

3.1.4 Example of the Original CPL Program

An example of the original CPL program is as follows:

```
Ports
        ComPort COM1   300  7  2  0;
        PortA   64256  Input;
        PortB   64257  Output;
        PortC   64259  Output;
End
Devices
        Conveyor        Coil   PortC 5;
        PhotoCell       Sensor  PortA 7;
        Robot           Programmable  LPT1;
        Lathe           Programmable  ComPort;
        Delay           Wait;
End
Procedure
        Lathe.Do(loadlath)
        Robot.Send("NT");
        Robot.Do(loadpart);
        Conveyor.On;
        Delay.500;
        Conveyor.Off;
End
```

3.2 Implementation of the Original CPL Language

The CPL system consists of two programs: a compiler and an interpreter. The compiler parses the source code, analyzes it and generates intermediate code (p-code) which is the input to the interpreter. The interpreter executes the p-code and performs low-level I/O interface to the data acquisition board and PC's serial and parallel communication ports.

The CPL system is based on the object-oriented design and implemented in Borland C++. The ports, the devices and procedure statements are considered as objects in the implementation.

In the CPL system, the PC's serial communication ports and their configurations as discussed previously are defined in the port declaration section by the user. This requires that the user know the PC's serial communication port configuration parameters, such as transmission speed, parity, etc.

3.3 Weakness of the Original CPL System

CPL can not handle error events since it does not incorporate any error detection or recovery strategies. When a device operation fails, the execution of the program could result in a unpredictable state. For example, to examine the state of sensor type devices, the software waits to read appropriate data from the data acquisition board. If a given device is never turned on/off, the program will be put on hold and wait forever. Thus, there is no timeout function in CPL language.

In CPL, there are no flow control features such as looping and conditional execution. All statements in the procedure section are executed in sequence. This limits the use of the language for student projects.

Also, there is no provision for an operator (user) interface to a CPL program. For example, the user can not control the execution of the program from the PC (once it is started), nor can the user enter data to a running CPL program or display output messages to the operator. Additionally, there is no support for variables in which data can be stored.

Another weakness is that there are no subroutines, so CPL programs can not be written in modular fashion. There is only one procedure in a CPL program. All statements (i.e. device operations) are carried out sequentially. Therefore, to repeat a certain set of device operations more than once, one has to duplicate the same code as many times as needed in the program. This is very inconvenient and makes it impossible to write a sophisticated CPL program.

Finally, the user is required to know the serial communication parameters because these are included in every CPL program that uses serial communication ports.

4. Extended CPL

Without flow control, error recovery, the operator interface to a CPL program, and subroutines, the original CPL lacks flexibility and power to write complex and sophisticated programs. A new version of the CPL language is developed to overcome the shortcomings of the original CPL by extending the language to include flow control, conditional execution, error recovery, an operator interface, string variables, subroutines, and a main program section. Also, a serial communication configuration file is added so that these parameters need only to be configured one time, instead of coded in every CPL program.

4.1 Flow Control and Conditional Execution

Extended CPL provides several ways for conditional branching and looping by adding the following control constructs: If, While, Until, and For.

The **If** control construct executes a procedure if a condition is true. An example of IF statement is:

ProcA.Run(If PartType.EQ("A"));

Where ProcA is a name of procedure which is defined earlier in the procedure section and PartType is a string variable defined in the devices section. The procedure ProcA will be executed once if the condition (PartType is equal to "A") is true.

The **While** loop executes a procedure repeatedly while a condition is true. The Following is an example of While statement:

ProcA.Run(While PartType.EQ("A"));

This statement will execute the procedure ProcA repeatedly while the condition (PartType is equal to "A") is true.

The **Until** loop executes a procedure repeatedly until the specified condition becomes true. The procedure will be executed at least once. An example of Until statement is shown as follow:

ProcA.Run(Until PartType.NE("A"));

This statement executes the procedure ProcA repeatedly until the condition (PartType is not equal to "A") becomes true.

The **For** loop executes a procedure as many times as specified. An example of For loop is:

```
ProcA.Run(For 10);
```

Where 10 is the specified execution times.

4.2 Error Recovery

Extended CPL supports error recovery strategies. One error recovery strategy is to have timeout checking for operations of sensor devices. When a sensor type device is not being switched on/off in the specified time period, the program is terminated and the error message is displayed to the operator. Another error recovery strategy is to associate an error handler procedure to an operation statement or a procedure. Once the execution of an operation or procedure fails, the error handler procedure associated with it is called to carry out some actions to handle error. The following is an example of error recovery statement:

```
PhotoCell.WaitOn(1000):ErrHdl;
```

This statement waits for the device PhotoCell to be switched on for a maximum of 1000 milliseconds. If waiting time expired and PhotoCell is still not switched on, the procedure ErrHdl, which is defined in the procedure section earlier, is executed to handle error event.

4.3 Operator Interface to CPL Program

In the extended CPL language, there is provision for an operator to interface to a CPL program. Two new device types are added to the CPL language: String and IO.

The String type allows the operator to define string variables. Operations on strings are assignment and comparison. String variables can be used to hold the strings entered by the operator. The additional implementation of relational operators makes conditional execution and flow control possible.

The other type is called IO. IO devices are used to get user input from the keyboard and to display output messages on the screen. There are three kinds of operations for the IO device. The Get operation will wait for an operator to enter data from keyboard, then assign the data to a specified string variable. The Poll operation is used to wait for a key stroke. The Put operation displays a message on the screen. IO devices types do not require a port name.

The following are examples of String and IO device declaration statements and operation statements:

```
Devices
        PartType        String;
        Console         IO;
End

Program
        Console.Put("Please enter part type(A/B): ");
        Console.Get(PartType);
        ProcA.Run(If PartType.EQ("A"));
        Console.Put("Press any key to stop ProcB");
        ProcB.Run(Until Console.Poll());
End
```

The first statement will put the message enclosed within quotes on the screen. The second statement will wait for the operator to input data from keyboard and then put it into the string variable PartType. The last statement executes procedure ProcB repeatedly until the operator presses a key.

4.4 Subroutines and Program

In the extended CPL, it is possible for a program to include any number of subroutines (i.e. procedures in CPL), each of which consists of device control statements as well as a "main" program. A procedure must be defined before it can be called in another procedure or in the program section. Each procedure should have a user defined procedure name for later reference. Procedures can not be nested, nor can they have their own port or device declarations, i.e., all ports and devices defined in the port declaration section and device declaration section are global in the CPL program. A new section, the program section, is added to the extended CPL language. The program section consists of device control statements and/or calls to procedures defined in an earlier procedure section. The main program is coded in the program section. The following is an example of procedures declaration and program declaration in the extended CPL:

```
Procedure ProcA
        Robot.Do(ldPartA);
        Lathe.Do(mkPartA);
        Robot.Do(mvPartA);
End

Procedure ProcB
```

```
        Robot.Do(ldPartB);
        Lathe.Do(mkPartB);
        Robot.Do(mvPartB);
End

Procedure Init
        Conveyor.Off;
        Robot.Send("RS");
End

Program
        Init.Run;
        Conveyor.On;
        PhotoCell.WaitOn(1000);
        Conveyor.Off;
        Console.Put("Enter part type(A/B): ");
        Console.Get(PartType);
        ProcA.Run(If PartType.EQ("A"));
        ProcB.Run(For 10);
End
```

4.5 Other Modifications

In the extended CPL language, serial communication port configuration parameters are defined in a separate file, called COMSETUP.INI which is a DOS text file used to specify the serial communication ports configuration parameters. The format of a configuration file is as following:

```
ComPort = COM1
BaudRate = 300
Parity = NONE
DataBit = 7
StopBit = 1
```

The statements can be written in any order. This provides an easy-to-understand way for users to setup serial communication ports configuration parameters once instead of in every CPL program as was required in the original CPL language. Another advantage of having separate serial ports configuration file is the ports configurations can be changed without changing the CPL program.

4.6 Grammar of the Extended CPL

The grammar for the extended CPL language is shown below.

```
<CPL_Program>           :=    <ports_declarations> <device_declarations> [<procedure_section>...]
                              <program_section>
<port_declaration>      :=    Ports <port_stmtlist> End
<device_declaration>    :=    Devices <device_stmtlist> End
<procedure_section>     :=    Procedure <procedure_name> <procedure_stmtlist> End
<program_section>       :=    Program <procedure_stmtlist> End
<port_stmtlist>         :=    <port_stmt> [<port_stmtlist>]
<port_stmt>             :=    <port_name> <port_address> <direction>;
<port_name>             :=    <identifier>
<port_address>          :=    <integer>
<direction>             :=    Input | Output
<device_stmtlist>       :=    <device_stmt> [device_stmtlist]
<device_stmt>           :=    <device_name> <device_type> [{<port_name> <bit_number>}|
                              {<predefined_port>}];
<device_name>           :=    <identifier>
<device_type>           :=    Coil | Sensor | Pulse | Programmable | Wait | IO | String
<bit_number>            :=    0 | 1 | 2 | 3 | 4 | 5 | 6 | 7
<predefined_port>       :=    LPT1 | COM1 | COM2
<procedure_name>        :=    <identifier>
<procedure_stmtlist>    :=    <procedure_stmt> [<procedure_stmtlist>...]
<procedure_stmt>        :=    <device_name>.<device_func>|<procedure_name>.Run(<ctrl_condition>);
<device_func>           :=    <coil_func>|<sensor_func>|<pulse_func>|<wait_time>|<programmable_func>|
                              <io_func>
<coil_func>             :=    On | Off
<sensor_func>           :=    WaitOn | WaitOff
<pulse_func>            :=    Strobe
<wait_time>             :=    <integer>
<programmable_func>     :=    Do | Send(<parameter>)
<parameter>             :=    <string>|<file_name>
<file_name>             :=    <identifier>
<io_func>               :=    Get(<var_name>) | Put(<string>) | Poll
<var_name>              :=    <identifier>
<ctrl_condition>        :=    For <integer>|{ If, While, Until} <condition>
<condition>             :=    <string_name>.<string_operation>(<string>)
<string_operation>      :=    EQ | NE | GT | LT | GE | LE
<identifier>            :=    <letter>[<letter>|<digit>...]
<string>                :=    “<ascii_character>“
```

5. Verification of Extended CPL

To verify the design of extended CPL, the new language constructs were implemented and tested in the compiler and interpreter. The implementation is described below.

5.1 Overview of Implementation

The Cell Programming Language (CPL) is designed and implemented using object-oriented techniques, written in Borland C++. It consists of a compiler and an interpreter. The compiler parses the CPL source code, analyzes it and generates intermediate code (called p-code). The interpreter takes p-code as input and executes the CPL program. The details about implementing the compiler and the interpreter are described in next sections.

5.2 Compiler Construction (written in BorlandC++)

In the design of the compiler, the entire compilation process is considered to be an object. Objects at the top level of abstraction include port objects, device objects, procedure objects and program object. At the next level of abstraction, a generic statement class serves to define the common properties of all statements in procedures and the program. Derived from the generic statement class are a variety of device statement classes for each different type of device. A token class is used to define the attributes and methods of tokens. The class hierarchy is shown in Figure 3.

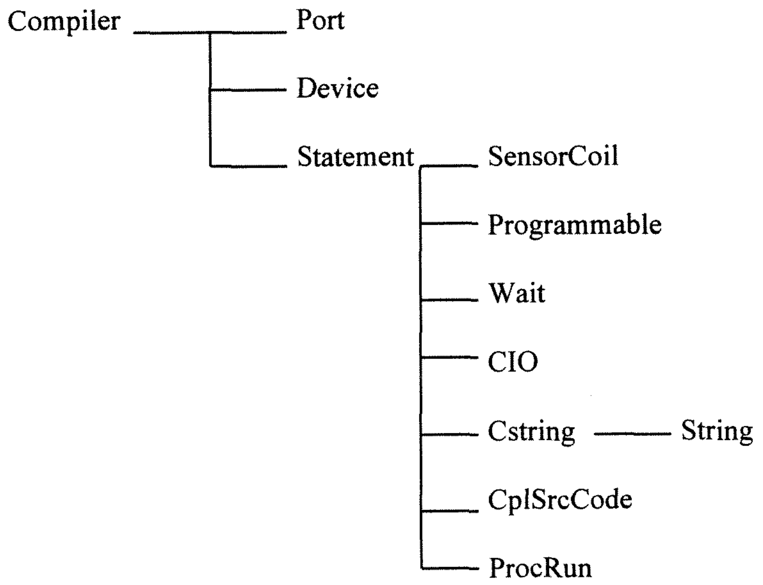

Figure 3. Class Hierarchy

The compiler is invoked with a CPL program which is instantiated as a compilation object starting the translation process. The compilation flow is shown in Figure 4.

The compilation starts with the Ports definition section. A new port object is created for every port variable declaration. Control is passed to a newly created port object to parse the port declaration itself. After the parsing, the compilation object regains control and adds the pointer, pointing to the newly created port object, to a linked list. Duplicate port names are not permitted and are flagged as an error. This linked list of port objects is used in device parsing to make sure a device is not assigned to an undefined port. The control is being passed back and forth between the compilation and a port object until all port declarations have been parsed.

The compilation next parses the Devices definition section in a similar way as with the Ports section. A linked list of pointers to all device objects is also created. Duplicate device names are not allowed. References to port names are looked up in the list of ports. The list of devices is used in parsing procedures and the program to ensure that statements reference valid device.

The compiling object now parses the procedure sections. A new procedure object is created for each procedure definition in the CPL program. Control is passed to the newly created procedure object to parse all statements in it. For each statement, a new statement object is created and the statement is parsed. Device names are looked up in the list of devices to ensure that they have been declared. As a result, a double linked list of pointers to all the statements in the procedure is created. Control is then returned to the compilation object and the newly created procedure object is added to a linked list. When all the procedure definition sections are parsed, the compilation continues to parse the Program section. It creates a program object and then passes the control to the program object. The program object parses every statement in it. A new statement object is created for each statement and the control is passed to the statement object. After the statement has parsed itself, control is returned to the program object and a pointer to the statement object is added to a double linked list. Control is being passed between the program object and a statement object until all statements in the program have been parsed as shown in Figure 4.

Finally, if the entire CPL program has been parsed successfully, the compilation object begins to generate the p-code for each procedure and then the program. Annotated p-code is shown in appendix A.

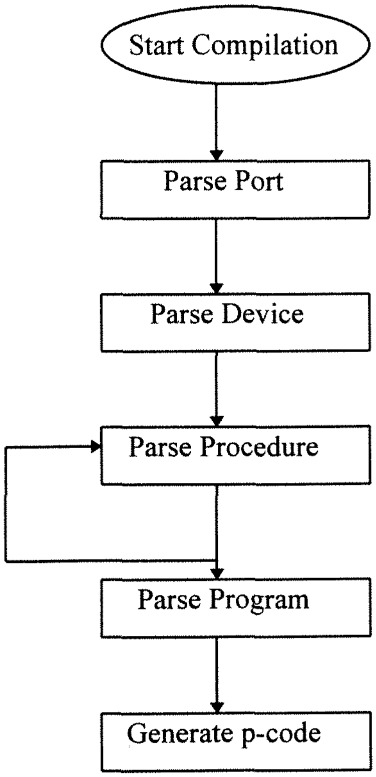

Figure 4. Compilation Flow Chart

5.3 Example of Extended CPL Program

An example of an Extended CPL source program is given in this section. The p-code for the following example is given in appendix A.

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

Devices
        PalletLiftUp    Pulse   PortC  4;
        Conveyor        Coil    PortC  5;
        PhotoCell       Sensor  PortA  7;
        PalletArrived   Sensor  PortA  6;
        ChuckOpen       Pulse   PortC  1;
        LatheG66inp     Pulse   PortB  1;
        Robot           Programmable   COM2;
        Lathe           Programmable   COM1;
        LatheStart      Pulse   PortC  2;
        LatheStop      Sensor  PortA  4;
        PalletLifted   Sensor  PortA  5;
        PalletStops    Coil    PortC  0;
        ChuckClose     Pulse   PortC  3;
        PalletLiftDown Pulse   PortC  6;
        LatheRunning   Sensor  PortA  2;
        LatheHandShk   Pulse   PortB   0;
        Delay          Wait;
        Console        IO;
        PartType       String;
        Var            String;
End

Procedure Init
        LatheHandShk.Strobe;
        LatheG66inp.Strobe;
        PalletStops.On;
        Robot.Send("NT");
End

Procedure Test1
        Lathe.Do(ldLatheA);
        Robot.Do(ldPartA);
        PalletStops.On;
End

Procedure Test2
        Lathe.Do(ldLatheB);
        Robot.Do(ldPartB);
        PalletStops.On;
End

Procedure Test3
        Console.Put("Please enter part type(A/B)");
        Console.Get(PartType);
        Test1.Run(If PartType.EQ("A"));
        Test2.Run(If PartType.EQ("B"));
End

Procedure ErrHdl
        Console.Put("Run time error, Please check the program.");
        Conveyor.Off;
        Robot.Send("NT");
End

Procedure Test4
        Init.Run;
        Test3.Run;
        Conveyor.On;
        PhotoCell.WaitOn(5000);
```

```
            PalletStops.Off;
            PalletArrived.WaitOn(5000):ErrHdl;
            Delay.1000;
            PalletLiftUp.Strobe;
            PalletLifted.WaitOn(5500);
            Conveyor.Off;
            ChuckOpen.Strobe;
            Delay.1000;
            ChuckClose.Strobe;
            Delay.2000;
            Robot.Do(moveaway);
            Delay.2000;
            LatheStart.Strobe;
            LatheStop.WaitOn(5000):ErrHdl;
            Robot.Do(moveback);
            Delay.2000;
            ChuckOpen.Strobe;
            Delay.2000;
            Robot.Do(getpart);
            PalletStops.On;
            PalletLiftDown.Strobe;
            Conveyor.On;
            Delay.500;
            Conveyor.Off;
            PalletStops.Off;
            LatheStart.Strobe;
            LatheHandShk.Strobe;
    End

    Program
            Test4.Run(For 2);
            Console.Put("Press any key to stop...");
            Test4.Run(Until Console.Poll());
            Console.Get(Var);
            Test1.Run(While Var.NE("N"));
    End
```

5.4 Interpreter Construction

The interpreter creates a procedure table to hold the information about the procedures. It now starts to read and analyze the p-code of each procedure. Every time a new procedure is encountered, the interpreter saves the procedure name and executes a loop to read each line of p-code. By checking the opcode of each line of p-code the interpreter knows if each line is source code or instruction. (Note: the source code is included in the p-code for trace debug output.) Accordingly it creates a new source code object or instruction object and attaches the pointer, pointing to the newly created object, to the list of pointers to the source code objects or the list of pointers to the instruction objects for the procedure. When the interpreter finishes reading and analyzing the p-code of all the procedures, it continues to read and analyze the p-code of the program in a similar way. After all the p-code of procedures and the program have been read and analyzed, the interpreter begins to execute each instruction by calling a member function execute() of each instruction object.

5.5 Verification

The new statements in the Extended CPL have been successfully implemented and tested. Variations of each statement were tested, as well as complete programs. The example program given in the section 5.3 is tested by executing it on the FMS cell in the CIM lab successfully.

6. Strengths and Weaknesses of the Extended CPL

In this section, we briefly review strengths and weaknesses of the extended CPL. In the next section, future development for implementing a complete CPL language will be discussed.

The extended CPL has many enhancements and improvements over the original CPL. It incorporates flow control features such as conditional procedure execution and looping and allows subroutines. It also supports error recovery strategies and provides for operator interface to CPL programs. It separates the setup of serial communication ports from CPL programs that use serial ports by defining the port configuration parameters in a text file. All these not only make the extended CPL much simpler and more useful, but it also provides additional programming flexibility and power.

Currently, the CPL compiler does not generate compiling reports. Thus, it lacks good diagnostic reports of syntax errors and cross reference reports. Also, there is some limitation on the flexibility of conditional branching and flow control. The extended CPL neither has logical operators such as NOT, AND and OR nor has mathematical operators to be used as counters, for example. Another important functionality which is not implemented in the extended CPL is parallel execution in procedures.

7. Conclusion and Future Development

The extensions to CPL have made the language more powerful and supports development of more complicated control programs. The addition of procedures support module programming. Also, error recovery is now incorporated into CPL using the timeout mechanism and procedures. Conditional execution of procedures is supported. Lastly, a simple operator interface has been added.

To implement a complete CPL language, future development could include the following:

1. Generating compilation reports such as cross references and robusting diagnostics of syntax errors;
2. Implementing logical operators such as NOT, OR, AND, numeric data type variables, and mathematical operators like counters to enhance the flexibility of conditional branching and conditional flow control;
3. Implementing the functionality of parallel execution of statements in procedures and the program to enable the integration of another part of FMS cell in the CIM lab.;
4. Constructing a better user interface. Ideally, a graphic user interface should be implemented to provide an integrated user interface. An editor and a full-fledged debugger could be provided with the compiler. Also, an integrated programming environment to different machine programming languages which are needed in a complete CPL project, such as RWARE for robot programming and SmartCAM for CNC machine programming, should be provided to enhance the consistency [Marcelo94];
5. Providing ability to create the Port Declaration section and the Device Declaration section in a separate file so that it can be reused. This will eliminate the need for the CPL programmer to memorize port address and bit number for each individual device. This would reduce the hardware dependency of CPL.

APPENDIX A: P-code for the CPL program shown in section 5.3

```
6                                       /number of procedures
4444 Init                               /start of procedure Init
10    LatheHandShk.Strobe;              /source code
7 641 0                                 /strobe bit 0 on port 641(PortB)
10    LatheG66inp.Strobe;
7 641 1                                 /strobe bit 1 on port 641
10    PalletStops.On;
1 642 0                                 /set bit 0 on on port 642 (PortC)
10    Robot.Send("NT");
8 21 NT                                 /send string “NT” to port COM2
5555                                    /end of procedure Init
4444 Test1                              /start of procedure Test1
10    Lathe.Do(ldLatheA);
8 20 %                                  /send lathe program “ldLatheA” to port COM1
8 20 N' G'  X '   Z '  F'  H
8 20 00M03
8 20 01 00    00 - 7100
8 20 02 01 - 100     00  80
8 20 03 01 -  50     50  25
8 20 04 01    00    500  25
8 20 05 01    50     50  25
8 20 06 00   100     00
8 20 07 00    00   6500
8 20 08M05
8 20 09M00
8 20 10M30
8 20 "
10    Robot.Do(ldPartA);                /source code
8 21 MI -2400,-1600,800,1570,1390,0     /send robot program “ldPartA” to port COM2
8 21 MI 0,-240,-540,225,-225,0
8 21 GC
8 21 MI 0,1020,-240,-55,55,0
8 21 MI -7200,200,1000,0,0,0
8 21 MI 250,-1800,1700,0,0,0
8 21 MI 20,-530,175,0,0,0
8 21 MI -130,0,0,0,0,0
10    PalletStops.On;
1 642 0                                 /set bit 0 on on port 642
5555                                    /end of procedure Test1
4444 Test2                              /start of procedure Test2
10    Lathe.Do(ldLatheB);
8 20 %
8 20 N' G'  X '   Z '  F'  H
8 20 00M03
8 20 01 00    00 - 7100
8 20 02 01 - 100     00  80
8 20 03 01 -  50     50  25
8 20 04 01    00    500  25
```

```
8 20 05 01   50    50  25
8 20 06 00  100    00
8 20 07 00   00  6500
8 20 08M05
8 20 09M00
8 20 10M30
8 20 "
10    Robot.Do(ldPartB);
8 21 MI -2400,-1600,800,1570,1390,0
8 21 MI 0,-240,-540,225,-225,0
8 21 GC
8 21 MI 0,1020,-240,-55,55,0
8 21 MI -7200,200,1000,0,0,0
8 21 MI 250,-1800,1700,0,0,0
8 21 MI 20,-530,175,0,0,0
8 21 MI -130,0,0,0,0,0
10    PalletStops.On;
1 642 0
5555
4444 Test3                              /start of procedure Test3
10    Console.Put("Please enter part type(A/B)");
5 26 22 0 "Please enter part type(A/B)"   /display string on the screen
10    Console.Get(PartType);
5 25 22 1 PartType                      /get string from the keyboard and assign
                                        /it to variable PartType
10    Test1.Run(If PartType.EQ("A"));
2222 Test1 35                           /conditionally call procedure Test1
                                        /condition type is If clause
9 28 PartType 0 "A"                     /condition: PartType equals "A"
3333                                    /end of condition
10    Test2.Run(If PartType.EQ("B"));
2222 Test2 35
9 28 PartType 0 "B"
3333
5555                                    /end of procedure Test3
4444 ErrHdl                             /start of procedure ErrHdl
10    Console.Put("Run time error, Please check the program.");
5 26 22 0 "Run time error, Please check the program."
10    Conveyor.Off;
2 642 5                                 /set bit 5 off on port 642
10    Robot.Send("NT");
8 21 NT                                 /send string "NT" to port COM2
5555                                    /end of procedure ErrHdl
4444 Test4
10    Init.Run;
1111 Init                               /call procedure Init
10    Test3.Run;
1111 Test3                              /call procedure Test3
10    Conveyor.On;
1 642 5
```

```
10   PhotoCell.WaitOn(5000);
3 640 7 5000                              /wait bit 7 switched to on on port 640(PortA) for
                                          /5000 milliseconds
10   PalletStops.Off;
2 642 0                                   /set bit 0 off on port 642
10   PalletArrived.WaitOn(5000):ErrHdl;
3 640 6 5000 1 ErrHdl                     /wait bit 6 switched on on port 640 for 5000
                                          /milliseconds, if timeout the bit is still not
                                               /switched on, then call procedure

ErrHdl
10   Delay.1000;
6 1000                                    /delay for 1000 milliseconds
10   PalletLiftUp.Strobe;
7 642 4                                   /strobe bit 4 on port 642
10   PalletLifted.WaitOn(5500);
3 640 5 5500 0
10   Conveyor.Off;
2 642 5
10   ChuckOpen.Strobe;
7 642 1
10   Delay.1000;
6 1000
10   ChuckClose.Strobe;
7 642 3
10   Delay.2000;
6 2000
10   Robot.Do(moveaway);
8 21 GO
8 21 MI 130,0,0,0,0,0
8 21 MI -20,530,-175,0,0,0
8 21 MI 0,2360,-2660,-1680,-1160,0
10   Delay.2000;
6 2000
10   LatheStart.Strobe;
7 642 2
10   LatheStop.WaitOn(5000):ErrHdl;
3 640 4 5000 1 ErrHdl
10   Robot.Do(moveback);
8 21 MI -250,-560,960,1680,1160,0
8 21 MI 250,-1800,1700,0,0,0
8 21 MI 20,-530,175,0,0,0
8 21 MI -130,0,0,0,0,0
8 21 GC
10   Delay.2000;
6 2000
10   ChuckOpen.Strobe;
7 642 1
10   Delay.2000;
6 2000
10   Robot.Do(getpart);
```

```
8 21 MI 130,0,0,0,0,0
8 21 MI -20,530,-175,0,0,0
8 21 MI -250,1800,-1700,0,0,0
8 21 MI 7200,0,0,0,0,0
8 21 MI 0,-1180,-760,55,-55,0
8 21 GO
8 21 MI 2300,1700,-160,-1695,-1265,0
8 21 NT
10    PalletStops.On;
1 642 0
10    PalletLiftDown.Strobe;
7 642 6
10    Conveyor.On;
1 642 5
10    Delay.500;
6 500
10    Conveyor.Off;
2 642 5
10    PalletStops.Off;
2 642 0
10    LatheStart.Strobe;
7 642 2
10    LatheHandShk.Strobe;
7 641 0
5555                                            /end of procedure Test4
6666                                            /start of program
10   Test4.Run(For 2);
2222 Test4 36                                   /conditionally call procedure Test4,
                                                /condition type is For loop

2                                               /repeat twice
3333                                            /end of condition
10   Console.Put("Press any key to stop...");
5 26 22 0 "Press any key to stop running procedure Test4..."
10   Test4.Run(Until Console.Poll());
2222 Test4 38                                   /run Test4 repeatedly until a key is pressed
5 27 22
3333
10   Console.Get(Var);
5 25 22 1 Var
10   Test1.Run(While Var.NE("N"));
2222 Test1 37
9 29 Var 0 "N"
3333
7777                                            /end of program
```

REFERENCES:

Alder, A(1988), "TDL: A task description language for programming automated robotic workcells, Control and Programming in Advanced Manufacturing," *International Trends in Manufacturing Technology,* K Rathmill, Ed. Springer-Verlag, Berlin, pp. 321-327.

Benhabib, B., Chen, C.Y., Johnson, W.R., "An Integrated Manufacturing Work Cell Management System," *Manufacturing Review* vol 2, no 4, December 1989.

Chang, Tien-Chien, Wysk, Richard A., Wang, Hsu-Pin, *Computer-Aided Manufacturing,* Prentice-Hall, Inc. Englewood Cliffs, N.J., 1987.

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

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

Higgins, B. M., Prater, J. B., *Object Orientation in the CIM Environment,* Working Paper Csc-91-3, Department of Computer Science, University of Missouri - Rolla.

Marcelo, L., *Investigation and Critique of the Software Development Environment for the Manufacturing Engineering CIM Lab,* Working Paper #94-003, Systems Analysis Dept., Miami Univ., Aug, 1994.

Martin, J.M.(1989), "Cells Drive Manufacturing Strategy," *Manufacturing Engineering,* January, pp. 49-54.

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.

Shimano, B. E, Geschke, C. C., Spalding III, C. H., Goldman, R., Scarborough, D. W.(1988), "AIM: A task-level control system for assembly, Control and Programming in Advanced Manufacturing," *International Trends in Manufacturing Technology,* K Rathmill, Ed. Springer-Verlag, Berlin, pp. 305-319.

Troy, A. D., Nugehally, M., Farooq., S., Hergert., D., "Object-Oriented Flexible Manufacturing Systems at Miami University," *Proceedings of ICOOMS'* 92, May 1992.