OBJECT-ORIENTED CELL CONTROLLER
FOR A MANUFACTURING SHOP FLOOR

A thesis Presented to
The Faculty of the College of Engineering and Technology
Ohio University

In Partial Fulfillment
of the Requirements for the Degree
Master of Science

by
Sathya Gopalreddy
March, 1994

OHIO UNIVERSITY LIBRARY
# TABLE OF CONTENTS

CHAPTER I
INTRODUCTION ......................................................... 1

CHAPTER II
LITERATURE REVIEW ...................................................... 5

CHAPTER III
CELL CONTROLLER ......................................................... 12

3.1 COMPONENTS OF A DISTRIBUTED ARCHITECTURE .................. 12
3.1.1 Physical objects and logical objects: ..................... 12
3.1.2 Negotiations ........................................... 13
3.1.3 Server ............................................... 15
3.1.4 Functions of Objects ...................................... 15

3.2 OBJECTS NEEDED FOR THE CELL CONTROLLER ............... 17
3.2.1 MILL Actor ........................................... 17
3.2.2 LATHE Actor: ........................................ 17
3.2.3 ROBOT Actor ......................................... 18
3.2.4 FEEDER Actors ......................................... 18
3.2.5 BIN Actor: ........................................... 18
3.2.6 PROCESSEXECUTOR Actors ......................... 18
3.2.7 SCHEDULER Actor ..................................... 19

3.3 INFORMATION FLOW AND SCHEDULING OF OBJECTS ........ 19
3.3.1 Information flow ....................................... 19
3.3.2 Scheduling Principle ...................................... 21

CHAPTER IV
GENERIC ACTOR ......................................................... 22
4.1 C DECLARATIONS ................................................ 23
4.2 GENERIC ACTOR ............................................... 26

4.2.1 Init Step: ........................................... 27
4.2.2 ConnectToServer Macro: ................................. 28
4.2.3 ReadMessages Macro: .................................. 30
4.2.4 Abort Macro: ......................................... 31
4.2.5 Process Macro: ........................................ 32
4.2.6 KillActor Transition Label: ............................ 33
4.2.7 CloseServerConnection Step: .......................... 33

CHAPTER V
CNC MACHINE: MILL OR LATHE ......................................... 35

5.1 C DECLARATIONS ........................................... 36
5.2 PROCESS MACRO ........................................... 37
<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>VI</td>
<td>ROBOT</td>
<td>46</td>
</tr>
<tr>
<td></td>
<td>6.1 C DECLARATIONS</td>
<td>46</td>
</tr>
<tr>
<td></td>
<td>6.2 PROCESS MACRO</td>
<td>48</td>
</tr>
<tr>
<td></td>
<td>6.2.1 Reserve Macro:</td>
<td>48</td>
</tr>
<tr>
<td></td>
<td>6.2.2 SendAvailable Macro:</td>
<td>49</td>
</tr>
<tr>
<td></td>
<td>6.2.3 Exec Macro:</td>
<td>51</td>
</tr>
<tr>
<td>VII</td>
<td>FEEDERS</td>
<td>60</td>
</tr>
<tr>
<td></td>
<td>7.1 C DECLARATIONS FOR PROCESS MACRO</td>
<td>60</td>
</tr>
<tr>
<td></td>
<td>7.2 PROCESS MACRO</td>
<td>60</td>
</tr>
<tr>
<td>VIII</td>
<td>BIN</td>
<td>64</td>
</tr>
<tr>
<td></td>
<td>8.1 PROCESS MACRO</td>
<td>64</td>
</tr>
<tr>
<td>IX</td>
<td>SCHEDULER</td>
<td>66</td>
</tr>
<tr>
<td></td>
<td>9.1 C DECLARATIONS FOR PROCESS MACRO</td>
<td>67</td>
</tr>
<tr>
<td></td>
<td>9.2 PROCESS MACRO</td>
<td>70</td>
</tr>
<tr>
<td></td>
<td>9.3 SCHEDULING CLASHES AND HOW IT IS SOLVED</td>
<td>74</td>
</tr>
<tr>
<td>X</td>
<td>PROCESSEXECUTOR</td>
<td>75</td>
</tr>
<tr>
<td></td>
<td>10.1 FORMAT OF A PROCESS PLAN</td>
<td>77</td>
</tr>
<tr>
<td></td>
<td>10.2 C DECLARATIONS</td>
<td>78</td>
</tr>
<tr>
<td></td>
<td>10.3 PROCESS MACRO</td>
<td>80</td>
</tr>
<tr>
<td>XI</td>
<td>IMPLEMENTATION OF CELL CONTROLLER</td>
<td>90</td>
</tr>
<tr>
<td></td>
<td>11.1 COMPONENTS OF CIMLAB</td>
<td>90</td>
</tr>
<tr>
<td></td>
<td>11.1.1 CNC Machines</td>
<td>90</td>
</tr>
<tr>
<td></td>
<td>11.1.2 Karel System</td>
<td>90</td>
</tr>
<tr>
<td></td>
<td>11.1.3 Feeders &amp; Bin</td>
<td>91</td>
</tr>
<tr>
<td></td>
<td>11.1.4 Computer</td>
<td>91</td>
</tr>
<tr>
<td></td>
<td>11.1.5 Network</td>
<td>92</td>
</tr>
<tr>
<td></td>
<td>11.1.6 Simulation tools</td>
<td>93</td>
</tr>
<tr>
<td></td>
<td>11.1.7 Ports and Connections</td>
<td>93</td>
</tr>
<tr>
<td></td>
<td>5.2.1 Reserve Macro:</td>
<td>37</td>
</tr>
<tr>
<td></td>
<td>5.2.2 SendAvailable Macro:</td>
<td>38</td>
</tr>
<tr>
<td></td>
<td>5.2.3 LoadProg Macro:</td>
<td>40</td>
</tr>
<tr>
<td></td>
<td>5.2.4 Exec Macro:</td>
<td>43</td>
</tr>
</tbody>
</table>
## 11.2 ACTORS NEEDED FOR CIMLAB

<table>
<thead>
<tr>
<th>Actor Type</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mill Actor</td>
<td>95</td>
</tr>
<tr>
<td>Lathe Actor</td>
<td>95</td>
</tr>
<tr>
<td>Robot Actor</td>
<td>95</td>
</tr>
<tr>
<td>Feeder1, Feeder2, Feeder3, Feeder4 Actors</td>
<td>96</td>
</tr>
<tr>
<td>Bin Actor</td>
<td>96</td>
</tr>
<tr>
<td>ProcessExecutor1, ProcessExecutor2 Actors</td>
<td>96</td>
</tr>
<tr>
<td>Scheduler Actor</td>
<td>96</td>
</tr>
</tbody>
</table>

## 11.3 IMPLEMENTATION

CHAPTER XII

CONCLUSIONS

REFERENCES

APPENDIX A

FLEXIS AND SFC

A.1 FLEXIS

<table>
<thead>
<tr>
<th>Subsection</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>A.1.1 Flexis Environment</td>
<td>103</td>
</tr>
<tr>
<td>A.1.2 SFC+ Language</td>
<td>106</td>
</tr>
<tr>
<td>A.1.3 Flexis C Language</td>
<td>110</td>
</tr>
<tr>
<td>A.1.4 Cross Reference Analysis</td>
<td>110</td>
</tr>
<tr>
<td>A.1.5 Local Grafcet Execution</td>
<td>111</td>
</tr>
<tr>
<td>A.1.6 Remote Grafcet Execution</td>
<td>112</td>
</tr>
<tr>
<td>A.1.7 Additional Supported Options</td>
<td>113</td>
</tr>
</tbody>
</table>

APPENDIX B

CIMLAB USERS

<table>
<thead>
<tr>
<th>Subsection</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>B.1 DIRECTORY STRUCTURE AND FILES</td>
<td>115</td>
</tr>
<tr>
<td>B.2 ABOUT FLEXIS</td>
<td>117</td>
</tr>
<tr>
<td>B.2.1 Flexis Environment</td>
<td>117</td>
</tr>
<tr>
<td>B.2.2 SFC+ Language</td>
<td>118</td>
</tr>
<tr>
<td>B.2.3 Flexis C Language</td>
<td>121</td>
</tr>
<tr>
<td>B.2.4 Cross Reference Analysis</td>
<td>121</td>
</tr>
<tr>
<td>B.2.5 Local Grafcet Execution</td>
<td>122</td>
</tr>
<tr>
<td>B.2.6 Text Editor in Flexis</td>
<td>123</td>
</tr>
<tr>
<td>B.2.7 Documentation and Printing Grafcet</td>
<td>123</td>
</tr>
<tr>
<td>B.3 HOW TO START CIMLAB PROGRAMS</td>
<td>124</td>
</tr>
<tr>
<td>B.3.1 How to Start Openwindows</td>
<td>124</td>
</tr>
<tr>
<td>B.3.2 How to Start Flexis</td>
<td>124</td>
</tr>
<tr>
<td>B.3.3 How to Create Grafcet Programs</td>
<td>124</td>
</tr>
<tr>
<td>B.3.4 How to Start Siman</td>
<td>129</td>
</tr>
<tr>
<td>B.3.5 How to Start Server</td>
<td>129</td>
</tr>
<tr>
<td>B.3.6 How to Run Simulation</td>
<td>130</td>
</tr>
</tbody>
</table>
## LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Negotiations</td>
<td>14</td>
</tr>
<tr>
<td>3.2</td>
<td>Communicating Objects</td>
<td>16</td>
</tr>
<tr>
<td>3.3</td>
<td>Information Flow</td>
<td>20</td>
</tr>
<tr>
<td>4.1</td>
<td>Actor Hierarchical Structure</td>
<td>23</td>
</tr>
<tr>
<td>4.2</td>
<td>Generic Actor</td>
<td>27</td>
</tr>
<tr>
<td>4.3</td>
<td>ConnectToServer Macro</td>
<td>28</td>
</tr>
<tr>
<td>4.4</td>
<td>ReadMessages Macro</td>
<td>30</td>
</tr>
<tr>
<td>4.5</td>
<td>Abort Macro</td>
<td>31</td>
</tr>
<tr>
<td>4.6</td>
<td>Lathe Process Macro</td>
<td>33</td>
</tr>
<tr>
<td>5.1</td>
<td>CNC Machine Process Macro</td>
<td>35</td>
</tr>
<tr>
<td>5.2</td>
<td>Reserve Macro</td>
<td>37</td>
</tr>
<tr>
<td>5.3</td>
<td>SendAvailable Macro</td>
<td>39</td>
</tr>
<tr>
<td>5.4</td>
<td>LoadProg Macro</td>
<td>40</td>
</tr>
<tr>
<td>5.5</td>
<td>Exec Macro</td>
<td>43</td>
</tr>
<tr>
<td>6.1</td>
<td>Robot Process Macro</td>
<td>46</td>
</tr>
<tr>
<td>6.2</td>
<td>Robot Reserve Macro</td>
<td>48</td>
</tr>
<tr>
<td>6.3</td>
<td>Robot SendAvailable Macro</td>
<td>50</td>
</tr>
<tr>
<td>6.4</td>
<td>Robot Exec Macro</td>
<td>52</td>
</tr>
<tr>
<td>7.1</td>
<td>Feeder Process Macro</td>
<td>61</td>
</tr>
<tr>
<td>8.1</td>
<td>Bin Process Macro</td>
<td>64</td>
</tr>
<tr>
<td>9.1</td>
<td>Scheduler Process Macro</td>
<td>67</td>
</tr>
<tr>
<td>10.1</td>
<td>ProcessExector Process Macro</td>
<td>76</td>
</tr>
<tr>
<td>11.1</td>
<td>Computer Connections</td>
<td>92</td>
</tr>
<tr>
<td>11.2</td>
<td>Connections</td>
<td>94</td>
</tr>
<tr>
<td>A.1</td>
<td>Fixed Menu</td>
<td>104</td>
</tr>
<tr>
<td>A.2</td>
<td>Flexis Menu</td>
<td>105</td>
</tr>
<tr>
<td>A.3</td>
<td>Hierarchial Structure</td>
<td>107</td>
</tr>
<tr>
<td>A.4</td>
<td>Actions Menu</td>
<td>109</td>
</tr>
<tr>
<td>A.5</td>
<td>A Sample Activation Program</td>
<td>110</td>
</tr>
<tr>
<td>A.6</td>
<td>Cross Reference Menu</td>
<td>111</td>
</tr>
<tr>
<td>A.7</td>
<td>Local Execution Menu</td>
<td>112</td>
</tr>
<tr>
<td>A.8</td>
<td>Remote Execution Menu</td>
<td>113</td>
</tr>
<tr>
<td>B.1</td>
<td>PROJECT1 ACTOR</td>
<td>125</td>
</tr>
<tr>
<td>B.2</td>
<td>Process Macro</td>
<td>126</td>
</tr>
<tr>
<td>D.1</td>
<td>devel Hierarchial Structure</td>
<td>157</td>
</tr>
</tbody>
</table>
CHAPTER I

INTRODUCTION

Consumer needs, requirements and desire to acquire new things are fast changing. Greater reliability of the product is in more demand, and the pressure to improve the quality and safety of products is increasing. Customer satisfaction has taken highest priority. People are seeing work as a way to enhance their skills as well as their source of income. Economic indicators, such as inflation and interest ratio, move in unprecedented ways. Market information, demand, customer orders and technical information has to be transferred instantly to any part of the globe. The market for manufacturing products is becoming increasingly international.

With the fast change in requirements in the global market, the demand for industry to keep pace is becoming more complex. New products must become more efficient and flexible. The manufacturing processes must be more pollutant free, and the products must be energy efficient, easily recyclable and manufactured with minimal use of materials that may harm the planet.

As a result of manufacturing companies being under diverse and increasing pressures, operations are becoming more of a challenge. New technologies and new applications of existing technologies offer new opportunities to companies and require new skills from workers and managers. The manufacturing units have to be integrated
to make quick decisions and to optimize the information flow. The decisions taken in any part of the enterprise must be unique and synchronized. Different functions, such as manufacturing, research, administration, marketing, suppliers and customers in an enterprise are to be co-ordinated with one another to meet the requirements. An integrated enterprise represents the ideal response to change the various parts of the enterprise to work more closely together and respond quickly to changes in technology and the market. Thus, Computer Integrated Manufacturing (CIM) has responded to this challenging market.

The evolution of CIM is directly attributed to software development and achieving the required levels of integration between systems. These problems are evident in the development of the low level part of CIM system; i.e., implementation of CIM in the shop floor. The integration of machine tools, robots and automatic guided vehicles (AGV's) with the centralized system is critical in implementing CIM. Based on the production requirements, the shop floor is responsible for planning, scheduling and controlling the shop floor equipment. Machine tools, robots and AGV's have to be utilized to the maximum extent and the throughput has to be optimized. The cell controller is responsible to control the shop floor and to achieve the desired results. The cell controller must integrate with the other levels of the CIM as well as achieve the desired production targets.

The functions and capabilities of the controller have made much progress in the
last decade, and individual controllers are now able to share the decision making capabilities in the manufacturing systems. The individual controllers hasn’t made much progress towards the independence of process plans, CNC programs and to integrate with the other levels of CIM. The reusability and maintainability of the software has become important while developing the software.

With the concept of object-oriented modelling, the barriers limiting the software to a fixed number of systems is eliminated. The object is defined for a particular type of manufacturing equipment and can be used for any similar equipment. To approach this Flexis, a grafcet package was used to build the objects and TCP/IP sockets was used to communicate between objects. This approach gives more flexibility to built the objects and eliminates the re-coding of software for any change in the structure of the shop floor. This is the concept used in building the proposed cell controller. The object in the cell controller is called an actor. An actor is built for a particular type of equipment so that it is not limited to only a particular structure of the shop floor, but also easily changed to any type of structure without re-coding the entire software. To solve the problem of having a cell controller to be independent of process plans and CNC programs a independent process plan executing actor was built which reads the process plans and CNC programs. Also a scheduler actor was built which is independent of process plans and to schedules any type of process plans. The proposed cell controller also integrates a variety of machines already operating independently in the manufacturing industries. The cell controller uses the network to integrate with the other levels of CIM.
Chapter 2 describes the different cell controller architectures and the proposed cell controller architecture. Chapter 3 describes the building of the cell controller, integration with the equipment and the concept of negotiations. Chapter 4 describes the generic actor built for the cell controller. Chapters 5 to 10 describes the different actors required for the cell controller. Chapter 11 describes the structure of the CIM laboratory and implementation of the cell controller in the CIM laboratory. The building software Flexis and SFC+, CIMLAB users, description for include files routines, how to run the cell controller and the actual source code for the include files was described in Appendix A, B, C, D and E respectively.
CHAPTER II
LITERATURE REVIEW

The initial manufacturing process is based on mechanization, in terms of increasing efficiency and control. Now the emphasis is on product and process performance and intelligence. With the evolution of information processing and networking, manufacturing entered into the era of CIM in the late 1980s. The initial trend in CIM is to automate the device controllers and integrate the business systems and CAD/CAE system. Many Computer Integrated Manufacturing architectures are proposed.

One CIM architecture for Flexible Manufacturing Systems (FMS) was presented by Hassan Gomaa [1986]1 of General Electric. In it he reviewed the information flow in the factory and the trends in computer technology. He described an architecture in which each level of the architecture accepts commands for the level above. He presented a shop floor control function as an on-line function, for managing and controlling the activities on the shop floor. The shop floor executes the schedule provided by business planning. The schedule is based on event driven inputs such as when equipment is down or material is not available. In his paper he describes a Flexible Manufacturing Cell Controller (FMCC) to manage and dispatch work to the manufacturing workstations. To efficiently route parts through the cell, FMCC either uses its own simple job scheduling

---

algorithm or is directed from a job schedule produced by a production management system. An automated FMCC requires sophisticated interaction between the FMCC and the device controller. He presented an example of a simple control definition diagram.

S.P. Moss [1989]² introduces Manufacturing Management and Control System (MMCS), a multi-level management and control architecture for factory floor CIM systems. He stressed developing a generic software for designing and implementing flexible CIM systems. The implementation of a CIM system is achieved through planned migration, in which each enhancement and addition is integrated within an established communication infrastructure. He presented the MMCS concepts as

"The multi-level structure with breakdown of goals and the production reference scheme allow problems to be resolved locally where the problems including rescheduling are relatively simple and therefore can be performed fast."

"The distinctions between job planning/scheduling and resource scheduling as well as job and resource monitoring provides a close-loop environment where effective utilization of resources can be combined with a high degree of flexibility."

"The combined predictive and event-driven job dispatching function at cell level allows activities/job to be performed concurrently, synchronized to

---

the latest observed events and intimated as late as possible, yet early enough to meet due dates. This approach minimizes the risk for blocking and overload and simplifies significantly recovery from errors or breakdowns."

Albert Jones and Abdol Saleh [1990]³ proposed a multi-layer/multi-level control architecture to intelligently manage shopfloor activities. They proposed an architecture based on multi-level and multi-layer control to decompose the planning and scheduling problems and partition their respective decision spaces and generate all the information needed to solve the resulting subproblems. Each decomposed structure performs three major control functions: adaptation, optimization and regulation. Adaptation is responsible for generating and updating plans for executing assigned tasks. Optimization is responsible for evaluating proposed plans and generating and updating schedules. Regulation is responsible for interfacing with subordinates and monitoring the execution of assigned tasks.

D.H. Norrie, O.R. Fauvel and B.R. Gaines [1990]⁴ described an object-oriented management planning systems (OBJIMPS) for advanced manufacturing, in which part objects, machine objects and other physical objects interact with the necessary


management objects for supervision planning and associated functions. OBJIMPS is an interactive system comprised of modules which carry out process planning, cell grouping, scheduling, simulation and other function, with each module having its own rules or expertise embedded within it. Each module, submodule or other component of the system is an object, with interaction by message sent from object to object. The process planning module determines the operations for a part, machines for operation, tools required, operation time needed and so on. The cell grouping include setting up parts machines and defining machines for part groups.

Paul Rogers [1992] presented the research on the development of object oriented techniques for the design and control of manufacturing systems. According to him such techniques have the potential both to reduce the system design cycle time and to result in systems capable of more intelligent decision making. The object oriented technique supports modularity, flexibility and the integration of modelling tools and techniques. He presented two types of responsiveness for the manufacturing systems: (1) behavioral responsiveness, which is more adaptive to changes in their state, and (2) design responsiveness techniques for decreasing the lead time from the conception to implementation of systems required. Object oriented modelling provide a flexible and integrative framework in which responsive control systems and a responsive modelling environment is constructed. A manufacturing entity is viewed as a state driven decision

---

making module. He presented his research on two projects: (1) real time scheduling of a small Flexible Manufacturing System (FMS), which addressed both behavioral and design responsiveness and a single control module to carry out the real time scheduling, and (2) dynamic cutting tool flow in FMS, which addressed how much autonomy is required in a particular manufacturing environment.

Toshimichi Moriwaki, Nobuhiro Sugimura, Yatna Y. Martawiry and Sri Haridjoko Wirjomartono [1992] proposed an object-oriented modeling of autonomous distributed manufacturing systems. In it they considered the autonomous functions of individual components from two different viewpoints: the physical viewpoint and the logical viewpoint. They considered the information processing devices, such as the MC machine tool and the NC robot, as a physical view and all the components in the manufacturing systems, such as the machine tools, AGV’s, workpieces and operators from the logical view. All the components in the manufacturing systems share the decision making functions of the production control. The autonomous components had (1) data representing their static and dynamic status and (2) procedures to make decisions and to communicate the data with the other components. An object is represented by the data and the methods in the object-oriented schema. They considered two types of relations between the objects: part_of relation and is_a relation. The part_of relation describes that a particular object is a component of another object. The is_a relation gives the relation

---

between an abstract object and a specialized object. The production schedules in the manufacturing systems are determined by the autonomous decision making by the objects of the machining cells and the lots. To avoid the collisions while making decisions, a negotiation mechanism is implemented and the co-ordinator -- which is an object -- co-ordinates the conflicts among the objects.

Jeffry S. Smith and Sanjay B.Joshi [1992] proposed a object oriented development of shop floor control systems (SFCS) for the Computer Integrated Manufacturing. The shop floor is presented as collections of interacting part objects, material handling objects and machine objects. The objective of the SFCS is to control the interaction of these objects so that the part throughput for the system is optimized.

To improve the controllability of the system and allows the control software to be more generic, they organized the equipment and control units into a three-level hierarchial structure. The three levels are the shop floor level, the workstation level and the equipment level. The equipment level provides a logical view of the physical machines on the shop floor. The workstation level is defined by the physical layout of the equipment. The shop level is responsible for determining the routes that the part will take through the shop. The controller is based on individual part states rather than part-machine contact states. The shop floor system is based on a message passing paradigm.

Control is exercised by executing a message based part state graph (MPSG). In the

---

operation, the workstation controller sends the messages to the equipment controller; then the equipment controller executes the message. The shop floor class has been implemented in the C++ language. Since their level of control is a hierarchy, the decisions have to pass through the levels. They feel the controllability of the system through a hierarchial structure allows the control software to be more generic.

The hierarchical control structure is suitable for productions in a steady state. For small batch productions with the dynamic changes in the volumes and varieties of the products, the hierarchial structure is not very suitable. The distributed control structure realizes more flexible control on the system. The decisions are taken on the same level. To implement the distributed structure objects are created in the same level. The objects are independent of one another. This gives more flexibility to have the number of objects for any manufacturing system. The number of objects depends on the number of machine tools, robots and AGV's present in the manufacturing system. To pass the messages between the objects, a concept of negotiations in implemented. The objects can talk with the other objects through a command or a request and reply. The concept of negotiations is discussed further in Chapter III.
The proposed controller is built on distributed architecture but without a centralized monitoring system. Each physical system in the manufacturing facility is represented as an object. Each object acts independently on its own and interacts with the other objects to complete its task. This control structure gives more flexibility to cope with the dynamic changes in the production at real time and also more flexibility to the physical changes in the manufacturing facility, like adding new CNC machines, robots, etc. The controller's aim is giving autonomous function to each component in the manufacturing facility. The controller can schedule the machines at real time and in parallel can function and control any control software, like simulation tools.

3.1 COMPONENTS OF A DISTRIBUTED ARCHITECTURE

This section describes the components of the proposed architecture. Physical and logical objects to execute the physical tasks and logical tasks is defined. The concept of negotiations and the server implementation is explained.

3.1.1 Physical objects and logical objects:

An object is a single processing device, which is responsible for executing certain tasks. The tasks can result from a physical function or a logical function. A physical task for an object is the steps required to execute the part, like loading CNC programs, running CNC machine, etc. A logical task for an object is the steps required to
synchronize the tasks, like reserving the CNC machine, releasing the CNC machine after execution, etc.

The objects are classified into physical objects and logical objects. A physical object is a representation of a physical system. It executes the process steps involved in the physical system and is responsible for the control of that physical system. A physical object can have physical tasks, logical tasks or both. For example consider a Mill object, which represents a physical Mill CNC machine. The CNC machine has physical tasks, like loading CNC programs and executing the part, and it has logical tasks, like a ProcessExecutor reserving and releasing the Mill semaphore. A logical object is a representation of logical tasks involved in streamlining the manufacturing process. For example, consider a ProcessExecutor object; its objective is to complete all the steps in the process plan of a part. The ProcessExecutor’s are not physical systems present in the manufacturing facility, but represents an object that streamlines the process.

3.1.2 Negotiations:

Figure 3.1 shows the types of negotiations between objects. A negotiation can be executed by a command or a request. An object can command other objects to execute a specific task to it, or it can request other objects for a specific task and wait for its reply. An object can negotiate with a real time machine to complete its task. The negotiations are synchronized through a server, which passes the messages between different objects.

COMMAND: A command is sent to an object which is waiting for the command.
Figure 3.1 Negotiations
When an object receives a command, it can read the parameters on the command and then the received command must be cleared.

**REQUEST:** A request is sent to an object which is waiting for the request. When the objects receives the request, it can read the parameters on the request; after finishing the required processing steps a reply must be given to the request, since the requesting object waits for the reply. The received request must be cleared after the reply sent.

### 3.1.3 Server:

The Server is the communication channel for all the objects, real time machines and different control tools, like Siman. All the objects, real time machines and control tools connect to the Server and then start the negotiations between them. Figure 3.2 shows the connections between the objects, real time machines and simulation tools, the server being the communication channel.

### 3.1.4 Functions of Objects:

The functions for physical objects and logical objects are different. A physical object talks to the different objects in the controller and also with its real time machine. A physical object can have a physical task, logical task or both. A logical objects talks to objects in the cell controller. It has only logical tasks. Each object must have a unique identification number to identify itself. On the basis of the identification number,
the actor negotiates.
3.2 OBJECTS NEEDED FOR THE CELL CONTROLLER

Objects representing the physical machines and logical functions are required. Since CIMLAB has a CNC mill machine, CNC lathe Machine, robot, feeders and bin, the cell controller requires Mill, Lathe, Robot, Feeder, Bin as actor objects. To execute the part and to schedule the part, logic objects like Scheduler and ProcessExecutor’s are required. Each object and its function is described below.

3.2.1 MILL Actor:

The basic function of the Mill object is to interact between the real time mill machine and the other actors in the controller. The Mill actor has logic functions and real time functions. The logical function is to maintain the machine semaphore. The real time functions are interaction with the mill machine, i.e, to load programs and to execute programs.

3.2.2 LATHE Actor:

The basic function of the Lathe object is to interact between the real time lathe machine and the other actors in the controller. The Lathe actor has logical functions and real time functions. The logical function is to maintain the machine semaphore. The real time functions are interaction with the lathe machine, i.e, to load programs and to execute programs.
3.2.3 ROBOT Actor:

The function of the Robot object is to interact between the real time robot machine and the other actors in the controller. The Robot actor has logic functions and real time functions. The logic function is to maintain the machine semaphore. The real time function is to interact with robot to move the robot, to unload a part and to load a part.

3.2.4 FEEDER Actors:

The function of the feeder is to maintain the part identification and the number of parts for each identification in the system. The Feeder actor doesn’t interact with the real feeders, but has the information about the number of parts and the process plan. It basic function is to send part information to the Scheduler. When ever robot picks a new part, the Feeder Actor sends the information about the next available part.

3.2.5 BIN Actor:

It is just an object to represent a bin, it doesn’t interact with the physical bin. Whenever robot puts a part in a bin, Robot actor sends a command to the Bin actor that a part is placed in bin.

3.2.6 PROCESSEXECUTOR Actors:

The function of the ProcessExecutor is to get a raw part and complete all the process steps required for the part. The number of ProcessExecutor’s depend on the number of parts that can flow simultaneously in the system. Since the CIMLAB has a
mill and a lathe machine, it can have, at most two parts at all times; therefore, there are two ProcessExecutor's implemented.

3.2.7 SCHEDULER Actor:

The function of the Scheduler is to schedule the parts that are being executed by the ProcessExecutor's and assign these parts to the ProcessExecutor's. The efficiency and utilization of the CNC machines depends on how the scheduler assigns the parts. We implemented a scheduler that will maximize the utilization of the machines with no deadlock.

3.3 INFORMATION FLOW AND SCHEDULING OF OBJECTS:

The information flow between different objects and the scheduling principle is explained below.

3.3.1 Information flow:

Figure 3.3 shows the information flow between the different objects. The information flow between all objects is through a server. Each object has to register itself with the server. The server is a program that allows multiple actors to talk with one another. Also, the objects can talk to the real time machines and simulation tools like Siman. The server gives a server port for each object; through this server port each object talks to the other object.
3.3.2 Scheduling Principle:

The assignment of CNC machines or robot is done as first-in first-out basis. The reserving object identification numbers are kept in a reserve queue, and whenever the machine is free, it is assigned to the ProcessExecutor that is first in this queue.

The Scheduler keeps track of the parts and part information it handles. Each Feeder informs the Scheduler about which resources the part requires and, in turn, the Scheduler determines which part can be processed to maximize the utilization of the machines. Next, the Scheduler assigns the part to a ProcessExecutor. The Scheduler is built to maximize the utilization of machines. It keeps track Work In Process (WIP) of the ProcessExecutor’s so that, whenever a ProcessExecutor is free, it can assign another part whose WIP doesn’t clash with the current WIP. The Scheduler is responsible to avoid deadlocks in the execution. The ProcessExecutor has to complete the part. It reserves the machines required and processes the part, moves the part to the machines and downloads the programs. The machine objects, Robot, Mill, Lathe have to keep track of which ProcessExecutor is requesting the machine and assign the machine for the requested ProcessExecutor. Also the machine objects have to talk with real time machines and get the part finished.
CHAPTER IV

GENERIC ACTOR

A single process in Flexis system is an object called actor. An actor represents a physical system or a logical system that is present in any manufacturing facility. All the process steps involved in a physical system or a logical system are represented sequentially in an actor. Each actor communicates with the other actors through a server. Each actor must register itself with server with its unique identification declared in system.dcls include file. The communications between actors is based on negotiations through a Command or a Request and a Reply. This means each actor can command any other actor to do a specific action, or it can request any other actor for an action and wait for the reply. This type of negotiation is an artificial intelligence technique in which the each actor completes its assigned tasks through negotiations.

The hierarchial structure (Figure 4.1) shows the nesting of the basic Grafcet macro steps in an actor. The first macro is ConnectToServer, which registers the actor with the server. Through a synchronous branching, ConnectToServer macro is connected to three parallel macro’s, namely, Abort, ReadMessages and Process. These three macros execute in parallel till the actor is aborted due to an error or a KILL message from the server. Each actor has C declarations declared globally. In addition, depending on the Actor requirements, declarations are declared for the process macro.
4.1 C DECLARATIONS

C Declarations for: actorName

```c
#include "../include/system.dcls"
#define MYID actorName
#define DEBUG
#include <stdlib.h>
#include <stdio.h>
#include <flexis.h>
#include <flxtcp.h>
#include <ctype.h>
#include <string.h>
#include <timer.h>
#include "../include/inet.h"
#include "../include/inet.errors"
#include "../include/inet.lib"
#include "../include/rnsg.h"
#include "../include/msg.lib"
#include "../include/flexis.dcls"
#include "../include/pp_plan.lib"
```

The include files are divided into two groups: (a) files that are available with the Flexis system, and (b) program files that are written for this project. The include files are described in the Appendix D. All the include files that are written for this project are placed in include directory.

system.dcls

The system.dcls file contains the C preprocessor defined, unique identification numbers for the actors and the message identification numbers for the messages. The identification names give an easy
way to identify the actors and the messages.

**MYID**

MYID is used in inet.lib library file to keep the file common to all the actors. So MYID is defined with the actorName to have unique identification for the actor. **NOTE:** system.dcls has to precede MYID since all the identification names are declared in system.dcls.

**DEBUG**

The C Preprocessor #define directive DEBUG is used to display debug messages in the Flexis trace window for analysis. If this line is removed the debug messages are not displayed in the Flexis trace window.

**stdlib.h**

The Flexis C library stdlib.h header file has functions of general utilities. The functions are used in the include files that are written for this project and the also in the actor.

**stdio.h**

The Flexis C library stdio.h header file is has functions to perform input and output operations. The functions are used in the include files that are written for this project and also in the actor.

**flexis.h**

The Flexis C library flexis.h header file has functions and types that are specific to the Flexis and Grafcet environment. The functions are used in the include files that are written for this project and also in the actor.

**flxtcp.h**

The optional Flexis C library flxtcp.h header file has functions and types that are specific for communication with the network. The
functions are used to open ports, to register with the server and to communicate with the server.

cctype.h
The Flexis C library cctype.h header file has macros that are useful for testing and mapping characters. The functions are used in PP_Plan.lib library file and also in the actor.

string.h
The Flexis C library string.h header file has a macro and functions that are useful for manipulating character type arrays. The functions are used in PP_Plan.lib library file and also in the actor.

timer.h
The Flexis C library timer.h header file declares a type definition and several functions that are used in the Flexis countdown timer operations. The functions are used in the actor.

inet.h
The inet.h header file is written for this project and it has preprocessor #define directives, type definitions for messages and port specifications. The file is used in inet.lib library file.

inet.errors
The inet.errors header file is written for this project and it has preprocessor #define directives, type definitions for error codes and message types that are used in inet.lib library file.

inet.lib
The inet.lib library file has functions that are specific for the actor to open a port with the server and to connect with the server. All these functions are used initially at the start of execution.

msg.h
The msg.h header file is written for this project and has preprocessor #define names that are specific for the message
processing. This file is used in msg.lib library file.

**msg.lib**
The msg.lib header library file is written for this project and has functions for message processing. The library routine processes received and sent messages by adding required header to the messages and removing the headers for the received messages.

**flexis.dcls**
The flexis.dcls header file is written for this project. It has preprocessor #define directives, types and functions that are specific for masking, *DeviceName* enumerations, etc..

**pp_plan.lib**
The pp_plan.lib header library file has macros, types and functions specific for reading the process plan. NOTE: This file is included only for the actors that are required to read the process plan, i.e, in PROCESEXECUTOR’s and SCHEDULER.

### 4.2 GENERIC ACTOR

Figure 4.2 shows the generic actor used in the project. The actor is constructed using steps, transitions and macros. The initial starting step is **Init** connected to **ConnectToServer** macro through an always true transition. Through a synchronous branching, the **ConnectToServer** macro is connected to **Abort**, **Process** and **ReadMessages** macros. The synchronous branching is done to run three parallel programs: (1) to abort the actor when there is an error, (2) to run a process and (3) to read the messages from the server. When the **kill** transition becomes true, the token is sucked out from the abort macro’s. Through a synchronous join, the token is passed to
CloseServerConnection step, which closes the server port.

NOTE: A step without an identification name represents a step without any C programs and is used to synchronize the token. A transition without a name represents an always true transition which allows a token to pass through.

4.2.1 Init Step:

Init step initializes the global variables.

Activation program in Init:
The activation program calls `init_actor()` function to initialize the global variables, i.e., it initializes `Error_code = 0`, `KillActor = False`, `Rec_kill_msg = False`.

### 4.2.2 ConnectToServer Macro:

Shown in the Figure 4.3 is ConnectToServer macro. It connects the actor to the server, registers itself with the server and waits for a `go` message from the server. Once the actor gets a `go` message, the token is passed to Abort, Process and ReadMessages macros.

![Figure 4.3 ConnectToServer Macro](image)

**Activation program in connect (IN):**

```c
Error_code = server_connect();
```

FLEXIS tcp_connect routine does not allow more than one actor to access a given port. To overcome this problem, a single `server_connect` function must first connect to the generic port, read the unique server port, connect
to the unique server port and close the generic port before the next actor is scanned. The function returns errors

COULD_NOT_OPEN_GENERIC_PORT,

COULD_NOT_OPEN_SERVER_PORT, if there is an error when connecting to the server.

**Activation program in Register:**

```c
server_register(MYID);
```

Registers its identification number with the server by calling low level tcp_write routine with unique server file descriptor and identification number as parameters. The token waits in the step until it reads the first message from the server, which is always a go message.

**Transition Label for MsgPresent:**

```c
tcp_chars_avail(Server_fd) > 0
```

The tcp_chars_avail routine in FLEXIS returns the number of characters available in its unique server port. If there are no characters available for the port, it returns zero.

**Activation program in ReadGoMessage (OUT):**

```c
Error_code = get_go();
```

Checks the first message from the server. The function returns NONE if there is no error; otherwise it returns GO_NOT_FIRST_MESSAGE. If the
first message is a go message the token is passed to Abort, Process, ReadMessages macros for execution.

4.2.3 ReadMessages Macro:

Shown in Figure 4.4 is ReadMessages macro. The token waits in WaitForMessage step. If there is any message present in the server socket, the MsgPresent transition becomes true. The ReadAndProcessMessage step reads the message and goes back and waits for the next message.

![Figure 4.4 ReadMessages Macro](image)

Transition Label for MsgPresent:

\[ \text{tcp\_chars\_avail(Server\_fd)} > 0 \]

The tcp_chars_avail routine in FLEXIS returns the number of characters available in its unique server port. If there are no characters available for the port, it returns zero.

Activation program in ReadAndProcessMessage:
\begin{verbatim}
Error_code = get_message();
\end{verbatim}

The step call get_message routine to get the message contents. The get_message routine calls rec_msg to get the type of message, and, depending on the type of message, the message is processed. The function returns errors, GO_SENT_AFTER_PORT_OPENED, HALT_NOT_SUPPORTED, MESSAGE_RETURNED_FROM_SERVER, RECEIVED_CLOSE, OR UNDEFINED_MESSAGE_TYPE.

4.2.4 Abort Macro:

Figure 4.5 shows an Abort macro. The token waits in IN step for the ErrorCode transition set to true. When there is an error in any part of the actor the transition becomes true and the token passes to the Abort step. The Abort step sets KillActor variable to true and sends a KILL message to the server.

\begin{figure}
\centering
\includegraphics{AbortMacro}
\caption{Abort Macro}
\end{figure}

Transition Label for ErrorCode:

\begin{verbatim}
Error_code < 0
\end{verbatim}

The Error_code is a global variable set to negative in any part of the actor if there is an error. When the Error_code is set to negative, the transition
becomes true and the token passes to the connected step.

**Activation program in Abort (OUT):**

```c
KillActor = T;
send_typed_msg(KILL, SERVER, " ");
```

The OUT step sets the global variable KillActor to true and calls the function send_typed_msg to send a KILL message to the server. The send_typed_msg function in turn calls a low level tcp_write routine to close server file descriptor.

### 4.2.5 Process Macro:

Depending on the type of actor, each actor has its own process. The process macro defines the steps involved to accomplish the actor task. The process can be a representation of a physical process or a logical process. The physical process is the representation of steps involved in a physical machine. The logical process is the representation of steps involved in a logical system to control the flow of a part and to schedule the processes.

For example, consider the Lathe actor’s process macro shown in Figure 4.6. It has four synchronized macros running in parallel. Each macro has its specific function to execute: (1) reserving the machine, (2) releasing the machine, (3) loading the program, and (4) running the machine. This gives more flexibility in defining actor. Each actor is discussed independently in the following chapters.

**NOTE:** To create a new actor with new functions, only the Process macro has to
be build and a unique identification number has to be assigned to it.

4.2.6 KillActor Transition Label:

Transition Label for KillActor:

abort_step(KillActor)

The abort_{step}(bool variable) is a FLEXIS routine. When the variable becomes true the routine gets the token out of an Abort macro and passes the token to the next connected step.

4.2.7 CloseServerConnection Step:

When the actor is killed, the CloseServerConnection step closes the server port.

Activation program in CloseServerConnection:

```c
if(tcp_status(Server_fd) == 1) {
    flx_trace("Port Open \n");
    if (!Rec_kill_msg)
        send_typed_msg(CLOSE, SERVER, " ");
```
The tcp_status checks the status of the server socket. If it is open, it checks whether the actor received a KILL message and calls tcp_close to close the server socket. If the actor receives a KILL message, there is no need to send a CLOSE server port message to the server. If the actor is sending a KILL message a CLOSE message is sent to the server. This is accomplished by checking the variable Rec_kill_msg. If Rec_kill_msg is false; a CLOSE message is then sent to the server.
CHAPTER V

CNC MACHINE: MILL OR LATHE

Both the CNC machines actors have same process steps. In each CNC Machine actor's Process macro there are four macros running parallel: Reserve, SendAvailable, Exec and LoadProg. Each macro has its own function. The Reserve macro keeps the identification numbers of the PROCESSEXECUTOR's which are reserving the machine. Depending on the availability of the machine the SendAvailable macro sends the availability of the machine to the ProcessExecutors. The Exec macro executes the CNC machine. LoadProg macro loads the requested programs in the CNC machine. Figure 5.1 shows the LATHE Process macro.

Figure 5.1 CNC Machine Process Macro

NOTE: Since MILL and LATHE actors have same Processes, only LATHE Process is
explained. Only difference between MILL and LATHE Processes is the MILL actor talks to the BETA PC and LATHE actor talks to GAMMA PC.

5.1 C DECLARATIONS

C declarations in Process Macro:

```c
#define MAXRESERVEQSIZE 5

int QueueStep = 0;
int reqQ[MAXRESERVEQSIZE];

void addReqQ(int id) {
if(QueueStep == MAXRESERVEQSIZE) {
    Error_code = MAX_reserve_Q_SIZE_EXCEEDED;
    return;
}
reqQ[QueueStep] = id;
QueueStep++;
}

int readReqQ(void) {
int id, i;

id = reqQ[0];
for(i=0; i<QueueStep; i++) {
    reqQ[i] = reqQ[i+1];
}
QueueStep--;
return (id);
}
```

QueueStep It is an Integer used between Reserve and SendAvailable macros to store the current free place in the reserve request queue. It is used in addReqQ and readReqQ functions.

reqQ[] It is the Queue for storing the identification numbers of the ProcessExecutors which are requesting the Machine.

addReqQ This function adds the identification number of the ProcessExecutor which is requesting the machine to the end of its request queue.

readReqQ This function returns the identification number of the
ProcessExecutor, of the first machine and removes it from the queue.

5.2 PROCESS MACRO

5.2.1 Reserve Macro:

Figure 5.2 shows the Reserve Macro. Its function is to add the identification numbers of the ProcessExecutors that are requesting the machine. When it receives a request for the machine, it adds the identification number of the request actor in the reserve queue and goes back to wait for another request.

![Figure 5.2 Reserve Macro](#)

**Activation program in Init:**

```c
int i;
for(i=0, i<MAXRESERVEQSIZE; i++) reqQ[i]=0;
```

Initialize the request queue.
Transition label for Reserve:

\texttt{wait\_cmd(RESERVE);};

In this transition, it waits for a command from ProcessExecutor reserving the machine.

Activation program in ReqFrom:

\texttt{int reqFrom;}\n\texttt{reqFrom = source\_cmd(RESERVE);}\n\texttt{clear\_cmd(RESERVE);}\n\texttt{addReqQ(reqFrom);};

When it receives a request from the ProcessExecutor, it gets the identification number of the ProcessExecutor by source\_cmd function. The received command is cleared and the identification number is added in the request queue.

5.2.2 SendAvailable Macro:

Figure 5.3 shows the SendAvailable Macro. This macro waits for a request for the machine. If a request is present and CNC Machine is free, the requesting actor is informed that it is free to use the machine. It uses FIFO sequence to assign the machines.

Transition label for MachineAvailable:

\texttt{QueueStep> 0} \n
Checks for any request present.
Activation program in SendAvailable:

```c
int dest;
dest = readReqQ();
cmd(dest, AVAILABLE, "");
```

It reads the request queue and gets the first ProcessExecutor ID in queue, assigns the machine to this ProcessExecutor and sends the command, AVAILABLE, to the ProcessExecutor.

Transition label for WaitForRelease:

```c
wait_cmd(RELEASE)
```

When the ProcessExecutor finishes the use of machine, it releases the machine by sending command RELEASE.
Activation program in Clear:

clear_cmd(RELEASE);

In this step it clears the received command RELEASE.

5.2.3 LoadProg Macro:

Figure 5.4 shows the LoadProg Macro. This macro communicates with the computer that controls the CNC machine to load program if and when required. It waits for a request from ProcessExecutor to load the program. When it gets the request, it sends the request to the computer that controls CNC machine, to Siman, a simulation tool, or simulates loading program for a specific time. When it receives the replies, it replies back to the ProcessExecutor.

![Figure 5.4 LoadProg Macro](image_url)
C declarations for LoadProg Macro:

```c
char *prog;
int MachineProg, SimanProg;
TIMER LatheTimer;
```

- `char *prog` It is a pointer to the CNC program name that is read from the request it received from the ProcessExecutor.
- `MachineProg` Used to store the transaction number returned by the request sent to CNC machine to load the CNC program.
- `SimanProg` Used to store the transaction number returned by the request sent to Siman to load the program.
- `LatheTimer` It is a pointer to the Flexis timer.

Activation program in Init:

```c
timer_set(&LatheTimer, 0);
```

LatheTimer is set to zero.

Transition label for LoadProg:

```c
wait_req(LOAD_PROG);
```

Waits for request from ProcessExecutor to load CNC program.

Activation program in LoadProg:

```c
prog = read_req(LOAD_PROG);
if((MACHINE_MASK & Device) == Machine) {
   MachineProg = req(GAMMA, LOAD_PROG, prog);
}
if(SIMAN_MASK & Device) == Siman) {
   SimanProg = req(SLATHE, LOAD_PROG, "");
}
if((FLEXIS_MASK & Device) == Flexis) {
   timer_start(&LatheTimer);
}
```
It reads the CNC program name to the pointer, *prog*. Device is a enumeration variable selected by the user, which has the option to communicate with either the CNC Machine, Siman, Flexis or any combination. MACHINE_MASK, SIMAN_MASK, FLEXIS_MASK are values assigned to have bitwise addition with the Device selected which gives the respective selection to communicate. Depending on the selection, the respective request is sent to the Machine, Siman or Flexis.

**Transition label for ProgLoaded:**

```plaintext
wait_replies(MachineProg, SimanProg, LatheTimer)
```

The function waits for replies from Machine, Siman, Flexis or any combination.

**Activation program in SendProgLoaded:**

```plaintext
if((MACHINE_MASK & Device) == Machine) {
    clear_reply(MachineProg);
}
if((SIMAN_MASK & Device) == Siman) {
    clear_reply(SimanProg);
}
reply(LOAD_PROG,**);
clear_req(LOAD_PROG);
```

If selected Device is Machine it clears the reply received from machine. If it is Siman, it clears the reply received from Siman, gives reply to ProcessExecutor and clears the request received from ProcessExecutor.

When the reply to the request LOAD_PROG is done, it goes back and waits for another request to load program.
5.2.4 Exec Macro:

Figure 5.5 shows the Exec Macro. This macro communicates with the computer that controls the CNC machine to execute the loaded part if and when required. It waits for a request from ProcessExecutor to execute the part. When it gets the request, it sends the request to the computer that controls the CNC machine, to Siman, a simulation tool, or simulates the execution of the part by delaying for a specific time. When it receives replies, it replies back to the ProcessExecutor.

```
C declarations for Exec Macro:

int MachineRun, SimanRun;
TIMER LatheTimer;

MachineRun  Used to store the transaction number returned by the request sent
             to the CNC machine to execute the part.
```
SimanProg Used to store the transaction number returned by the request sent to Siman to execute the part.

LatheTimer It is a pointer to the timer. It is used when enum Machine is set to Flexis to act for 50 seconds, as if executing the part.

**Activation program in Init:**

timer_set(&LatheTimer, 5000);

Sets the timer to 50 seconds.

**Transition label for LatheExec:**

wait_req(EXEC);

Waits for request from ProcessExecutor to execute the part.

**Activation program in Run:**

cchar *progTimePtr;

progTimePtr = read_req(EXEC);

flx_trace("SimanLathe Program running time : %s\n",progTimePtr);

if((MACHINE_MASK & Device) == Machine) {
    MachineRun = req(GAMMA, EXEC_PROG, "");
}

if((SIMAN_MASK & Device) == Siman) {
    SimanRun = req(SLATHE, EXEC_PROG, progTimePtr);
}

if((FLEXIS_MASK & Device) == Flexis) {
    timer_start(&LatheTimer);
}

First, it gets the execution time for the machine if it is to be executed by Siman. This value is stored in the buffer progTimePtr. Device is an enumeration variable selected by the user, which has the option to communicate with either the CNC Machine, Siman, Flexis or any combination. MACHINE_MASK,
SIMAN_MASK, FLEXIS_MASK are values assigned to have bitwise addition with the Device selected, which gives the respective selection with which to communicate. Depending on the selection, a request is sent to the respective Machine, Siman or Flexis.

**Transition label for LatheDone:**

```plaintext
wait_replies(MachineProg, SimanProg, LatheTimer)
```

The function waits for replies from Machine, Siman, Flexis or any combination.

**Activation program in SendMsg:**

```plaintext
if((MACHINE_MASK & Device) == Machine) {
    clear_reply(MachineRun);
}
if((SIMAN_MASK & Device) == Siman) {
    clear_reply(SimanRun);
}
reply(EXEC,"");
clear_req(EXEC);
```

If selected Device is Machine, it clears the reply received from Machine. If it is Siman, it clears the reply received from Siman. It also sends Reply EXEC_PROG to ProcessExecutor and clears the request received from ProcessExecutor.

When the reply to the request EXEC_PROG is done, it goes back and waits for another request to load the program.
Robot has three macros running in parallel: Reserve, SendAvailable and Exec. Each macro has its own function. The Reserve macro keeps the identification numbers of the ProcessExecutors which are reserving the Robot. Depending on the availability of Robot, the SendAvailable macro assigns the Robot to the ProcessExecutors. The Exec macro executes the Robot. Figure 6.1 shows the Robot Process macro.

![Figure 6.1 Robot Process Macro](image)

### 6.1 C DECLARATIONS

C declarations for Process Macro:

```c
#define MAXRESERVEQSIZE 5

int QueueStep = 0;
int reqQ[MAXRESERVEQSIZE];
```
void addReqQ(int id) {
    if (QueueStep == MAXRESERVEQSIZE) {
        Error_code = MAX_RESERVE_Q_SIZE_EXCEEDED;
        return;
    }
    reqQ[QueueStep] = id;
    QueueStep++;
}

int readReqQ(void) {
    int id, i;
    id = reqQ[0];
    for(i=0; i<=QueueStep; i++) {
        reqQ[i] = reqQ[i+1];
    }
    QueueStep--;
    return(id);
}

QueueStep   It is an Integer used between Reserve and SendAvailable macros
to store the current free place in the reserve request queue. It is
used in addReqQ and readReqQ functions.

reqQ[]      It is the queue for storing the identification numbers of the
ProcessExecutors which are requesting the robot.

addReqQ    This function adds the identification number of the
ProcessExecutor, which is requesting the robot to the end of its
request queue.

readReqQ  This function returns the identification number of the
ProcessExecutor of the first machine and removes it from the
queue.

6.2 PROCESS MACRO

6.2.1 Reserve Macro:

Figure 6.2 shows the Reserve macro. It's function is to add the identification
numbers of the ProcessExecutors that are requesting the machine. When it receives a request for a robot it adds the identification number of the requested actor in the reserve queue and goes back to wait for another request.

Activation program in Init:

```c
int i;
for(i=0, i<MAXRESERVEQSIZE; i++) reqQ[i]=0;
```

Initialize the request queue.

Transition label for Reserve:

```c
wait_cmd(RESERVE);
```

In this transition, it waits for a command from ProcessExecutor reserving the robot.

Activation program in ReqFrom:

```c
int reqFrom;
reqFrom = source_cmd(RESERVE);
```
When it receives a request from the ProcessExecutor, it gets the identification number of the ProcessExecutor by source_cmd function. The received command is cleared, and the identification number is added in the request queue.

6.2.2 SendAvailable Macro:

Figure 6.3 shows the SendAvailable macro. This macro waits for a request for the robot. If a request is present and robot is free, the requesting actor is informed that it is free to use the robot.

**Transition label for MachineAvailable:**

QueueStep > 0
Checks for any request present.

**Activation program in SendAvailable:**

```c
int dest;
dest = readReqQ();
cmd(dest, AVAILABLE, "");
```

It reads the request queue and gets the first ProcessExecutor identification number in queue. Assigns the robot to this ProcessExecutor and sends the command, AVAILABLE, to the ProcessExecutor.

**Transition label for WaitForRelease:**

```c
wait_cmd(RELEASE)
```

When the ProcessExecutor finishes with the use of robot, it releases the robot by
sending command RELEASE.

**Activation program in Clear:**

```plaintext
clear_cmd(RELEASE);
```

In this step it clears the received command RELEASE.

### 6.2.3 Exec Macro:

Figure 6.4 shows the Exec Macro. This macro communicates with the computer that controls the Robot to get or remove a part from the CNC machines. When it receives a request from ProcessExecutor to execute, it sends the request to the computer that controls robot, to Siman, a simulation tool, or simulates executing the robot by delay for the specific time. When it receives the replies, it replies
back to the ProcessExecutor.

C declarations for Exec Macro:

```c
TIMER RobotTimer;
int Source, Dest, Loop;
int MachineSource, MachineDest, SimanSource, SimanDest;
int MachineUnload, MachineLoad, SimanUnload, SimanLoad;
char pick[20], Place[20];
```

RobotTimer  It is a pointer to the timer. It is used when Machine variable is set
to Flexis to act for 20 seconds as if the robot is going to a source
or a destination.

Source  It is an integer to store the source for the robot, where it has to
pick a part.

Dest  It is an integer to store the destination for the robot, where it has
to place a part.

Loop  It is an integer. In Karel system, the positions of machines and
feeders are in a sequence loop; i.e., loop 1 is lathe, loop 3 if feeder,
loop 5 is mill. In Flexis, the identifications numbers are not same;
i.e. mill has 1 and lathe has 2 and different feeders has 5, 6, 7 and
Figure 6.4 Robot Exec Macro
8. To send the exact location for the karel system Loop variable is used to store the location and sent to the robot.

MachineSource, MachineUnload, MachineDest, MachineLoad

To store the transaction numbers returned by the requests sent to robot to go to a source, to pick a part, to go to a destination, to place a part.

SimanSource, SimanUnload, SimanDest, SimanLoad

To store the transaction numbers returned by the requests sent to Siman to go to a source, to pick a part, to go to a destination, to place a part.

**Activation program in Init:**

```c
timer_set(&RobotTimer, 2000);
```

Sets the timer to 20 seconds.

**Transition label for PickPart:**

```c
wait_req(PICK);
```

Waits for a request from ProcessExecutor to pick the part and move it to the destination.

**Activation program in ReadMsg:**

```c
char *infoptr;
infoptr = read_req(PICK);
Source = atoi(infoptr);
strncpy(Pick, infoptr+5, 16);
```
Dest = atoi(infoptr+21);
strncpy(Place,infoptr+26,16);
flx_trace("ROBOT source : %d\n",Source);
flx_trace("ROBOT pick : %s\n",Pick);
flx_trace("ROBOT dest : %d\n",Dest);
flx_trace("ROBOT place : %s\n",Place);

The ProcessExecutor sends the request, PICK, with the parameters source, place
to pick, destination, place to place the part in the format "%4d %15s %4d %15s".
The parameters are read into the required variables.

Activation program in ToSource:

char mbuf[15], sbuf[15];
if((MACHINE_MASK & Device) == Machine) {
    switch(Source) {
    case LATHE:
        Loop = 1;
        break;
    case MILL:
        Loop = 8;
        break;
    case FEEDERL:
        Loop = 3;
        break;
    case FEEDER:
        Loop = 6;
        break;
    }
    sprintf(mbuf,"%-4d\n",Loop);
    MachineSource = req(Delta,GOTO,mbuf);
}
if((SIMAN_MASK & Device) == Siman) {
    switch(Source) {
    case LATHE:
        Loop = 1;
        break;
    case MILL:
        Loop = 8;
        break;
    case FEEDERL:
        Loop = 3;
        break;
    case FEEDER:
        Loop = 6;
        break;
    }
    sprintf(sbuf,"%-4d\n",Loop);
    MachineSource = req(SROBOT,GOTO,sbuf);
}
if((FLEXIS_MASK & Device) == Flexis) {
    timer_start(&RobotTimer);
}
Device is an enumeration variable selected by the user, which has the options to communicate with either the robot, Siman, Flexis or any combination. MACHINE_MASK, SIMAN_MASK, FLEXIS_MASK are values assigned to have bitwise addition with the device selected, which gives the respective selection to with which to communicate. Depending on the selection, a request is sent to the machine, Siman or Flexis. In Karel system the positions of machines and feeders are in a sequence loop; i.e., loop 1 is Lathe, loop 3 is Lathe_Feeder, loop 4 is Lathe_Bin, loop 5 is Mill_Bin, loop 6 is Mill_Feeder and loop 8 is the Mill. Using a switch statement, the exact location is obtained and a request is sent to the required destinations. The parameter is left aligned, because in the karel system the GOTO command and location are separated by a single space. If selected device is Flexis, the timer is started assuming the robot is moving for the specified time.

**Transition label for MoveDone:**

wait_replies(MachineSource, SimanSource, RobotTimer)

The function waits for replies from machine, Siman, Flexis or any combination.

**Activation program in Pick:**

```c
char buf[30];
if((MACHINE_MASK & Device) == Machine) {
    clear_reply(MachineSource);
    sprintf(buf,"%-20s",Pick);
    MachineUnload = req(DELTA,PICK,buf);
}
if((SIMAN_MASK & Device) == Siman) {
    clear_reply(SimanSource);
```
SimanUnload = req(SROBOT, PICK, "");
}
switch(Source) {
    case FEEDERL:
        cmd(FEEDER3, UNLOAD, "");
        break;
    case FEEDERM:
        cmd(FEEDER2, UNLOAD, "");
        break;
}
The received replies to the request GOTO are cleared, and the request is sent to pick the part. In Karel system, the part location is a string with a single space between PICK command and location, so with the PICK command, the location is left justified. For Siman, the locations are not sent, since the Siman doesn’t have locations. If robot is picking a raw part from the feeder, a command is sent to the respective feeder actor that robot is picking a part from, so that the feeder actor can send the command for the next part to the Scheduler.

Transition label for Done:

wait_replies(MachineLoad, SimanLoad, RobotTimer)
The function waits for replies from Machine, Siman, Flexis or any combination.

Activation program in ToDest:

char mbuf[15], sbuf[15];
if((MACHINE_MASK & Device) == Machine) {
    clear_reply(MachineUnload);
    switch(Source) {
        case LATHE:
            Loop = 1;
            break;
        case MILL:
            Loop = 8;
            break;
        case BINL:
            Loop = 4;
            break;
        case BINM:
Loop = 5;
break;
}
sprintf(mbuf,"%-4d",Loop);
MachineSource = req(DELTA,GOTO,mbuf);
}
if((SIMAN_MASK & Device) == Siman) {
clear_reply(SimanUnload);
switch(Source) {
  case LATHE:
    Loop = 1;
    break;
  case MILL:
    Loop = 8;
    break;
  case BINL:
    Loop = 4;
    break;
  case BINM:
    Loop = 5;
    break;
}
sprintf(sbuf,"%-4d",Loop);
MachineSource = req(SROBOT,GOTO,sbuf);
}
if((FLEXIS_MASK & Device) == Flexis) {
timer_start(&RobotTimer);
}

The received replies to the request PICK are cleared, and the request is sent to move the robot to a destination. If selected Device is Flexis, the timer is started assuming the robot is moving for the specified time.

**Transition label for MoveDone:**

wait_replies(MachineDest, SimanDest, RobotTimer)

The function waits for replies from machine, Siman, Flexis or any combination.

**Activation program in WaitForLoad:**

if((MACHINE_MASK & Device) == Machine) {
clear_reply(MachineDest);
}
if((SIMAN_MASK & Device) == Siman) {
clear_reply(SimanDest);
}
reply(PICK,"");  
clear_req(PICK);
The received replies to the request GOTO are cleared. When the Robot is moved to the destination, a reply is sent to the ProcessExecutor.

**Transition label for PlacePart:**

```c
wait_req(PLACE);
```

It waits for the request from the ProcessExecutor to place a part in the machine.

**Activation program in Place:**

```c
char buf[30];
if((MACHINE_MASK & Device) == Machine) {
    sprintf(buf, "%-20s", Place);
    MachineLoad = req(Delta, PLACE, buf);
}
if((SIMAN_MASK & Device) == Siman) {
    SimanLoad = req(SROBOT, PLACE, buf);
}
```

Request is sent to place the part. In Karel system, the part location is a string with a single space between PICK command and location, so with the PICK command, the location is left justified. For Siman, the locations are not sent, since the Siman doesn’t have locations.

**Transition label for Done:**

```c
wait_replies(MachineUnload, SimanUnload, RobotTimer)
```

The function waits for replies from Machine, Siman, Flexis or any combination.

**Activation program in Loaded:**

```c
if((MACHINE_MASK & Device) == Machine) {
    clear_reply(MachineLoad);
}
```
if((SIMAN_MASK & Device) == Siman) {
    clear_reply(SimanLoad);
}
reply(PLACE,"");
clear_req(PLACE);

The received replies for the request PLACE are cleared and a reply is sent to the ProcessExecutor.

When the robot finishes placing the part, it goes back and waits for another request to move a part.
CHAPTER VII

FEEDERS

Feeders are just mechanical fixtures to hold parts. The Feeder Actor is used to track the number of parts available and to negotiate with the Scheduler to get its part processed. When it has a part, it sends a command to the Scheduler to process the part and waits for the Robot to pick the part. When the Robot picks the part from the Feeder, it sends another command to the Scheduler to process the next part. When all parts are over, it waits for reload. Figure 7.1 shows the Feeder Process Macro.

7.1 C DECLARATIONS FOR PROCESS MACRO

C declarations for Process Macro:

```c
int NumParts;
NumParts It is an integer to store the current number of parts that are available in the feeder for processing.
```

7.2 PROCESS MACRO

Transition label for LoadParts:

```
LoadParts == Yes
Flexis uses enumeration variables to interact with the user. After the user loads parts into the physical feeder, he sets the LoadParts variable ‘Yes’. This transition becomes true and the parts are processed.
```
Activation program in Load:

\[ \text{NumParts} = \text{INIT\_NUM\_PARTS}; \]

The initial number of parts is initialized to INIT\_NUM\_PARTS, which is declared in flexis.dcls.

Activation program in SendInfo:

\[ \text{cmd}(\text{SCHEDULER}, \text{FEEDERINFO}, ""); \]

A command is sent to the Scheduler to process the part.
Transition label for PickUp:

\texttt{wait\_cmd(UNLOAD);} 
Waits for the command UNLOAD from the ROBOT actor, which means the robot 
is picking up the part from the Feeder. The Feeder can send a command to the 
Scheduler to process the next part.

Activation Program in DecNumParts:

\texttt{clear\_cmd(UNLOAD); NumParts--;} 
The received command is cleared, and since Robot is picking a part, the number 
of parts in the feeder is decremented.

Transition label for PartAvailable:

\texttt{NumParts > 0} 
The transition is connected to SendInfo step. If the parts are available in the 
feeder, it sends next part information to the Scheduler.

Transition Label for PartsNotAvailable:

\texttt{NumParts == 0} 
If all the parts in the feeder are completed this transition becomes true, causing 
the actor to wait for more parts to be reload.

Activation program in WaitReload:

\texttt{LoadParts = No;}

LoadParts is set to 'No' so that the actor can wait for the user to set it to true after he loads another batch.

**Transition label for Reload:**

LoadParts == Yes

Waits for the user to reload the parts.

**Activation program in Reloaded:**

NumParts = INIT_NUM_PARTS;

The number of parts is reset.
CHAPTER VIII

BIN

Bin is an actor to represent a physical system which hold the finished parts. The robot places the finished parts in the Bin. It can be used to count the number of finished parts from different feeders. Figure 8.1 shows the simple grafcet representation of Bin.

8.1 PROCESS MACRO

Transition label for Load:

wait_cmd(LOAD)

Waits for a command LOAD from the robot. The robot is placing a part in Bin.
Activation program in Loaded:

clear_cmd(LOAD);

Clears the received command. The step is connected back to IN step, where it waits for another load command.
CHAPTER IX

SCHEDULER

The function of the Scheduler is to read the process plans and assign the jobs to the ProcessExecutors. The maximum utilization of machines depends on the scheduling of jobs efficiently. To see that the machines are utilized to the maximum extent, the scheduler keeps track of the process step each ProcessExecutor is executing. If ProcessExecutor wants a new part it assigns the part that doesn’t clash with the process plan that is being done by the other ProcessExecutor. This is done using a bitwise addition of available resources required for the feeder parts and the current resources needed for the part that is being executed. To use bitwise addition MILL actor is #defined as 1, LATHE actor is #defined as 2. Also Mill resource is #defined as 1, Lathe resource is #defined as 2, Mill and then Lathe resource is #defined as 4, and Lathe and then Mill resource is #defined as 8.

As shown in Figure 9.1, the Scheduler has three programs running in parallel. One program waits for the part information from the feeders; one program updates the resources used by the ProcessExecutors, and the other program schedules the parts to the ProcessExecutors.
9.1 C DECLARATIONS FOR PROCESS MACRO

C declaration in Process Macro:

```c
#define MAXQSIZE 10
FILE *PP_file;
int SelectQ;
bool ScheduleWIP;

struct resource {
    int resAssigned;
    }ProcessExecutor[2];

struct FeederQ {
    int feederID;
    int resNeeded;
    }feederInfo[MAXQSIZE];

int get_resource_needed(int step) {
    int resource = 0, res;
    int first_step = step;

    if(PP[step].machine == BINM || PP[step].machine == BINL)
        return(ANY);
    while(!(PP[step].machine == BINM || PP[step].machine == BINL)) {
        resource = (resource | PP[step].machine);
    }
```
switch (resource) {
    case MILL:
        res = MILLRES;
        break;
    case LATHE:
        res = LATHERES;
        break;
    case 3:
        if (PP[first_step].machine == MILL) {
            res = MILLLATHERES;
            break;
        }
        if (PP[first_step].machine == LATHE) {
            res = LATHEMILLRES;
            break;
        }
    }
    return(res);
}

int add_feeder_reqQ(int ID, int res)
{
    feederinfo[SelectQ].feederID = ID;
    feederInfo[SelectQ].resNeeded = res;
    SelectQ++;
    return 0;
}

char *search_feederQ(int res)
{
    int i, j;
    char buf[20];
    for(i = 0; i < MAXQSIZE; i++) {
        if(((feederInfo[i].resNeeded & res) != 0) {
            sprintf(buf, "%4d %4d", feederInfo[i].feederID,
                    feederInfo[i].resNeeded);
            for(j = i; j <= min(SelectQ, MAXQSIZE); j++) {
                feederInfo[j].feederID = feederInfo[j+1].feederID;
                feederInfo[j].resNeeded = feederInfo[j+1].resNeeded;
            }
            SelectQ--;
            return(buf);
        }
    }
    sprintf(buf, "%4d %4d", NOT_AVAILABLE, NOT_AVAILABLE);
    return(buf);
}

PP_file Pointer to the currently opened process plan file.
SelectQ This is an integer to store the current queue position.
ScheduleWIP This a boolean variable. Whenever the Scheduler receives
information from feeders or from the ProcessExecutors, it is set to True. When the ScheduleWIP becomes true the ScheduleWIP step assigns the job to the ProcessExecutors.

struct resource

Stores the current step that is being executed by the ProcessExecutors; i.e., the current resource that is being used by the ProcessExecutors.

struct FeederQ

Stores the feeder identification number that has request to execute its part and the resources required for that part.

get_resource_needed

This function returns the resources needed for the part from the current step of the process plan. From the given step, it reads the process plan and does bitwise union of machines required (which has 1 for MILL and 2 for LATHE actors). If it requires only mill resource it returns 1, if it requires only lathe resource it returns 2, and it returns 4 or 8 depending on the current step; i.e., if the current step is mill, then it requires mill and then lathe, otherwise lathe and then mill.

add_feeder_reqQ

This function adds the feeder identification number and the resource required for the part in the feederInfo structure.

search_feederQ

For the given resources that a ProcessExecutor can have, the function returns the feeder ID and the resource needed to the part which is first in the queue. This is implemented
by a bitwise addition of the resources it can have with the resNeeded for the feeders.

9.2 PROCESS MACRO

**Activation program in Init:**

```c
int i;

ProcessExecutor[0].resAssigned = 0;
ProcessExecutor[1].resAssigned = 0;
ScheduleWIP = P;

for(i=0;i<MAXQSIZE;i++) {
  feederInfo[i].feederID = 0;
  feederInfo[i].resNeeded = 0;
}
int_mach_map();
```

In this step the variables are initialized.

**Transition label for FeederInfo:**

```c
wait_cmd(FEEDERINFO)
```

The transition waits for the command from the FEEDERs to execute its part.

**Activation program in FeederMsg:**

```c
int feeder, feederId, resInfo;

feeder = source_cmd(FEEDERINFO);
clear_cmd(FEEDERINFO);
switch(feeder) {
  case FEEDER1:
    PP_file = fopen ("/usr/local/devel/flexis/plans/
                     feeder1plan", "r");
    get_pp(PP_file);
    fclose(PP_file);
    feederID = FEEDER1;
    resInfo = get_resource_needed(1);
    add_feeder_reqQ(feederID,resInfo);
    break;
  case FEEDER2:
    PP_file = fopen ("/usr/local/devel/flexis/plans/
                     feeder2plan", "r");
```
get_pp(PP_file);
fclose(PP-file);
feederID = FEEDER2;
resInfo = get_resource_needed(1);
add_feeder_reqQ(feederID,resInfo);
break;
case FEEDER3:
    PP_file = fopen("/usr/local/devel/flexis/plans/feeder3plan", "r");
    get_pp(PP_file);
    fclose(PP_file);
    feederID = FEEDER3;
    resInfo = get_resource_needed(1);
    add_feeder_reqQ(feederID,resInfo);
    break;
case FEEDER4:
    PP_file = fopen("/usr/local/devel/flexis/plans/feeder4plan", "r");
    get_pp(PP_file);
    fclose(PP_file);
    feederID = FEEDER4;
    resInfo = get_resource_needed(1);
    add_feeder_reqQ(feederID,resInfo);
    break;
}
ScheduleWIP = T;

The feeder identification is obtained using source_cmd. Depending on the feeder, the corresponding feeder plan file is opened and the process plan is read. Since it receives a new part information, the resources needed for the part is obtained from step number 1. The feeder ID and the resources needed for the part is added in the feeder information queue. The ScheduleWIP variable is set to true.

**Transition label for ResourceNeeded:**

```
wait_cmd(RESOURCENEEDED)
```

Transition waits for the command from the ProcessExecutor, which indicates the resources they need to complete the process plan.

**Activation program in AssignWIP:**

```
int source, resource;
char *resptr;
```
resptr = read_cmd(RESOURCENEEDED);
resource = atoi(resptr);
source = source_cmd(RESOURCENEEDED);
clear_cmd(RESOURCENEEDED);

switch(source) {
    case PROCESSEXECUTOR1:
        ProcessExecutor[0].resAssigned = resource;
        break;
    case PROCESSEXECUTOR2:
        ProcessExecutor[1].resAssigned = resource;
        break;
}
ScheduleWIP = T;

This step updates the current step being executed by the ProcessExecutor. It reads the resources required for the current step from the received command parameters. From the source_cmd function, it gets the ProcessExecutors ID and updates its resource assigned variable. It sets ScheduleWIP variable to true, because if a ProcessExecutor is idle, it can be checked again to assign a part with the updated assigned resources.

**Transition label for ScheduleWIP:**

ScheduleWIP

Transition waits for ScheduleWIP variable to become true. It can become true whenever Scheduler receives a part information or whenever a ProcessExecutor sends the current resources needed for it.

**Activation program in ScheduleWIP:**

```c
int select, other, resources;
char buf[10];
char *ptr;
int feederID, resAssign;

for(select = 0; select < 2; select++) {
    if(PROCESSEXECUTOR[select].resAssigned == ANY) {
        if(select == 0) other = 1;
    }
```
if(select == 1) other = 0;
switch(ProcessExecutor[other].resAssigned) {
    case ANY:
        resources = ANY;
        break;
    case MILLRES:
        resources = (LATHERES | LATHEMILLRES);
        break;
    case MILLLATHERES:
        resources = LATHERES;
        break;
    case LATHEMILLRES:
        resources = MILLRES;
        break;
    case LATHERES:
        resources = MILLRES | MILLLATHERES);
        break;
    case 0:
        resources = 0;
        break;
}

ptr = search_feederQ(resources);
feederID = atoi(ptr);
resAssign = atoi(ptr+5);
if(feederID == NOT_AVAILABLE) break;
if(select == 0) {
    ProcessExecutor[0].resAssigned = resAssign;
    sprintf(buf, "%4d", feederID);
    cmd(PROCESSEXECUTOR1, EXEC, buf);
}

if(select == 1) {
    ProcessExecutor[1].resAssigned = resAssign;
    sprintf(buf, "%4d", feederID);
    cmd(PROCESSEXECUTOR2, EXEC, buf);
}
}

ScheduleWIP = F;

ScheduleWIP is the important step that assigns the parts, such that the machines have maximum utilization. It assigns a part to the ProcessExecutor by looking at the part that is being executed by the other ProcessExecutor. If the ProcessExecutor needs a new part, it checks the other ProcessExecutor and the resources required for it. If the resource required for the other ProcessExecutor is ANY, then the current ProcessExecutor can also have any part, or else it assigns a part that doesn’t clash with the other ProcessExecutor WIP.
9.3 SCHEDULING CLASHES AND HOW IT IS SOLVED

This section describes the clashes that can occur when assigning the parts to the ProcessExecutors and how it is resolved with the Scheduling principle.

1. Suppose a ProcessExecutor is executing a Mill resource part, and if another ProcessExecutor is assigned, a Mill resource or MillLathe resource part, then it has to wait until the mill machine is free, which reduces the utilization of lathe machine and robot. This situation is avoided by assigning a Lathe resource or LatheMill resource part to the other ProcessExecutor.

2. Suppose a ProcessExecutor is executing a LatheMill resource part, and if another ProcessExecutor is assigned, a Lathe resource part will reduce the utilization of Mill machine, or if another ProcessExecutor is assigned a MillLathe resource part, then there is a deadlock situation in which the robot cannot move a Mill part to lathe machine or Lathe part to mill machine, since both the machines are busy. This situation is avoided by assigning only a Mill resource part to the other ProcessExecutor.

3. Suppose there are only MillLathe resource and LatheMill resource parts to be processed, then the Scheduler is going to assign only one type of resource part to both the ProcessExecutors, until all the parts are completed. Then it will assign the other type of resource parts to the ProcessExecutors, until they are finished. This is done to have more utilization of machines.
PROCESSEXECUTOR

ProcessExecutor is a logical system that is present in the cell controller. It's function is to get a part and complete all the process steps in the process plan. Based on negotiation concepts, the ProcessExecutor informs the Scheduler that it is available to process a new part. When the Scheduler assigns a part, the ProcessExecutor reads the process plan for the part and executes the steps involved in it. It reserves the required machines for each process step, and gets the process plan implemented. Reserving machines for each step is done since common resources are to be shared and to avoid deadlocks. The resources are released when they are no longer needed to make it available for other ProcessExecutors. Since only two CNC machines and a robot are available, first a machine is reserved. When the machine is available the robot is reserved, and when the robot is available, the part is moved and executed. Since the robot is a shared resource, it is reserved after getting the CNC machine to avoid deadlock situations. A deadlock situation can arise if the Robot is reserved first. For example, consider the situation in which ProcessExecutor1 executes a part in mill and the part has to be moved to lathe. Further assume ProcessExecutor2 has another part in the lathe, which is to moved to the bin. If ProcessExecutor1 finishes first and it grabs the robot, then there is no way of moving a part to the lathe, since lathe still has a part, and the lathe part can't be removed, since robot is reserved by ProcessExecutor1. These
Figure 10.1 ProcessExecutor Process Macro
situations are avoided by first reserving the CNC machines and then reserving the shared resources. The ProcessExecutors can execute programs in parallel to gain more efficiency. For example, the required CNC program can be loaded to a machine and simultaneously the part is moved to the CNC Machine. Once all the process steps are done the ProcessExecutor becomes free and makes itself available to execute another part. Figure 10.1 shows the ProcessExecutors Process Macro.

10.1 FORMAT OF A PROCESS PLAN

The format for the process plan looks like

#This is for piston

Name:piston

#Each line should have the following format

#<Machine>:<place location>,<program>,<program time>,<pick location>

Lathe: lathe_insert ,e:piston.prg ,10 ,lathe_pick

The # represents a commented line which is ignored while reading the process plan. Each field is updated in the respective fields in the PP array by reading each line at a time. Any one of this fields may be blank. All tabs and spaces are ignored and all letters are converted to upper case letters.
Name: Describes the name of the part.

Machine: Describes the machine from which the part has to be transferred. It can be a feeder1, feeder2, feeder3, feeder4, bin, mill or lathe.

Place Location: The location in which the part has to be placed.

program: The program that has to be loaded to execute the part.

program time: The time required for the part to execute. This is needed to send to Siman for animating the execution of part for the specified time.

pick location: The location from which the part has to be picked up.

The ProcessExecutor reads the complete process plan and executes each line at a time. The routine to read process plan is explained in Appendix C.

10.2 C DECLARATIONS

C declarations for process macro:

```c
FILE *PP_file;

int MachineAssigned, Step;
int MachineExec, GetPart, LoadPart, LoadProg;
bool Prog;

int get_resource_needed(int step)
{
    int resource = 0, res;
    int first_step = step;

    if(PP[step].machine == BINM || PP[step].machine == BINL)
        return(ANY);
    while(!(PP[step].machine == BIN || PP[step].machine == BINL) {
        (resource = resource | PP[step].machine);
        step++;
    }
    switch(resource) {
        case MILL:
            res = MILLRES;
```
break;
case LATHE:
    res = LATHERES;
    break;
case 3:
    if(PP[first_step].machine == MILL) {
        res = MILLLATHERES;
        break;
    }
    if(PP[first_step].machine == LATHE) {
        res = LATHEMILLRES;
        break;
    }
return(res);
}

PP_file is the file pointer to the process plan.

MachineAssigned Stores the current machine being used; i.e. it can be Mill, Lathe or Bin.

Step Stores the current step in the process plan.

MachineExec Stores the transaction number returned by the request sent to execute the machine.

GetPart Stores the transaction number returned by the request sent to Robot to get a part.

LoadPart Stores the transaction number returned by the request sent to Robot to load a part.

LoadProg Stores the transaction number returned by the request sent to Robot to load program in the machine.

Prog A flag indicating whether load program request is sent to the machines or not.

get_resource_needed The deadlock algorithm needs to know both the current and next resources. This function gets the resources needed for the current process plan step. If the given step in the
process plan is BIN, it returns the resource required as ANY, i.e., NONE. To have bit wise addition, MILL resource is \#defined as 1 and LATHE resource is \#defined as 2. For the remaining steps in the process plan, the required machine for each step is bitwise added to get 1, 2, or 3. If it is 1, it represents only Mill resource. If it is 2 it represents only Lathe resource. If it is 3, then the initial resource that is required is determined from the process plan.

10.3 PROCESS MACRO

Activation program in Init step:

```c
Step = 0;
init_mach_map();
```

Step variable is used to store the current step in the executing process plan. `init_mach_map` is a function declared in `flexis.dcls`, which has the a name for each \#defined number. This is used in the trace windows to have more readability.

Activation program in Process_Send_available:

```c
char buf[10];
sprintf(buf, "%d", ANY);
cmd(SCHEDULER, RESOURCENEEDED, buf);
flx_dtrace("Executor 1: Available\n");
```

In this step a command is sent to scheduler informing that it is available to execute a new part. `flx_dtrace` is Flexis routine to display the messages in device
trace window; i.e., a type of debug message to know the state of the process.

**Transition label for WaitCmd:**

`wait_cmd(EXEC);`

The transition waits for command EXEC from the Scheduler. Whenever there is a part to be processed, the Scheduler sends a EXEC command to the ProcessExecutors.

**Activation program in GetProcPlan:**

```c
char* feederPtr;
int feeder, feederNum;

feederPtr = read_cmd(EXEC);
feeder = atoi(feederPtr);
clear_cmd(EXEC);

switch(Feeder) {
    case FEEDER1:
        PP_file = fopen("feeder1plan","r");
        get_pp(PP_file);
        fclose(PP_file);
        break;
    case FEEDER2:
        PP_file = fopen("feeder2plan","r");
        get_pp(PP_file);
        fclose(PP_file);
        feederNum = FEEDERM;
        break;
    case FEEDER3:
        PP_file = fopen("feeder3plan","r");
        get_pp(PP_file);
        fclose(PP_file);
        feederNum = FEEDERL;
        break;
    case FEEDER4:
        PP_file = fopen("feeder4plan","r");
        get_pp(PP_file);
        fclose(PP_file);
        break;
}

flx_trace("Executor 1: Began Processing %s from %s\n", PP_name,
           machine_name(feederNum));
```

It reads the feeder identification number from the received command. Depending
on the feeder, the process plan file is opened, read using get_pp function and the
file is closed.

NOTE: The feederNum is assigned only in FEEDER2 and FEEDER3 because the
project was demonstrated with only Mill resource and Lathe resource parts. Also
FEEDER2 and FEEDER3 names are declared in Machine_map structure of PP-
Plan library.

**Activation program in ReserveMachine:**

```java
switch(PP[step+1].machine) {
    case MILL:
        cmd (MILL, RESERVE, "");
        MachineAssigned = MILL;
        break;
    case LATHE:
        cmd (LATHE, RESERVE, "");
        MachineAssigned = LATHE;
        break;
    case BINM:
        MachineAssigned = BIN;
        break;
    case BINL:
        MachineAssigned = BIN;
        break;
}
```

Step+1 has the information on the next place the part is to be moved. The
required machine is reserved. The MachineAssigned variable is set to the machine
so that the transition MachineAvailable can wait for the command, AVAILABLE,
from the machine Actor.

**Transition label for MachineAvailable:**

```java
((MachineAssigned == MILL || MachineAssigned == LATHE) &&
 wait_cmd(AVAILABLE) || MachineAssigned == BIN
```

If the process plan step is Mill or Lathe, it waits for the command, AVAILABLE,
from the respective CNC Machine Actor. If the Process plan step is Bin, there is no needed to reserve or wait for command from Bin Actor.

**Activation program in ReserveRobot:**

```plaintext
if((MachineAssigned == MILL) || (MachineAssigned == LATHE))
    clear_cmd(AVAILABLE);
    cmd(ROBOT, RESERVE, "");
```

It clears the command received from the Machine and sends a command to the Robot Actor to reserve the robot.

**Transition label for RobotAvailable:**

```plaintext
wait_cmd(AVAILABLE);
```

Waits for the command, AVAILABLE, from the Robot Actor.

**Activation program in GetPart:**

```plaintext
int source, dest;
char buf[55];
clear_cmd(AVAILABLE);
source = PP[Step].machine;
dest = PP[Step+1].machine;
sprintf(buf, "%4d %15s %4d %15s", source, PP[Step].pick, dest,
       PP[Step+1].place);
GetPart = req(ROBOT, PICK, buf);
flx_dtrace("Executor 1: Robot moving from \%s to \%s\n",
           machine_name(source), machine_name(dest));
```

As of now it is assumed that all the feeders are at the same position. The current step and next step becomes the source and destination. The source, the place to pick, destination, and the place to put the part is copied to a buf in the format "%4d %15s %4d %15s" and the request is sent to the robot to pick the part. The
flx_dtrace displays the action in the device trace window.

**Activation program in ReleaseMachine:**

```c
if((Step > 0) && (PP[Step].machine != PP[Step+1].machine)) {
    switch(PP[Step].machine) {
        case MILL:
            cmd(MILL, RELEASE, "");
            break;
        case LATHE:
            cmd(LATHE, RELEASE, "");
            break;
    }
}
```

If the current step and the previous step in the process plan uses the same machine then the Machine is not released; otherwise the previous step machine is released. If the Step is zero, there is no need to release a Feeder since it only holds raw parts.

**Transition label for PartArrived:**

```c
wait_reply(GetPart);
```

Waits for the reply from the Robot Actor that part has been moved to the destination.

**Activation program in Sync1:**

```c
clear_reply(GetPart);
```

Clears the received reply and waits till loading program is also done.

**Activation program in LoadProg:**

```c
if((PP[Step+1].machine != FEEDER) && (PP[Step+1].machine != BIN)) {
    Prog = TRUE;
```
LoadProg = req(PP[Step+1].machine, LOAD_PROG, PP[Step+1].prog);
else  Prog = FALSE;

If the current step is other than a Feeder or Bin than a CNC program has to be loaded. A variable prog is set to true and a request is sent to the respective machine to load the program. If the destination step is feeder or bin, the loading program is set to false, so that in the transition, loadProgDone, there is no need to wait for the program to be loaded.

**Transition label for LoadProgDone:**

```
((wait_reply(LoadProg) && Prog == TRUE) || Prog == FALSE)
```

Waits for the reply from the respective CNC machine that the program is loaded. If the destination step is other than the CNC machine there, is no need to wait for the program to be loaded.

**Activation program in Sync2:**

```
if (Prog == TRUE)
    clear_reply(LoadProg);
```

Clears the received reply from the machines and waits for the part to be moved to the destination.

**Activation program in LoadPart:**

```
LoadPart = req(ROBOT, PLACE, "*");
```

When the part is moved to the destination and the program is loaded, the Robot is requested to place the part in the machine. This is done after the program is
loaded. If the Robot is placing a part when the program is being loaded, the CNC machine will generate errors.

**Transition label for Loaded:**

```plaintext
call_reply(LoadPart);
```

Waits for the reply from the Robot Actor that the part is loaded.

**Activation program in ReleaseRobot:**

```plaintext
call_reply(LoadPart);
cmd(ROBOT, RELEASE, "");
```

The received reply for the load part is cleared and the robot is released so that it can be made available to the other ProcessExecutor.

**Activation program in RunMachine:**

```plaintext
switch(PP[Step].machine) {
  case MILL:
    MachineAssigned = MILL;
    MachineExec = req(MILL, EXEC, ");
    flx_dtrace("Executor 1: Milling the %s\n", PP_name);
    break;
  case LATHE:
    MachineAssigned = LATHE;
    MachineExec = req(LATHE, EXEC, ");
    flx_dtrace("Executor 1: Turning the %s\n", PP_name);
    break;
  case BINM:
    MachineAssigned = BIN;
    break;
  case BINL:
    MachineAssigned = BIN;
    break;
}
```

Depending on the type of machine, a request is sent to the corresponding machine to execute the part. If the machine is Bin, there is no need to send a request. The MachineID is stored in the variable MachineAssigned. The action is displayed in
the device trace window.

**Transition label for MachineDone:**

\[ (((\text{MachineAssigned} = \text{MILL} \mid \text{MachineAssigned} = \text{LATHE}) \ \&\& \ \text{wait_reply(MachineExec)}) \mid \mid \text{MachineAssigned} = \text{BIN}) \]

If the process step machine is Mill or Lathe, it waits for the reply from the Machine. If the step is Bin it will not wait.

**Activation program in IncProcPlan:**

```java
Step++;
if(((\text{MachineAssigned} = \text{MILL}) \mid (\text{MachineAssigned} = \text{LATHE}))
clear_reply(MachineExec);
switch(\text{PP[step].machine}){
  \text{/* Check next step is Bin */}
  \text{case BINM:}
    \text{MachineAssigned = BINM;}
    \text{break;}
  \text{case BINL:}
    \text{MachineAssigned = BINL;}
    \text{break;}
}
```

The process plan step is incremented and the received reply from the Machine Actor is cleared. If the next process step is bin then it is stores the bin identification in variable, MachineAssigned.

**Transition label for SameMachine:**

\[ \text{PP[Step].machine} = \text{PP[Step+1].machine} \]

If the current step and the next step are the same, there is no need to reserve the machine again; it can load back the part in the same machine.
Transition label for ProcPlanNotDone:

\[(PP[Step].machine \neq PP[Step+1].machine) \&\& (MachineAssigned \neq BIN)\]

If the current step and the next step is not the same, then the process plan would have been completed or the machine required for the next process step is different from the current step. If the next process step is not bin, it must be a CNC machine. It implies the process plan is not yet finished, and the next machine is to be reserved.

Transition label for ProcPlanDone:

\[(PP[Step].machine \neq PP[Step+1].machine) \&\& (MachineAssigned == BIN)\]

If the current step and the next step is not same, then the process plan would have been completed or the machine required for the next process step is different form the current step. If the next process step is bin it implies that the process plan be done, and the ProcessExecutor informs the Scheduler that it is free.

Activation program in ResourceNeeded:

```c
int res;
char buf[10];
res = get_resource_needed(Step+1);
if (res \neq ANY) {
    sprintf(buf, "%d", res);
    cmd(SCHEDULER, RESOURCENEEDED, buf);
}
```

The resources that are needed for the remaining process plan steps are obtained. If the resources needed are other than ANY, a command is sent to the Scheduler to update the resources that are required for the ProcessExecutor in the next step of the process plan. If the next step is moving the part to bin, the
get_resource_needed function returns ANY. Since a command is sent in SendAvailable step, there is no need to send before the part is moved into Bin.

**Activation program in ReInit:**

```
Step = 0;
```

The variable Step is initialized to zero to restart the process plan from the beginning.
CHAPTER XI
IMPLEMENTATION OF CELL CONTROLLER

11.1 COMPONENTS OF CIMLAB

The components in cimlab are described below. It contains a EMCO VMC-100 CNC mill, EMCOTURN 120P CNC lathe, GMFanuc karel S-10 system, feeders and bin.

11.1.1 CNC Machines:

The EMCO VMC-100 milling machine operates on x,y,z axis with 10 toolhandles and a digital port to control the machine tool. It is operated with emcotronic TM02 CNC controller, which has a control panel with LCD display and RS232 serial port to load and execute the programs. The EMCOTURN 120P turning machine operates on x,z axis with a digital port to control the machine tool. It is operated with emcotronic TM02 CNC controller, which has a control panel with LCD display and RS232 serial port to load and execute the programs.

11.1.2 Karel System:

The KAREL system include a robot mechanical unit, controller hardware, and KAREL system software. The S-10 robot mechanical unit is a six-axes articulated robot powered by brushless AC servomotors. The motions of all axis are controlled simultaneously. The KAREL R-H controller includes operator panel, teach pendant and CRT/KB, RS232 ports. The operator panel is equipped with lights, buttons and key
switches and is located on the front of the controller cabinet. The teach pendant is equipped with a keypad and LCD screen. It is connected to the controller by a cable that plugs into the shared RAM board inside the controller. The CRT/KB port is connected to a computer terminal to perform file operations, display status and diagnostic information, and to enter, translate, debug programs.

11.1.3 Feeders & Bin

The feeders are mechanical units to get a part into the position that to be picked by the robot. The bin is just a bin in which to place the finished parts.

11.1.4 Computer:

The lab is equipped with a Sun Sparc workstation and three IBM compatible computers. The Sun Sparc workstation is the cell controller; it uses a UNIX operating system and executes programs developed using Flexis toolbox. Two PCs are intel 80286 microprocessor based systems using the DOS operating system. These computers are used to interface to the CNC machines. They use a digital I/O port to control the machine tool unit and a RS232 port to communicate with CNC emcotronic02 controller to load and execute the programs. The third computer is intel 80486 microprocessor based system operating on DOS. This computer is used to communicate with the KAREL R-H controller by using two RS232 ports. One port is used for file operations and the other port is used for off-line programming. The three DOS computers use Ohio University developed menu-driven software to control the CNC machines and robot and
also to simulate them. The PC programs can be referred from the documentation "Cimlab Server, PC programs and CNC Machines" documented by Mr. SenthilKumar. M.

11.1.5 Network:

Figure 11.1 shows the network connections between the workstation, PC's and the outside world.

![Diagram of network connections](image)

**Figure 11.1 Computer Connections**

The PC's uses TCP/IP software to connect to the network and to communicate with the outside world. The messages between SPARC workstation and the PC's are passed using their IP address. This type of connection gives more flexibility to load and execute the programs from the outside world. A server developed by Ohio University is used to communicate between the PC's and workstation. The server is based on UNIX sockets; it opens a socket for each client, accepts the incoming messages and passes the
messages to the requested destination. The Server program can be referred from "Cimlab Server, PC programs and CNC Machines", documented by Mr. SenthilKumar. M.

11.1.6 Simulation tools:

The Cimlab is animated with CINEMA, a software package and animation is linked to the simulation language, Siman. The Cell Controller talks to the simulation tool and executes the Cimlab animation. The Simulation programs can be referred from "Object-Oriented Simulation at RealTime for a Manufacturing facility", the thesis documentation of Mr. Somanath Lanka.

11.1.7 Ports and Connections:

Figure 11.2 shows the connections between the PC's, controllers and the machines. The 486 computer is connected to the KAREL controller with CRT/KB and RS232 port. The 486 computer uses serial port 1 and Ohio University developed software to communicates with KAREL system software. Also, the 486 computer uses serial port 2 to act as a KAREL controller for startup and to perform the file operations. The KAREL controller communicates with the robot mechanical unit and also with the machine tools. The 286 computers uses serial port 1 (RS232) and Ohio University developed software to communicate with machine tools. Karel system provides Input/Output (I/O) system to interface KAREL through user-defined I/O signals, system defined I/O signals and communication ports. The karel DIN/DOUT signals are used to interface with CNC.
Figure 11.2 Connections
machines to open doors, release chuck, etc. when the Robot comes to pick or place a part. To interface the computer with CNC machines, a general purpose TTL interface is provided through a 48 channel digital I/O interface board. Through a interface board built by Ohio University, the computer digital signals are converted to the X22 standards. The CNC machines use X22 standard to interface with other peripherals.

11.2 ACTORS NEEDED FOR CIMLAB

**Mill Actor:**

Mill actor has to assign the mill machine to ProcessExecutors, load programs and execute the part. This is implemented by having four programs running in parallel: Reserve, SendAvailable, LoadProg, and Exec macro's.

**Lathe Actor:**

Lathe actor has to assign the lathe machine to ProcessExecutors, load programs and execute the part. This is implemented by having four programs running in parallel: Reserve, SendAvailable, LoadProg, and Exec macro's.

**Robot Actor:**

Robot Actor has to assign the robot to the ProcessExecutors and move the part to the desired place. This is implemented by having three program macro's running in parallel: Reserve, SendAvailable, Exec macro's.
**Feeder1, Feeder2, Feeder3, Feeder4 Actors:**

The CIMLAB is tested with four types of feeder parts. Feeder1 is a mill and then lathe assigning parts, Feeder2 is only mill assigning parts, Feeder3 is only lathe assigning parts, and Feeder4 is lathe and then mill assigning parts. While implementing the controller any feeder can be switched ON/OFF.

**Bin Actor:**

Bin actor takes care of the number of parts processed. The robot actor informs the bin actor when delivering a finished part to the bin.

**ProcessExecutor1, ProcessExecutor2 Actors:**

At any time in the CIMLAB execution, the system can handle two parts, so to execute two parts at any item requires two ProcessExecutors. The functions of the ProcessExecutor is to get a part assigned from the Scheduler and execute the process plan for that part. The ProcessExecutors have to reserve the machines and robot and load programs to the machines.

**Scheduler Actor:**

The function of the scheduler actor is to receive part information from the feeders and assign the part to the ProcessExecutors. The scheduler has to take care to avoiding deadlocks and assign parts, such that the machines are utilized to the maximum.
11.3 IMPLEMENTATION

The cell controller is successfully implemented in the CIMLAB with all four feeders turned ON. When only MillLathe and LatheMill resource feeders were turned ON, the cell controller executed first only MillLathe resource parts and then completed LatheMill resources part. This was due to considering the utilization of the machines as the highest priority. The scheduler assigned only one type of resource part to both the ProcessExecutors until all the parts were completed and then assigned the other resource parts to the ProcessExecutors; i.e., when a MillLathe resource part is executed in lathe by a ProcessExecutor1, the scheduler assigned another MillLathe resource part to the other ProcessExecutor2 to utilize the Mill machine.
CHAPTER XII

CONCLUSIONS

The cell controller was proposed to co-ordinate the manufacturing systems i.e., to co-ordinate Mill, Lathe, Robot. The proposed cell controller has to be independent of process plans, CNC programs. Also the cell controller has to integrate with the other levels of CIM and the reusability and maintainability of the software has to be maintained.

The proposed cell controller was built using Flexis, TCP/IP sockets, Objects and Negotiations. Objects are built using Flexis. TCP/IP sockets are used to open server ports and to implement the negotiations. Negotiations concept is defined using command and request and reply procedures.

The cell controller coordinates the Mill, Lathe CNC Machines and Robot using the negotiations. This gave realtime implementation of the manufacturing process. The Mill, Lathe, Robot actors negotiate with their respective real time machine controllers and successfully implemented the co-ordination between the CNC machines and Robot.

The ProcessExecutors and the Scheduler were implemented to have the cell controller independent of process plans and CNC programs. The user can define the process plan and select the CNC program names to implement the cell controller. This
was implemented by having a process plan format to define the process plan steps and the CNC programs. The ProcessExecutors and the Scheduler reads the process plan and implement the steps involved in it.

The integration of the cell controller with the other levels of CIM was also successfully implemented by having the server and reading the process plans at the instant of processing the part. Through the network the process plans can be changed at any instant of time, for which the cell controller reads the process plans and implements the process plan at realtime.

The concept of objects gave more flexibility for implementing the cell controller in a easy way. The objects can be fixed depending on the number of manufacturing systems. Using the grafcet implementation the state of the systems can be known at any instant of time.
REFERENCES


APPENDIX A

FLEXIS AND SFC

A.1 FLEXIS

FLEXIS is a control tool which helps in constructing control applications for cell control and executes programs to manage factory floor devices. It can be used to model organizations, parts flow and integrate factory processes. FLEXIS uses SFC+ (Sequential Function Chart), which is a language for expressing control flow between various types of steps connected to one another by transitions. SFC+ programs can be watched on the FLEXIS display while they are being executed. Actions within the sequential function chart are programmed in the C programming language. Once models are constructed, they can be executed using basic FLEXIS or FLEXIS RunTime package. RunTime executes the programs tenfold faster than the basic FLEXIS package.

A.1.1 Flexis Environment:

Mouse

FLEXIS features a multi-windowing environment that requires the use of the mouse. The mouse with left, middle and right button controls is used. The left button is used to select one or more items out of a group. The middle button is used to bring up the menu of command options that are relevant to the object pointed at by the mouse arrow. The right button shows the window menu options for modifying the display.
Menus

The FLEXIS menu system is dynamic. Only the available choices are present in FLEXIS menus, which are classified into three basic types: fixed, pop-up and slide-out menus. Fixed menus are top level FLEXIS menus, such as the process directory menu which is fixed on the display. Pop-up menus are available for a FLEXIS object when the middle mouse button is pressed while pointing at a FLEXIS object. Slide-out menus are represented by a triangle next to a menu choice indicating the sub-options with which it is associated. These sub-options are available via slide-out menus by using a left mouse button in fixed menus, a middle mouse button in FLEXIS object menus and a right mouse button for window or background menus. Figure A.1 shows the menu system in FLEXIS environment.

![Fixed Menu](Figure A.1 Fixed Menu)
Windows

The most flexible and powerful feature of the FLEXIS toolset is its ability to display many parts of a design on the screen at the same time using overlapping windows. FLEXIS allows an unlimited number of windows to be open and displayed at the same time. Every FLEXIS window has a title and a horizontal menu (with choices EXIT, HELP, EDIT, SCALE) displayed. The EDIT displays fixed menu for actions and objects that are specific for Sequential Function Chart building. Every window in FLEXIS is associated with a pop-up menu to move, shape, bury, display and print the window. Figure A.2 shows a basic window and its menu options.

![Figure A.2 Flexis Menu](image-url)
Operator Messages

The operator messages window is displayed during various FLEXIS operations, such as compiling sequential charts and during remote execution operations.

Non-Flexis Operations

Flexis operations, such as loading models, window manipulation and low level debugging, are provided. When the mouse cursor is positioned anywhere in the FLEXIS background and pressing the right mouse button displays the Background pop-up menu, which has functions like opening FileBrowser window, save virtual memory, TextEdit, etc. While a program is running, an error causes the Break window to be displayed and stops the execution.

A.1.2 SFC+ Language:

Sequential Function Chart (SFC+) is the language used by FLEXIS for stating high-level control. It is a simple language based on control flow between steps of various types. Steps are connected to one another by transitions; transitions have C expressions which determine when control is to pass from one step to another. SFC+ is a finite state language which has a considerable advantage; during execution the operator can see what step(or state) it is in. SFC+ expresses parallelism, branching, and synchronization.

Hierarchy

The hierarchical structure (Figure A.3) shows the nesting of the Grafcet macro steps. A pop-up menu displays the different operations on the objects.

Functions: The middle mouse button on the desired macro step displays a pop-up
menu for viewing objects, replacing text, compilation and printing.

![Hierarchial Structure Diagram](image)

**Figure A.3 Hierarchial Structure**

**SFC+ Display**

A Grafcet display can be produced from the hierarchy, the process directory, or the middle button menu of a macro step in the Grafcet window. The Grafcet can be changed to the required size and fonts.

**Creating SFC+**

In FLEXIS, Grafcet programs are called processes or actors. By selecting the **create** in the process directory and entering the new name, it will create a new actor to edit.

**Adding Elements: Regular step.** Using the left mouse button, the regular step icon displays a step in the window, which is moved and placed at the required position, and a name can be given to it. The name to a step can be done later using the label command. Slide out of the regular step icon displays a slide-out menu for a SEND, RECEIVE, INITIAL IN or OUT steps. **Macro step.** Using
the left mouse button, the macro step icon displays a macro step in the window, which is moved and placed at the required position, and a name can be given to it. The name to a step can be done later using the label command. Slide out of the macro step icon displays a slide-out menu for an ABORT macro. Abort step is used to stop the execution of a macro step, deactivate all of its active steps and have control pass immediately to the following step. **Transition**, Using the left mouse button, the transition icon displays a macro step in the window, which is moved and placed at the required position, and a boolean C expression is given to it. The boolean C expression can be done later using the label command. Slide out of the transition icon displays a slide-out menu for any previous, otherwise, or ground transition. Any previous transition become true if any of the previous transitions (to its left) in a regular asynchronous branch becomes true. The otherwise branch become true if none of the previous transitions (to its left) in a regular or first true asynchronous branch becomes true. The ground transition terminates the flow; i.e., when the transition is true the control flows to the ground and is terminated.

**Deleting Elements:** An object can be deleted using the delete command from the Actions menu and selecting the object. The mouse confirmation is also asked before deleting the object.

**Moving Elements:** An object or set of objects or a link within a Gratcet window can be moved selecting the move command from the Actions menu and the object or group of objects.
**Copying Elements:** An object or set of objects or a link within a Grafcet window can be copied selecting the copy command from the Actions menu and the object.

**Links and Linking:** Selecting link command in Actions menu and selecting the source and destination links both the objects. Linking one step to more than one transition creates an asynchronous branch, which gives two types of branches: **First true transition**, the token is passed to the single step for which the transition becomes true from left to right, and **All true transition** the token is passed to all steps for which the transitions are true. Linking one transition to many steps creates a synchronous branch. Linking many transitions to one step creates an asynchronous join. Linking many steps to one transition creates a synchronous join. Figure A.4 shows the Actions and Objects that can be selected.

![Actions Menu](image)
A.1.3 Flexis C Language:

C code and function calls in FLEXIS appear in step programs and transitions. All variables, user defined functions and include functions are defined in the C declarations associated with a Grafcet or a macro step. All step programs are written in activation, continuous and deactivation parts, and declarations are displayed in a C window. An activation program is executed when token enters into the step, a continuous program is executed for the time the token is present in the step, and a deactivation program is executed when the token is leaving the step. Three dots from top to bottom in the step is displayed to represent the programs. Figure A.5 shows the sample activation program.

![Sample Activation Program](image)

**Figure A.5** A Sample Activation Program

A.1.4 Cross Reference Analysis:

Using the mouse middle button, the Grafcet steps and transitions, templates and hierarchy elements are analyzed using a pop-up menu that includes analysis options. Only objects that have been last compiled successfully will be analyzed. Variables used, fetched and declared can be analyzed, and variables are also set for the analyzing object.
Functions defined and called are analyzed. Arrays, Pointers, and Structures are examined using the mouse middle button. Also, who declares, uses, sets and fetches is analyzed using the cross reference window. Figure A.6 shows a sample cross reference menu.

![Cross Reference Menu](image)

**Figure A.6 Cross Reference Menu**

A.1.5 Local Grafcet Execution:

By selecting one or more process windows and buttoning the execute command, the process directory displays the process execution window. Grafcet executes on a scan principle. A scan is a pass through all the active steps of the running Grafcet processes as follows:

1. Evaluate all the enabled transitions.
2. For each enabled transition that is true, run the deactivation part of the steps being deactivated. Then run the activation part of all the steps being active.
3. Run the continuous part of all active steps.

When an execution is running, when any Regular step becomes active, its internal region highlights in reverse video. If any Macro step is activated, the top half of the internal region is highlighted until the OUT step becomes active, and the lower half is highlighted when the OUT step is active.
Grafceť is executed using start command in the process execution window. The execution can be stopped in the middle, executed single scan or executed fast/slow. Grafceť windows and trace windows are displayed to view the execution. The variables associated with the processes can be set, displayed for selected or all of the variables. Figure A.7 shows the Local Execution menu.

![Figure A.7 Local Execution Menu](image)

A.1.6 Remote Grafceť Execution:

Remote execution is executed via RunTime. It is executed from the slide-out window from Execute, which opens a Remote Execution window which looks like the local Grafceť Execution window. The maximum rate scan is a function of the complexity of the process and the availability of the system resources. FLEXIS RunTime manages
remote resources accessed using the network interface, in order to prevent deadlocks and optimize system throughput.

Grafct in the Runtime environment is executed by first connecting to the processes. The processes must be downloaded and compiled in the target environment, and then loaded for remote execution. The execution can be stopped, unloaded and disconnected from the RunTime. Figure A.8 shows the Remote Execution menu.

Figure A.8 Remote Execution Menu

A.1.7 Additional Supported Options:

Animation

A FLEXIS animation is a schematic picture, whose parts can be moved by C function calls from step programs in Grafct. The user creates an initial state for an
animation through an editing process using basic objects, macro panels and regular panels. The executing processes can then manipulate the animation using a library of C routines called from the Grafcet step programs.

**Project Management**

A FLEXIS project is a collection of sysouts, its associated files and a set of users who can work on them. The project management interface will permit the start up of any of the sysouts and the tools for maintaining them.

**Ladder**

Ladder code in FLEXIS appears in step programs and transitions. FLEXIS ladder is based on the IEC draft specification for programming languages in programmable controllers, IEC SC65A/WG6/TF3.

**Library Manager**

The library manager is a tool for managing FLEXIS objects of different types. The Library Manager appears as an assortment of nodes that are linked together. Nodes can contain objects and can be linked with various types of relationships.
APPENDIX B

CIMLAB USERS

B.1 DIRECTORY STRUCTURE AND FILES

Appendix B describes how to create the users for the CIMLAB. It is assumed that the CIMLAB users interact only with the CIMLAB animation, rather than realtime machines. Under /usr/local of the phoenix -- a sun SPARC Station 10, a directory cim690 is created to have common files and CIMLAB users. The directory cim690 consists of the following:

**Dot Files**

All the dot files required for the CIMLAB users. Mainly the .cshrc file is modified to have a path for cim690.

**include**

An include directory is created to have the include files required in the development of Flexis programs for the CIMLAB users. The inet.h, inet.errors, and inet.lib are the files required for the Flexis Actors to connect to the server. The msg.h and msg.lib are the files required to interact between the Flexis Actors and Siman. The siman.dcls is a modified version of system.dcls, which has the identification numbers required to interact between Flexis and Siman. The pp_plan.lib has functions to read the process plan.

**NEWCLIENT**

NEWCLIENT is an Actor required for the CIMLAB users to start with. The NEWCLIENT Actor has ConnectToServer step to connect the Actor to the server, a ReadMessages macro to read the
server messages, an *Abort* macro to check the error codes, an empty *Process* macro in which the CIMLAB users create the grafcet and a *CloseServerConnection* step to close the connection to the Server.

**lisp.virtualmem**

It is the file required for the CIMLAB users to start with the flexis. All the optional software required for the Flexis is loaded in lisp.virtualmem. The NEWCLIENT Actor is also loaded in the lisp.virtualmem.

**cimlab, cimlab.lay, cimlab.p, cimlab.file**

All these files are CIMLAB animation developed using SIMAN and CINEMA.

**gosiman**

It is a shell program to start the CIMLAB animation, which call cimlab with cimlab.p and cimlab.lay as its parameters.

**server**

It is the server that is needed for the communication between Flexis actors, Siman, and Realtime Machines.

The CIMLAB users are created under */usr/local/cim690*. All the dot files, lisp.virtualmem and NEWCLIENT files are copied from the */usr/local/cim690* to the CIMLAB user directories. Since .cshrc has a path to the */usr/local/cim690*, all the other files required for the CIMLAB users are shared from that directory.

The following notes were prepared for CIMLAB users to start with and describe the Flexis grafcet and how to start the Flexis, Siman and Server. The functions required to interact between Flexis Actors and Siman are described in the cimlab user perspective.
B.2 ABOUT FLEXIS

B.2.1 Flexis Environment:

Mouse

Flexis features a multi environment that requires the use of the mouse. The mouse has left, middle and right buttons. The left button is used to select one or more items out of a group of items. The middle button is used to bring up the menu of command options that are relevant to the object pointed at by the mouse arrow. The right button shows the window menu and options for modifying the display.

Menus

The Flexis menu system is dynamic. Flexis menus always present only the available choices. Flexis has three basic types of menus: fixed, pop-up and slide-out. Fixed menus are top-level Flexis menus, such as the process directory menu which is fixed on the display. Pop-up menus are available for a Flexis object when the middle mouse button is pressed while pointing at a Flexis object. Slide-out menus are represented by a triangle next to a menu choice indicating the sub-options associated with it. These sub-options are available via slide-out menus by using a left mouse button in fixed menus, a middle mouse button in Flexis object menus and a right mouse button for window or background menus.

Windows

The most flexible and powerful feature of the Flexis ToolSet is its ability to display many parts of a design on the screen at the same time using overlapping windows. Flexis allows an unlimited number of windows to be opened and displayed at the same
time. Every Flexis window has a title and a horizontal window (with choices EXIT, HELP, EDIT, SCALE) displayed. The EDIT displays a fixed menu for actions and objects that are specific to Sequential Function Chart building. Every window in Flexis is associated with a pop-up menu that allows the user to move, shape, bury, display and print the window.

**Operator Messages**

The operator messages window is displayed during various Flexis operations, such as compiling sequential charts and during remote execution operations.

**Non-Flexis Operations**

Non-Flexis operations, such as loading models, window manipulation and low level debugging, are provided. When the mouse cursor is positioned anywhere in the Flexis background and pressing the right mouse button displays the Background pop-up menu, it has functions like opening the FileBrowser window, to save virtual memory, TextEdit, etc. While a program is running, an error causes the Break window to be displayed and stops the execution.

**B.2.2 SFC+ Language:**

Sequential Function Chart (SFC+) is the language used by Flexis for stating high-level control. It is a simple language based on control flow between steps of various types. Steps are connected to one another by transitions; transitions have C expressions on them that determine when control is to pass from one step to another. SFC+ is a finite state language which gives considerable advantage that during execution we can see what
step (or state) it is in. SFC+ expresses parallelism, branching, and synchronization.

**Hierarchy**

The hierarchical structure shows the nesting of the Grafcet macro steps. A pop-up menu displays different operations on objects.

**Functions:** The middle mouse button on the desired macro step displays a pop-up menu for viewing objects, replace text, compilation, printing.

**SFC+ Display**

A Grafcet display can be produced from the hierarchy, the process directory, or the middle button menu of a macro step in the Grafcet window. The Grafcet can be changed to the required size and fonts.

**Creating SFC+**

In Flexis, Grafcet programs are called processes or actors. By selecting the create in the process directory and entering the new name will create a new actor to edit.

**Adding Elements: Regular step**, left mouse button the regular step icon displays a step in the window, which is moved and placed at the required position and a name can be given to it. The name to a step is done later using label command. Slide out of the regular step icon displays a slide-out menu for a SEND, RECEIVE, INITIAL IN or OUT steps.

**Macro step**, left mouse button the macro step icon displays a macro step in the window, which is moved and placed at the required position and a name can be given to it. The name to a step is done later using label command. Slide out of the macro step icon displays a slide-out menu for a ABORT macro. Abort step
is used to stop execution of a macro step, deactivate all of its active steps and have control pass immediately to the following step.

**Transition**, left mouse button the transition icon displays a macro step in the window, which is moved and placed at the required position and a boolean C expression is given to it. The boolean C expression can be done later using label command. Slide out of the transition icon display a slide-out menu for anyprevious, otherwise, or ground transition. The anyprevious transition becomes true if any of the previous transitions (to its left) in a regular asynchronous branch becomes true. The otherwise branch becomes true if none of the previous transitions (to its left) in a regular or first true asynchronous branch becomes true. The ground transition terminates the flow i.e., when the transition is true the control flows to the ground and terminated.

**Deleting Elements**: An object can be deleted using delete command from Actions menu and selecting the object. The mouse confirmation is also asked before deleting the object.

**Moving Elements**: An object or set of objects or a link within a Grafcet window can be moved selecting the move command from Actions menu and object or group of objects.

**Copying Elements**: An object or set of objects or a link within a Grafcet window can be copied selecting copy command from Actions menu and object.

**Links and Linking**: Selecting link command in Actions menu and selecting the source and destination links both the objects. Linking one step to more than one
transition creates an asynchronous branch, which gives two types of branches, **First true transition** the token is passed to the single step for which the transition becomes true from left to right, **All true transition** the token is passed to all steps for which the transitions are true. Linking one transition to many steps creates a synchronous branch. Linking many transitions to one step creates an asynchronous join. Linking many steps to one transition creates a synchronous join.

**B.2.3 Flexis C Language:**

C code and function calls in Flexis appear in step programs, transitions. All variables, user defined functions and include functions are defined in the C declarations associated with a Grafcet or a macro step. All step programs are written in activation, continuous, and deactivation parts and declarations are declared in a C window. An **activation program** is executed when token enters into the step, a **continuous program** is executed for the time the token is present in the step, and **deactivation program** is executed when the token is leaving the step. Three dots from top to bottom in the step are displayed to represent the programs.

**B.2.4 Cross Reference Analysis:**

Using mouse middle button the Grafcet steps and transitions, and hierarchy elements are analyzed using a pop-up menu that includes analysis options. Only objects that have been last compiled successfully will be analyzed. Variables used, fetched,
declared can be analyzed and variables can be set for the analyzing object. Functions
defined and called can be analyzed. Arrays, Pointers, and Structures are examined using
mouse middle button.

**B.2.5 Local Grafcet Execution:**

By selecting the one or more process windows and buttoning the execute
command in process directory displays the process execution window. Grafcet executes
on a scan principle. A scan is a pass through all the active steps of the running Grafcet
processes as follows:

1. Evaluate all the enabled transitions.

2. For each enabled transition that is true, run the deactivation part of the steps
   being deactivated. Then run the activation part of all the steps being active.

3. Run the continuous part of all active steps.

When an execution is running, any Regular step becomes active, its internal region
highlights in reverse video, if any Macro step is activate the top half of the internal region
is highlighted, and the lower half is highlighted when the OUT step is active.

Grafcet is executed using Start command in Process Execution window. The
execution can be stopped in the middle, executed single scan or executed fast/slow.
Grafcet windows and trace windows are displayed to view the execution. The variables
associated with the processes can be set, selected variables or all the variables can be
displayed.
B.2.6 Text Editor in Flexis:

Text editor is used to create and modify files. The left mouse button in the flexis background displays a pop-up window, select *TEdit*. Flexis displays a resizable window, move to a location and drag with left mouse button to size it conveniently. The middle mouse button on the top black bar displays a pop up window, which has commands like *Put*, *Get*, *Quit* etc. *Put* writes to a file, *Get* gets already created file and *Quit* quits the text editor window.

B.2.7 Documentation and Printing Grafcet:

Documentation gives a printout of the grafcet and the C programs. Hardcopy gives a printout of that particular grafcet. Documenting an actor to a file or a printer is done using the Document command in the ProcessDirectory menu. Dragging the middle mouse button on the *Document* command, flexis displays a pop-up window and selects *Document All* to print all of the Actor. To Document a particular hierarchial level: (1) Get the Hierarchical Structure by selecting *Hierarchy* in ProcessDirectory menu; (2) The middle mouse button on any level displays a pop-up window in it to drag the middle mouse button on *Document* and select *to printer* to print the document. To print a grafcet, the left mouse button is pressed in any part of the grafcet. Flexis displays a pop-up window, select *Hardcopy* to print the grafcet.
B.3 HOW TO START CIMLAB PROGRAMS

B.3.1 How to Start Openwindows:

Step 1) Login under group number.
Step 2) Run openwindows.

B.3.2 How to Start Flexis:

Step 1) Run goflexis in flexis cmdtool.

Note: The window is not available for other applications until the user quits Flexis.

When the user starts Flexis, the Flexis ToolSet window is displayed. In Flexis ToolSet window, the user builds applications for cell control programs to manage factory floor devices or to model an organization's information or parts flow.

B.3.3 How to Create Grafcet Programs:

The Actor for a project is nested with four macros. The first macro is ConnectToServer, which registers the Actor with the Server. The Server is a program that allows multiple actors to talk with one another and to the simulation of the cimlab. From ConnectToServer through a synchronous branching, it is branched to three parallel macro's: Abort, ReadMessages and Process. The Abort macro aborts the Actor when there is an error. The ReadMessages macro reads the received messages and processes it. The Process macro defines the steps involved in accomplishing the specified tasks.

To accomplish a task to create programs only in the Process macro.
(i) Create NEW Actor:

Step 1) From the Process Directory copy the NEWCLIENT actor to create a new actor.

Step a) Select the NEWCLIENT Actor from the Process Directory.

Step b) Click the left mouse button on Copy command in the Process Directory.

Step c) A prompt is given for the name to the new Actor. Give a name to the Actor.

Step d) Flexis display a window to change the patterns. There is no need to change any patterns, so click the left mouse button on EXIT.


Example: A a new actor with the name PROJECT1

(ii) View the Actor:

Step 2) Unselect NEWCLIENT Actor and select an Actor. To view the actor,
click left mouse button on View command.

**Note:** To view the C declarations, slide-out the middle mouse button on the View command. A pop-up menu displays the options, and from the options select the view C declarations.

**Example:** Figure B.1 shows the actor PROJECT1.

(iii) **Give Identification Number to the Actor:**

Step 3) Since the Actor has to identify itself with the Server, an identification number is given to it.

```c
#define MYID 10
#include <stdlib.h>
#include <stdio.h>
#include <flexis.h>
#include <flxtcp.h>
#include <ctype.h>
#include <string.h>
#include <timer.h>
#include "./include/siman.dclls"
#include "./include/inet.h"
#include "./include/inet.errors"
#include "./include/inet.lib"
#include "./include/msg.h"
#include "./include/msg.lib"
```

**Figure B.2 C Declarations**

**Example:** Figure B.2 shows the C declarations for the Actor PROJECT1. MYID is #define with 10.

Note: MYID can be #defined with a pre #defined actorName. If using multiple actors, it is convenient to #define all actorid's andmsgid's in a
common file and include it before #defining MYID.

(iv) **Create a Process:**

Step 4) Click the middle mouse button on the Process macro; a pop-up menu is displayed. Select *Create Macro Step* to create a new macro or *View Macro* if it is already created. Note: For a macro there will be an *IN* and *OUT* step.

Step a) Using left mouse button displays the required object i.e., step, macro, or transition, move the object to the required position and click the left mouse button to fix the object in that position; then flexis prompts for the name to a step and a boolean condition to transition. Type the required name or boolean condition.

Note: Once the object is positioned or if a name has not been given the object can be moved or labelled by using the commands in actions menu.

Step b) Link the objects according to the requirements; i.e a synchronous join, asynchronous branching, etc..

Note: Two of same objects cannot be linked.

Step c) The middle mouse button on a *Step* displays a pop-up menu to create an activation, continuous or deactivation programs. The middle mouse button on a *Transition* display a pop-up menu to EDIT the transition.
Example: A Process macro is created to send a request to the siman lathe machine and waits for the reply from the siman lathe.

Figure B.3 shows the Process macro.

C declarations for Process macro:

```c
int LatheExec;
```

**Description:** To get the transaction number returned by `req` function.

**Activation program in ExecLathe step:**

```c
LatheExec = req(SLATHE, EXEC_PROG, "10");
```

**Description:** A request is sent to the SLATHE to execute the program for 10 seconds. EXEC_PROG is the message identification number, SLATHE is the identification number for siman Lathe, #defined in system.dcls.

**Transition label for WaitReply:**

```c
wait_reply(LatheExec);
```
Description: wait_reply waits for the reply that is sent. LatheExec is the transaction number returned by the req function.

Activation program in ClearReply step:

clear_reply(LatheExec);

Description: clear_reply clears the reply that is received.

(v) Compile the Actor:

Step 5) Click the left mouse button on Compile in the Process Directory. Flexis displays an operators message and shows the status of the compilation.

(vi) Process Execution:

Step 6) Click the left mouse button on Execute in the Process Directory. A fixed menu for Process Execution is displayed; move it to the required position and place the Process Execution window.

B.3.4 How to Start Siman:

Step 1) Run gosiman in siman cmdtool. Siman displays a CINEMA window.

B.3.5 How to Start Server:

Step 1) Run server with parameters in server cmdtool

Usage: server [-m#] [-M#] [-t#] [-g#] [-i#] [-b#] [-f#] [-p#] [-a]

Where

- \( m \) denotes minimum number of clients specified by #
- \( M \) denotes maximum number of clients specified by #
- \( i \) denotes important id specified by #
$t$ denotes timeout (in sec’s)

$g$ denotes number of GENERIC_PORTS specified by #

$f$ denotes the number of flexis clients specified by #

$p/b$ number of ports to be opened

$a$ don’t kill all in case of error

**Example:** Since a flexis actor and siman are connected, the command to run the server is

server -fl -i201 -i202 -i203

where 201, 202, 203 are identification numbers for the Siman Mill, Siman Lathe, Siman Robot.

**B.3.6 How to Run Simulation:**

Step 1) Click the left mouse button on **Start** in the Process Execution window.

Note 1: If you are restart make sure you initialize the Actor.

Note 2: If Flexis displays a pop-up window saying unix error, press the middle button on the window to display a pop-up window, select up-arrow to Abort the Actor. Initialize the Actor by selecting Initialize in the Process Execution. Terminate the server by **ctrl c** in server window, restart the server and Start the Execution.

Step 2) Point mouse cursor in CINEMA window and press **esc**. The CINEMA window displays the menu options. Click **RUN** from the menu options.

Step 3) Observe CINEMA to see your program executing.
**B.3.7 How to Exit Simulation:**

**Step 1)** If CINEMA window is running, click *QUIT* from menu options.

**Step 2)** In Flexis click left mouse button on *Stop* in the Process Execution window to stop the process. Click left mouse button on *Logout* in Flexis ToolSet Window to save current status. Flexis prompts for confirmation *Yes/No*; Confirm Yes to save the current status. To logout without the current status slide-out of the middle mouse button on logout which gives the options to logout without saving the current status.

**Step 3)** Exit Openwindows.

**Step 4)** Logout from the machine.
B.4 FUNCTIONS TO INTERACT

Library routines are used by the actor to send a command, request or reply and to process the received command, request or reply.

To use these library files, the following files has been included in Actor C declarations.

```
#define MYID ActorName
#include <stdlib.h>
#include <stdio.h>
#include <flexis.h>
#include <flxtcp.h>
#include <timer.h>
#include "siman.dcls"
#include "inet.h"
#include "inet.errors"
#include "inet.lib"
#include "msg.h"
#include "msg.lib"
```

Note: ActorName is the identification number for the actor. The identification numbers and message identification numbers can be #defined for all actors in a common file and included before MYID is defined or an integer is defined for MYID.

---

**req**

Sends a request to the specified destination

**Synopsis**

```
int req(int dest, intmsgid, char* param)
```

**Arguments**

- **int dest**
  
  The destination Identification number for the message.

- **int msgid**

  The message identification number.

- **char* param**

  Pointer to the parameters that are being sent with the request.

**Description**

The req function sends a request to the specified destination. The msgid is the message identification number on which the destination is waiting for the request using the wait_req function. When the specified destination
receives the request, the wait_req function returns true. The param is a pointer to the message information that is being sent.

**Returns**
The transaction number to be used in the wait_reply function.

**Example**
int GetPart;
sprintf(buf,"%4d %15s %4d %15s",source,pickLocation,dest,placeLocation);
GetPart = req(ROBOT, PICK, buf);

Note: In this example the robot has to pick a part and place it. To pick a part the robot has to know the source and exact location of the part and the destination and location to place the part. All this information is sent through buf, a pointer to information.

Note: GetPart is declared globally since it must be used in wait_reply, read_reply, clear_reply.

---

**wait_reply**
Waits for the reply message.

**Synopsis**
bool wait_reply(int tn)

**Arguments**
int tn
Transaction number of the request that is being waited.

**Description**
The wait_reply function waits for the reply that it has sent. The function waits on the transaction number that is returned by the request function. When it receives the reply for the request, the function returns true.

**Returns**
Returns true if the reply has been received.

**Example**
wait_reply(GetPart);

Note: GetPart is an integer variable globally declared, and it is the transaction number returned by the req function.

---

**read_reply**
Reads the contents of the received reply message.

**Synopsis**
char* read_reply(int tn)
Arguments int tn
Transaction number of the request that is being waited.

Description The read_reply function reads the contents of the received reply message. The function returns a pointer to the contents buffer which can store in a specified pointer.

Returns Returns a pointer to the string buffer of the contents.

Example char* replyMsg;
replyMsg = read_reply(GetPart);

Note: GetPart is an integer variable globally declared, and it is the transaction number returned by the req function.

---

clear_reply
Clears the received reply.

Synopsis void clear_reply(int tn)

Arguments int tn
Transaction number of the request that received the reply.

Description The clear_reply function clears the Req_msg entry pointed to by transaction number. A second transaction cannot be sent until the previous one was cleared.

Returns None

Example clear_reply(GetPart);

Note: GetPart is an integer variable globally declared, and it is the transaction number returned by the req function.

---

cmd
Sends a command to the specified destination.

Synopsis void cmd(int dest, int msgid, char* param)

Arguments int dest
The destination Identification number for the message.

int msgid
    The message identification number.
char* param
    Pointer to the parameters that are being sent with the command.

**Description** The cmd function sends a command to the specified destination. The msgid is the message identification number on which the destination is waiting for the command using wait_cmd function. When the specified destination receives the command, the wait_cmd function returns true. The param is a pointer to any message information that is being sent.

**Returns** None

**Example**
cmd(SCHEDULER, AVAILABLE, "I am free to process a new part");

---

**wait_cmd**
Waits for the command message.

**Synopsis**
bool wait_cmd(int msgid)

**Arguments**
int msgid
    The message identification number.

**Description** The wait_cmd function waits for the command message. The function waits on msgid, a message identification number that is being sent by the cmd function.

**Returns** Returns true if the command has been received.

**Example**
wait_cmd(EXEC);

---

**read_cmd**
Reads a received command message.

**Synopsis**
char* read_cmd(int msgid)

**Arguments**
int msgid
    Message identification number for the command that is read.
**Description**
The read_cmd function reads the contents of the received reply message. The function returns a pointer to the contents buffer which can be stored in a specified pointer.

**Returns**
Returns a pointer to the string buffer of the contents.

**Example**
```c
char* cmdMsg;
cmdMsg = read_cmd(EXEC);
```

---

**clear_cmd**
Clears the received command.

**Synopsis**
```c
void clear_cmd(int tn)
```

**Arguments**
```c
int tn
```
Transaction number of the request that received the reply.

**Description**
The clear_cmd function clears the command entry. After a command is read it must be cleared before a wait_cmd can be executed.

**Returns**
None

**Example**
```c
clear_cmd(EXEC);
```

---

**source_cmd**
Gets the source for the received command.

**Synopsis**
```c
int source_cmd(int msgid)
```

**Arguments**
```c
int msgid
```
Message identification number for the command.

**Description**
The source_cmd function reads the source of the received command message. This procedure must be executed before a command is cleared.

**Returns**
Source of the command.

**Example**
```c
int source;
source = source_cmd(EXEC);
```
**wait_req**
Waits for the request message.

**Synopsis**
```c
bool wait_req(int msgid)
```

**Arguments**
- `int msgid`
  Message identification number for the request that is being waited.

**Description**
The `wait_req` function waits for the req that it has sent. The function waits on `msgid`, a message identification number that is being sent by the req function.

**Returns**
Returns true if the request has been received.

**Example**
```c
wait_req(EXEC);
```

---

**read_req**
Reads the contents of the received req message.

**Synopsis**
```c
char* read_req(int msgid)
```

**Arguments**
- `int msgid`
  Message identification number for the request that is being waited.

**Description**
The `read_req` function reads the contents of the received request message. The function returns a pointer to the contents buffer, which can be stored in a specified pointer.

**Returns**
Returns a pointer to the string buffer of the contents.

**Example**
```c
char* reqMsg;
reqMsg = read_req(EXEC);
```

---

**reply**
Sends a reply to the received request.

**Synopsis**
```c
void reply(int msgid, char* param)
```
Arguments

int msgid
   The message identification number.
char* param
   Pointer to the parameters that are being sent with the reply.

Description
The req function sends a request to the specified destination. The msgid
is the message identification number on which the destination is waiting
for the reply using wait_reply function. When the specified destination
receives the reply, the wait_reply function returns true. The param is a
pointer to any message information that is being sent.

Returns
None

Example
reply(EXEC, "Hi I got your request");

clear_req
Clears the received request.

Synopsis
void clear_req(int msgid)

Arguments
int msgid
   Message identification number for the request that is being waited for.

Description
The clear_req function clears the request pointed by message identification
number. After a request has been read, it must be cleared before a
wait_req can be executed.

Returns
None

Example
clear_req(EXEC);

B.5 SIMAN INTERFACE

The following functions form the interface with the Siman simulation of the
CIMLAB. Siman destinations and message identification numbers are declared in the
Siman has three destinations SMILL, SLATHE, SROBOT. SMILL is mill representation, SLATHE is lathe representation and SROBOT is robot representation.

Siman message identification numbers are LOAD_PROG, EXEC_PROG, GOTO, PICK, PLACE.

1) The LOAD_PROG request loads the CNC machines with a program.

2) The EXEC_PROG request executes the program for a specified number of seconds. The number of seconds the CNC machines takes to execute the program is given in the param.

3) The GOTO request instructs the robot to move to the destination specified by param.

4) The PICK request instructs the robot to pick up a part.

5) The PLACE request instructs the robot to release the part in its gripper.

The following are examples of functions that are used to interface with Siman simulation of the CIMLAB.

LoadProg = req(SMILL,LOAD_PROG,"");

ExecProg = req(SLATHE,EXEC_PROG,"10");

wait_reply(LoadProg);

clear_reply(LoadProg);

RobotGoto = req(SROBOT,GOTO,"1");

Note: (1) In req with EXEC_PROG msgid, the number can be any integer value
which represents the number of seconds the program takes to execute.

(2) In req with GOTO msgid, the number has to be 1, 3, 5 only. Number 1 represents Lathe, 3 represents Feeder and 5 represents Mill.

**SFC Example:**

The following SFC example starts the Mill and wait for it to complete.
C.1 INET.LIB

Library routines that are used by the actor to connect to the server, to register with the server and to get a go message are defined in inet.lib library.

To use these library files, the following files has to be included in Actor C declarations:

```c
#include "system.dcls"
#define MYID actorName
#include <stdlib.h>
#include <stdio.h>
#include <flexis.h>
#include <flxtcp.h>
#include "inet.h"
#include "inet.errors"
#include "inet.lib"
```

**init_actor**
Initialize the global variables for the actor.

**Synopsis**
void init_actor()

**Arguments**
None

**Description**
The init_actor function initializes the global variables: Error_code, KillActor, and RecKillMsg. Error_code is an integer variable set to zero. KillActor and RecKillMsg are boolean variables set to False.

**Returns**
None

**Example**
init_actor();

**server_connect**
To connect the actor to the server.
## server_connect

**Synopsis**

```c
int server_connect()
```

**Arguments**

None

**Description**

The `server_connect` function connects the actor to the unix tcp server. It first connects to the known generic port on the server. The server opens a unique server socket for the actor and sends the identification number of this socket across the generic port. The actor connects to the unique server socket and closes the generic port.

**Returns**

If success returns 0, else returns error `COULD_NOT_OPEN GENERIC_PORT`, or `COULD_NOT_OPEN_SERVER_PORT`.

**Example**

```c
Error_code = server_connect();
```

## server_register

**Synopsis**

```c
void server_register(int id)
```

**Arguments**

```c
int id
```

Actor unique identification number.

**Description**

The `server_connect` function registers the identification number with the server by sending it out through the server tcp port. The `server_connect` function calls `tcp_write` routine to write the message on the server socket.

**Returns**

None

**Example**

```c
server_register(MYID);
```

## get_go

**Synopsis**

```c
int get_go()
```

**Arguments**

None

**Description**

The server first sends a `GO` message when all the specified ports are
connected. The get_go function checks that the first received message from
the server is a GO message or not. If first message is not a GO message
it returns an error.

Returns If it is success, it returns 0, otherwise, it returns error
GO_NOT_FIRST_MESSAGE.

Example Error_code = get_go();

send_TYPED_MSG
Sends message out of the actor's socket.

Synopsis void send_TYPED_MSG(int type, int dest, char* contents)

Arguments int type
The type of message
int dest
The identification number of the destination for the message
char* contents
Pointer to the message string.

Description The send_TYPED_MSG function sends a message out of the actor's socket.
The message format is "<len> <dest> <source> <type> <contents>". Len
is the total number of bytes in the message and source is the sending
actor's identification number. Len, dest, source and type are all 4 digit
integers separated by a single space. Contents is a pointer to the
parameters string. The send_TYPED_MSG function calls the tcp_write routine
to write the message on the server socket.

Returns None, but the function assigns error COULD_NOT_WRITE_PORT to the
global Error_code variable.

Example send_TYPED_MSG(KILL, SERVER, " ");

send_MSG
Sends a normal message out of actor's socket.

Synopsis void send_MSG(int dest, char* contents)

Arguments int dest
Identification number of the destination for the message
char* contents
   Pointer to the message string.

Description The send_msg function sends a message out of the actor's socket. This is implemented as a macro defining send_TYPED_msg routine as a normal message. The message format is "<len> <dest> <source> <type> <contents>". Len is the total number of bytes in the message, source is the sending actor's identification number and type is the NORMAL message. Len, dest, source and type are all 4 digit integers separated by a single space. Contents is a pointer to the parameters string. The send_TYPED_msg function calls tcp_write routine to write the message on the server socket.

Returns None, but the function assigns error COULD_NOT_WRITE_PORT to the global Error_code variable.

Note This function is not available for the user. It is used in command, request and reply functions to send NORMAL messages.

rec_msg
Reads the message out of the actor's socket.

Synopsis int rec_msg(int source, char* contents)

Arguments int source
   The identification number of the source of the message.
char* contents
   Pointer to the string that the message to be placed.

Description The rec_msg function receives a message out the of the actor's socket in the format "<len> <dest> <source> <type> <contents>". Len is the total number of bytes in the message, dest is the destination of the message, source is this actor's identification number, type is the type of message and contents is a pointer to the parameters string. Len, dest, source and type are all 4 digit integers separated by a space. Contents is a pointer to the parameter string. The rec_msg function calls tcp_read routine to read the message on the server socket.

Returns If successful, it returns the type of message; otherwise, it returns error COULD_NOT_READ_PORT.

Note This function is not available for the user. It is used in get_message
get_message
Processes a message on the tcp port.

Synopsis  int get_message()

Arguments  None

Description The get_message function gets the type of message from rec_msg function
and processes the message. If the type is NORMAL, it processes the
message by calling proc_msg function. If the type is KILL, it sets the
KillActor and Rec_kill_msg global variables to true. If the type is PING,
it does nothing. If the type is a GO, HALT, RETURNED message,
CLOSE or default it assigns the respective error to the error variable.

Returns  If the type of message is NORMAL, KILL or PING it returns none;
otherwise, it returns error equal to 0 if the type of message is NORMAL
or KILL, or it returns error GO_SENT_AFTER_PORT_OPENED,
HALT_NOT_SUPPORTED,MESSAGE_RETURNED_FROM_SERVER,
RECEIVED_CLOSE, or UNDEFINED_MESSAGE_TYPE.

Example  Error_code = get_message();

C.2 MSG.LIB
Library routines that are used by the actor to send a command, request or
reply and to process the received command, request or reply as defined in
msg.lib library.

To use these library files, the following files have to be included in Actor
C declarations:

#include "system.dcls"
#define MYID actorName
#include <stdlib.h>
#include <stdio.h>
#include <flexis.h>
#include <flxtcp.h>
#include "inet.h"
#include "inet.errors"
#include "inet.lib"
#include "msg.h"
#include "msg.lib"

proc_msg
Process the arrived message.

Synopsis void proc_msg(int source, char* msg)

Arguments int source
    Source of the arrived message.
char* msg
    Pointer to the message string.

Description The proc_msg function processes the message depending on the type of message. If the message is a command, it updates the message identification number in Input_msg array. If the message is a request, it updates the message transaction number and message identification number in Input_msg array. If the message is a reply, it updates the message transaction number in Input_msg array. For all three types, it updates the Input_msg array's state field to MSG_ARRIVED and contents field with the message contents. If any message attempts to overwrite a message, it sets Error_code to MESSAGE_OVERWRITE.

Returns None.

Note: This function is not available for the user. It is used in get_message function to process the received message.

req
Sends a request to the specified destination.

Synopsis int req(int dest, intmsgid, char* param)

Arguments int dest
    The destination identification number for the message.
int msgid
    The message identification number.
char* param
    Pointer to the parameters that are being sent with the request.

Description
    The req function sends a request to the specified destination. The routine
    first finds an empty location in the Req_msg array, updates the fields in
    this structure and sends the message along with the message string pointed
    to param by calling send_msg routine of inet.lib. The message format is
    <REQ> <tn> <msgid> <param> where REQ is the type of message, tn is
    the transaction number of the index into the Req_msg array, msgid is the
    message identification number and param is a pointer to the contents
    buffer. REQ is a one digit, tn is a four digit, msgid is a four digit number,
    each separated by a space. If Req_msg array is full, it sets the Error_code
    to the error REQ_ARRAY_SIZE_EXCEEDED.

Returns
    Returns the message transaction number that will be used in the routines
    wait_reply, read_reply, clear_reply.

Example
    int GetPart;
    sprintf(buf, "%4d %15s %4d %15s", source, PP[step].pick, dest,
            PP[step].place);
    GetPart = req(ROBOT, PICK, buf);

    Note: GetPart is declared globally in the macro, because it is used in
    wait_reply, read_reply, clear_reply.

wait_reply
    Waits for the reply message.

Synopsis
    bool wait_reply(int tn)

Arguments
    int tn
        Transaction number of the request that is being waited.

Description
    The wait_reply function waits for the reply that it has sent. This is
    implemented as a macro by verifying the state field in Req_msg array to
    be REPLY_ARRIVED.

Returns
    Returns true if the reply has been received.

Example
    wait_reply(GetPart);

    Note: GetPart is an integer variable globally declared in the macro, and
it is the transaction number returned by the req function.

---

**read_reply**
Reads the contents of the received reply message.

**Synopsis**
```
char* read_reply(int tn)
```

**Arguments**
```
int tn
```
Transaction number of the request that is being waited.

**Description**
The read_reply function reads the contents of the received reply message. This is implemented as a macro by pointing to the content field in Req_msg array.

**Returns**
Returns a pointer to the string buffer of the contents.

**Example**
```
char* replyMsg;
replyMsg = read_reply(GetPart);
```

**Note:** GetPart is an integer variable globally declared in the macro, and it is the transaction number returned by the req function.

---

**clear_reply**
Clears the received reply.

**Synopsis**
```
void clear_reply(int tn)
```

**Arguments**
```
int tn
```
Transaction number of the request that received the reply.

**Description**
The clear_reply function clears the Req_msg entry pointed to by the transaction number. It is implemented as a macro by updating the state field in Req_msg array to MSG_CLEAR.

**Returns**
None

**Example**
```
clear_reply(GetPart);
```

**Note:** GetPart is an integer variable globally declared, and it is the transaction number returned by the req function.
**cmd**
Sends a command to the specified destination.

**Synopsis**
void cmd(int dest, int msgid, char* param)

**Arguments**
- int dest
  - The destination identification number for the message.
- int msgid
  - The message identification number.
- char* param
  - Pointer to the parameters that are being sent with the command.

**Description**
The cmd function sends a command to the specified destination by calling the send_msg routine of inet.lib. The message format is `<CMD> <msgid> <param>` where CMD is the type of the message, msgid is the message identification number and param is a pointer to the contents string. The CMD is 1 digit, msgid is 4 digit numbers, each separated by a space.

**Returns**
None

**Example**
cmd(SCHEDULER, AVAILABLE, "I am free to process a new part");

---

**wait_cmd**
Waits for the command message.

**Synopsis**
bool wait_cmd(int msgid)

**Arguments**
- int msgid
  - The message identification number.

**Description**
The wait_cmd function waits for the command message. This is implemented as a macro by verifying the state field in Input_msg array to be MSG_ARRIVED.

**Returns**
Returns true if the command has been received.

**Example**
wait_cmd(EXEC);
**read_cmd**
Reads a received command message.

**Synopsis**
```
char* read_cmd(int msgid)
```

**Arguments**
int msgid
Message identification number for the command that is read.

**Description**
The `read_cmd` function reads the contents of the received reply message. This is implemented as a macro by pointing to the content field in `Input_msg` array.

**Returns**
Returns a pointer to the string buffer of the contents.

**Example**
```
char* cmdMsg;
cmdMsg = read_cmd(EXEC);
```

---

**clear_cmd**
Clears the received command.

**Synopsis**
```
void clear_cmd(int tn)
```

**Arguments**
int tn
Transaction number of the request that received the reply.

**Description**
The `clear_cmd` function clears the `Input_msg` entry pointed to by transaction number. It is implemented as a macro by updating the state field in `Input_msg` array to MSG_CLEAR.

**Returns**
None

**Example**
```
clear_cmd(EXEC);
```

---

**source_cmd**
Gets the source for the received command.

**Synopsis**
```
int source_cmd(intmsgid)
```

**Arguments**
int msgid
Message identification number for the command.
**Description**  The source_cmd function reads the source of the received command message. This is implemented as a macro by defining the source field in Input_msg array as the source_cmd function.

**Returns**  Source of the command

**Example**  
```c
int source;
source = source_cmd(EXEC);
```

**wait_req**  
Waits for the request message.

**Synopsis**  
```c
bool wait_req(int msgid)
```

**Arguments**  
```c
int msgid
```
Message identification number for the request that is being waited.

**Description**  
The wait_req function waits for the req that it has sent. This is implemented as a macro by verifying the state field in Input_msg array to be MSG_ARRIVED.

**Returns**  Returns true if the request has been received.

**Example**  
```c
wait_req(EXEC);
```

**read_req**  
Reads the contents of the received req message.

**Synopsis**  
```c
char* read_req(int msgid)
```

**Arguments**  
```c
int msgid
```
Message identification number for the request that is being waited.

**Description**  
The read_req function reads the contents of the received request message. This is implemented as a macro by pointing to the content field in Input_msg array.

**Returns**  Returns a pointer to the string buffer of the contents.

**Example**  
```c
char* reqMsg;
```
reqMsg = read_req(EXEC);

reply
Sends a reply to the received request.

Synopsis       void reply(int msgid, char* param)

Arguments
int msgid
    The message identification number.
char* param
    Pointer to the parameters that are being sent with the reply.

Description The req function sends a request to the specified destination by calling send_msg routine of inet.lib. The message format is <REPLY> <tn> <msgid> <param> where REPLd is the type of the message, tn is the transaction number of the received request, msgid is the message identification number and param is a pointer to the contents string. The REPLd is a one digit, tn is a four digit, msgid is a four digit number, each separated by a space.

Returns      None

Example      reply(EXEC, "Hi I got your request");

clear_req
Clears the received request.

Synopsis       void clear_req(int msgid)

Arguments
int msgid
    Message identification number for the request that is being waited

Description The clear_req function clears the Input_msg entry pointed by message identification number. It is implemented as a macro by updating the state field in Input_msg array to MSG_CLEAR.

Returns      None

Example      clear_req(EXEC);
C.3 PP_PLAN.LIB

Library routines that are used to read a process plan are defined in pp_plan.lib library.

To use these library files, the following files have to be included in Actor C declarations:

```c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <ctype.h>
#include "pp_plan.lib"
```

**get_p**

Gets the process plan for the part.

**Synopsis**

```c
void get_p(FILE* stream)
```

**Arguments**

- **FILE* stream**
  - Process plan file that is to be read.

**Description**

The get_p function gets the name and process plan steps for the part in PP array. This is implemented, first by reading each uncommented line and getting the name and process plan steps. The format for the process plan looks like:

```plaintext
#This is for piston
Name:piston

#Each line should have the following format
#<Machine>:<place location>,<program>,<program time>,<pick location>
Lathe: lathe_insert ,e:\piston.prg ,10 ,lathe_pick
```

The # represents a commented line which is ignored while reading the process plan. Each field is updated in the respective fields in the PP array by reading each line at a time. Any one of these fields may be blank. All tabs and spaces are ignored and all letters are converted to upper case letters.

Returns
None

Example
PP_file = fopen("feeder1plan","r");
get_pp(PP-file);
fclose(PP_file);

machine_name
Gets the name of the machine.

Synopsis
char* machine_name(int machineNum)

Arguments
int machineNum
machine identification number

Returns
Pointer to the machine name.

Description
The machine_name function gets the name of the machine from the Machine_map structure for the corresponding machine identification number.

Example
char* feederName;
feederName = machine_name(MILL);

C.4 FLEXIS.DCLS

Library routines that are used only in the Flexis environment are defined in flexis.dcls library.

To use these library files, the following files have to be included in Actor C declarations:

#include "system.dcls"
wait-replies
Wait for replies from a machine, siman, timer or any combination.

Synopsis
bool wait_replies(int machineTN, int simanTN, TIMER timer)

Arguments
int machineTN
   Transaction number of the request sent to a machine.
int simanTN
   Transaction number of the request sent to SIMAN.
TIMER timer
   To start a TIMER specified by timer.

Description
The wait_replies function checks for the replies from all the requests that are sent. This is implemented by using wait_reply from the msg.lib library and TIMER from flexis.

Returns
Returns true if all the replies from the request destination arrive.

Example
wait_replies(ExecMill, ExecSMill, ExecTimer);

Note: ExecMill and ExecSMill are integers globally declared for the macro. ExecMill and ExecSMill are transaction numbers returned by the request functions. ExecTimer is a Flexis timer set to run in Flexis environment.
APPENDIX D

HOW TO USE AND RUN CIMLAB CELL CONTROLLER

D.1 CIM LAB LOGINS

Appendix D describes the hierarchial structure of CIMLAB logins and how to run the Cell Controller. To develop the CIM lab two users, devel and demo are created in alpha, a sun SPARC Station 2 networked with internet.

D.1.1 demo login:

The demo login is used when the CIM lab is used for the demo purpose. The hierarchial structure described in section D.2 is applicable to demo login, except for a sub-directory demolispvm in flexis directory. To install a new demo, all the files from the devel login are copied to the respective directories. The flexis virtual memory file, lisp.virtualmem, is copied from demolispvm in devel login. When new demo files are installed, the lisp.virtualmem has to be loaded with the new Flexis actors. Note: The path for the process plans has to be changed in the Actors.

D.1.2 devel login:

The devel login is used when the CIM lab is in the development and creating of new programs. The hierarchial structure is described in section D.2.

The simulation software, SIMAN, has permission only in phoenix, another sun SPARC Station 10, so demo and devel logins are created in phoenix for SIMAN, and it is remotely executed from alpha.
D.2 CIM LAB LOGINS HIERARCHIAL STRUCTURE

Figure D.1 shows the hierarchial structure built for the CIM lab.

**beta**

*beta* is created to have mill programs. The directory is mounted from the PC, beta, and it is used to load the programs for CNC Mill Machine.

**gamma**

*gamma* is created to have lathe programs. The directory is mounted from the PC, gamma, and it is used to load the programs for CNC Lathe Machine.
**delta**  
*delta* is created to have different robot programs. The directory is mounted from the PC, *delta*.

**pc**  
*pc* is created to have a source code and executable programs for the robot and cnc machines. *pc/bin* directory contains the executable files for the execution of the robot and cnc machines. *pc/src* directory is further divided into cnc and robot to have the source codes for the programs to run cnc machines and the robot. The *pc/bin* directory is mounted from the PC's to execute the programs. The *pc/src* directory is mounted from the PC's only when the CIM lab programs are in the development stage.

**server**  
*server* is created to have source code and executable programs for the server. *server/bin* directory contains the executable files to start the server. *server/src* contains the source code for the server programs.

**flexis**  
*flexis* directory has sub-directories lispfiles, include, plans, user, temp and demolispvm.

  *lispfiles* directory contains the flexis files, dot files and shell programs to execute the programs.

  *user* directory contains sub directories for all the flexis actors, created by the flexis runtime.

  *include* directory contains the include files for the flexis programs.

  *plans* directory contains the feeder process plans.

  *temp* is a directory created by the flexis runtime to run the programs in remote execution.
The `demolispvm` directory contains the lisp virtual memory file, a default file to load when the programs are installed in demo login. Flexis saves the lisp virtual memory file in the login directory, so `lispfiles` directory is used as the login directory. When the Flexis program is running in the Local Execution mode, the flexis has a default to look into the login directory for files. When the Flexis program is running in the Remote Execution mode, the flexis looks into the files from the `user` directory. So, the directories `lispfiles` and `user` are created in the same hierarchial level to share the include files and the process plans.

### D.3 HOW TO START CELL CONTROLLER

Shell programs are written to run different CIM lab programs. `gocim` is a shell program which opens different windows for flexis, runtime, server and cinema and start all the programs. To run individual programs, `goflexrun`, `goserver` and `gosiman` are used. `goflexrun` runs only the flexis and runtime programs, `goserver` runs only the server with the options of selecting the server to run with realtime machines and/or cinema and `gosiman` runs only the CINEMA program.

#### D.3.1 Local Grafcet Execution

To run the program in local execution, `runtime` is not needed. The programs run slow in local execution. A click with the left mouse button on the Execute in the Process Directory menu gives the Local Execution menu. A click with the left mouse button on `start` starts the Local Execution with default flexis variables; i.e., `Device` set to
MachineAndSiman and LoadParts set to No.

D.3.2 Remote Grafcet Execution

To execute the programs by remote first, the runtime program is started. The middle mouse button is slid on the Execute in the Process Directory menu to get the Remote Execution menu. A click with the left mouse button on connect connects the lisp.virtualmem to the runtime. A click with the left mouse button on Load loads the actor programs in runtime environment. A click with left mouse button on Continue starts the Remote Execution with default flexis variables; i.e., Device is set to MachineAndSiman and LoadParts is set to No.

D.3.3 How to Set Flexis Variables

In Flexis, two enumeration variables, Device and LoadParts have to be set when it is started. A click with the middle mouse button on the variables displays the different values that the variable can have. A click with the left mouse button on the required value selects the value for the variable.

Device: The variable Device has four values Machine = 1, Siman = 2, MachineAndSiman = 3 and Flexis = 4. MachineAndSiman is assigned 3 to have a bitwise verification, to select both Machine and Siman. The values are used to communicate with Realtime Machines, Siman and Flexis. If the value selected is Machine, Cell Controller talks to Realtime Machines and to the Flexis actors; if the value selected is Siman, it talks
to Siman and to the Flexis actors; if the value selected is MachineAndSiman, it talks to Realtime Machines, Siman and Flexis actors; and, if the value selected is Flexis, it talks only to the Flexis actors. The default value is set to MachineAndSiman.

**LoadParts:** The variable LoadParts has two values: Yes = 1, No = 2. The values are used to know whether parts are loaded into Feeders. If parts are loaded in a Feeder, the respective feeder LoadParts variable is set to Yes. Once the LoadParts is set to Yes, the Feeder actors inform about the parts to the Scheduler actor to process its part.

### D.4 HOW TO START CIMLAB

The CIMLAB Cell Controller runs with Siman, Machine or both. Different Shell programs are written to start different programs. The required program is started to run the CIMLAB.

- **gocim** This shell program starts four windows and starts Flexis, Runtime, Siman and Server in each window. It prompts for the Server options; i.e., whether to run Server with options for Siman or MachineAndSiman.

- **goflexrun** This shell program starts two windows and starts Flexis and Runtime in each window.

- **gosiman** This shell program starts a window with Siman CINEMA program.

- **goserver** This shell program starts a window with Server. It prompts for
options of selecting the server with realtime machines, Siman or for an empty window.

The following steps describe how to start the CIM laboratory:

D.4.1 With Siman:

Step 1:       Start Openwindows.
Step 2:       Start go
cim and give N (No) to the Server options to start the Server for only Siman. Note: go
cim starts runtime also.
Step 3:       Start the Flexis as described in Section D.3 for Remote Grafcet Execution.
Note: Set Device = Siman.
Step 4:       Start the CINEMA.
Step 5:       When the execution is over, Exit the CINEMA, which also kills the Server. Exit the Flexis as described in Section D.5. Exit Runtime by typing shutdown in runtime window.

D.4.2 With Machines:

Step 1:       Start Openwindows.
Step 2:       Start go
cim and give any key to get an empty window for the Server. Start the server with options -fi1 \-i101 -i102 -i103. Note: go
cim starts runtime also.
Step 3:       Start the CNC Machines, Robot and the beta, gamma and delta PC's.
Step 4:       Start the Flexis as described in Section D.3 for Remote Grafcet Execution.
Step 5: In PC's goto cimlab directory, and run demo or devel to mount the required directories for the PC's.

Step 7: Run run to start the programs. Note: To simulate the RealTime Machines, run sim to start simulation programs.

Step 8: When the execution is over, Stop and Initialize the Flexis actors which kill the Server. Exit the Flexis as described in Section D.5. Exit Runtime by typing shutdown in runtime window.

D.4.2 With Machines and Siman:

Step 1: Start Openwindows.

Step 2: Start gocim and give Y (Yes) to the Server options to start the Server for Siman and RealTime Machines. Note: gocim starts runtime also.

Step 3: Start the CNC Machines, Robot and the beta, gamma and delta PC's.

Step 4: Start the Flexis as described in Section D.3 for Remote Grafcet Execution. Note: Set Device = MachineAndSiman.

Step 5: Start the CINEMA.

Step 6: In PC's goto cimlab directory, run demo or devel to mount the required directories for the PC's.

Step 7: Run run to start the programs. Note: To simulate the RealTime Machines run sim to start simulation programs.

Step 8: When the execution is over, Exit the CINEMA, which also kills the
Server. Exit Flexis as described in Section D.5. Exit Runtime by typing shutdown in runtime window.

D.5 HOW TO EXIT CELL CONTROLLER

D.5.1 Local Grafcet Execution:

(1) A click with the left mouse button on stop stops the Cell Controller.

(2) A click with the left mouse button on initialize initializes the Cell Controller, which in turn terminates the connection with the Server.

(3) Drag the middle mouse button on the Logout in the Process Directory menu to get a pop-up menu; select Logout and DO NOT SAVE STATE to exit the Cell Controller without saving the current state or select Logout and save state to exit the Cell Controller and to save the current status of the Controller. Generally if the login is demo, it is Logout without saving the current status; otherwise, if the login is devel, it is Logout with save.

D.5.2 Remote Grafcet Execution:

(1) A click with the left mouse button on stop stops the Cell Controller.

(2) A click with the left mouse button on initialize initializes the Cell Controller, which in turn terminates the connection with the Server.

(3) Drag the middle mouse button on the Load to get a pop-up menu and select Unload to remove the programs from Runtime environment.

(4) Drag the middle mouse button on the Connect to get a pop-up menu and
select *Disconnect* to terminate the connection with the runtime.

(5) Drag the middle mouse button on the *Logout* in the Process Directory menu to get a pop-up menu, select *Logout and DO NOT SAVE STATE* to exit the Cell Controller without saving the current state, or select *Logout and save state* to exit the Cell Controller and to save the current status of the Controller. Generally if the login is *demo*, it is Logout without saving the current status; otherwise, if the login is *devel*, it is Logout with save.

**D.6 COMMON ERRORS**

1. If the Cell Controller is started without starting the Server, *flexis* gives a *INTERLISP-ERROR* pop-up window with the Unix error number: 0. A middle mouse button on the pop-up window again displays a pop-up window; in it, select the *up arrow* to stop the process. Initialize the program and start with the Server.

2. Sometimes in the Remote Grafcet Execution when Flexis is connected to the Runtime, an E7 Error occurs. This error occurs probably due to not unloading the *flexis* actors in previous execution or runtime is not properly shutdown. This is overcome by exiting the *flexis* and rebooting the system.