An Object-Based Run-Time Executive for Control of Flexible Manufacturing Systems

Jonghun Park[1], Jin-Woo Park[2], Joongwon Woo[3], Jongwon Kim[3] and Wook-Hyun Kwon[1]

[1]Engineering Research Center for Advanced Control & Instrumentation,
[2]Department of Industrial Engineering, [3]Department of Mechanical Design & Production Engineering,
Seoul National University,
San 56-1, Shilim-Dong, Kwanak-Gu, Seoul, 151-742, Korea
e-mail : jonghun@asri.snu.ac.kr

ABSTRACT

We present a new design specification of a run-time executive for control of flexible manufacturing system (FMS)s, building on object-based techniques. Classes required to control FMS activities are defined and implemented based on the strategy which separates the control and information objects from the FMS physical objects [9]. Several control objects which could be served as reusable software elements in implementing case-specific FMSs are identified and defined. The primary objective of our system is to achieve a capability for dynamic reconfiguration to the change of FMS configuration, scheduling and control logic, and communication platform, for the control of FMS. We have used C and C++ as base languages under UNIX environment for implementing this object-based run-time executive.

1. INTRODUCTION

An FMS consists of a group of processing stations (e.g. CNC machine tools) interconnected by means of an automated material handling and storage system and also controlled by an integrated computer system [2]. In order to exploit its full flexibility while retaining productivity, the control software which responds to real-time events by identifying short-term requirements and generates control commands accordingly in run-time, plays an important role [8].

Unfortunately the current state of the art in design and implementation of the control software for an FMS suffers from the complexity that arises as the configuration of resources and/or part types change. In addition to that, although the machine tools, robots, and computer hardware necessary for FMS are generally available as stock items, the control software for these systems must be developed specifically for each system [13].

This kind of software falls into the category of large-scale distributed program which is difficult to develop and maintain [12]. Therefore *reusability* and *extensibility* become critical properties, since most FMS control software systems are used for long periods of time and are frequently modified, for instance, to add new resources or change part types produced.

In this line of research, there have been works on object-orientation in building manufacturing softwares [2], [4], [8], [9], [11], [13]. Lin [8] developed object-oriented software for control of a robotic flexible manufacturing cell (FMC). Smith [13] presented concepts of object-orientation in a scaleable FMS control architecture, automatically generated contol software, and object-oriented programming of equipment components.

The objective of this paper is to present a new design specification of an object-based run-time executive which has a *dynamic reconfiguration* capability to the change of FMS configuration, scheduling and control logic, and communication platform, for control of an FMS. By applying object-based technique and designing modular functional objects, we could make software systems more understandable, extensible, and reusable. Software reuse could provide tools necessary for manufacturing engineers to design, implement, and maintain customized FMS control softwares in their environment [13].

0-7803-2559-1/95 $4.00 © 1995 IEEE

2. THE OBJECT MODELING

The classes identified in our system are depicted in Figure 1 which shows the O-R (Object-Relationship) model [6] used to extract classes in this application.

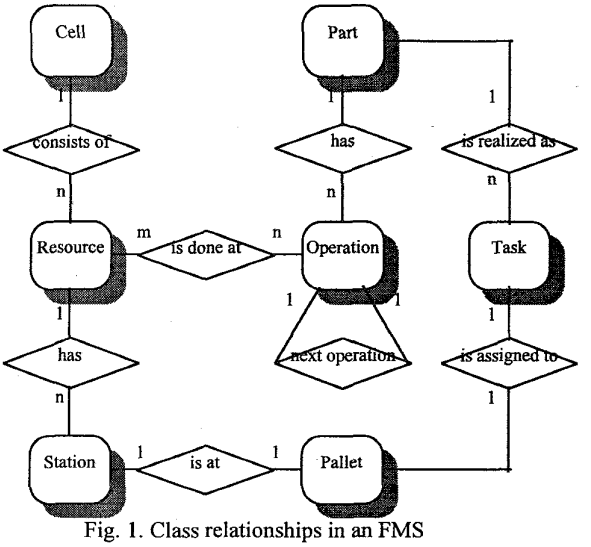

Fig. 1. Class relationships in an FMS

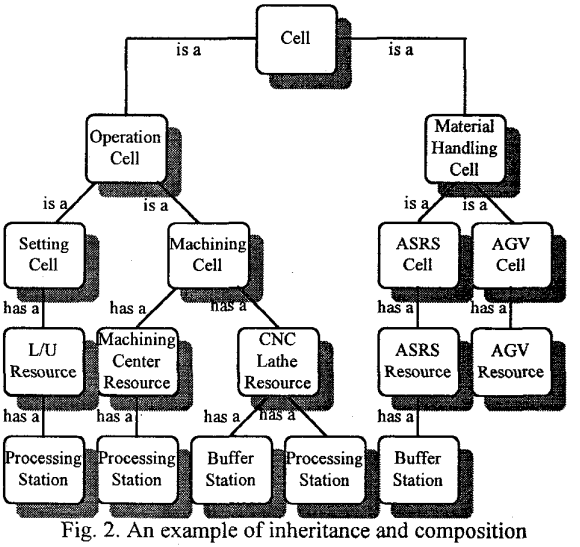

Fig. 2. An example of inheritance and composition hierarchy for Flexible Manufacturing Cell (FMC)s

The class *part* has class *operation(s)* which can be performed by class *resource(s)*. The class *cell* consists of the *resource(s)* which may have one or many *stations*. The class *pallet* can be located at the *station*. The class *task* is devised here to represent a job the control software could construct, update, schedule, dispatch and finally destruct. A task object is created when a pallet is setup with workpiece to start its processing and destroyed when the workpiece in the pallet has completed its processing requirements. Our class design strategy is based on Mize's work [9], where three components describing a system (physical, information and control objects) are defined. Furthermore, several control objects are defined and discussed later in this paper.

Figure 1 shows the abstract classes which are mainly used to define public interfaces [3]. The concrete classes can be inherited from the abstract classes which are denoted by *is a* relation, in Figure 2. The classes which have composition hierarchy are denoted by *has a* relation. In Figure 2, physical components which can be found in typical FMS configuration are depicted.

3. THE DESIGN OF RUN-TIME EXECUTIVE

Since it is commonly found in a real situation that an FMS is composed of several FMCs which are interconnected by central material handling system and communication network like ethernet, we have designed dedicated processes to communicate between FMS control software and softwares in FMCs. Figure 3 shows this situation where *dispatcher* process in FMS control software dispatches a command to FMCs and *monitor* process monitors an event from them. Due to the asynchronous nature of sending and receiving of events and commands, each control software in FMCs has reporter and monitor processes repectively.

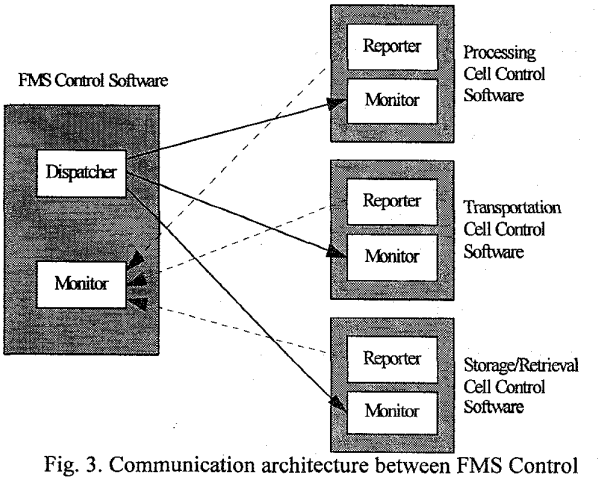

Fig. 3. Communication architecture between FMS Control Software and constituent FMCs

An architecture of object-based run-time executive for control of an FMS is shown in Figure 4 where rectangles represent processes, and rounded ones represent objects. The arrows are used to represent process dependency, whereas dotted arrows to represent

inter-process communication (IPC) message flows. Active objects such as *dispatcher*, *monitor*, *system supervisor* and *user interface manager* are implemented as UNIX processes.

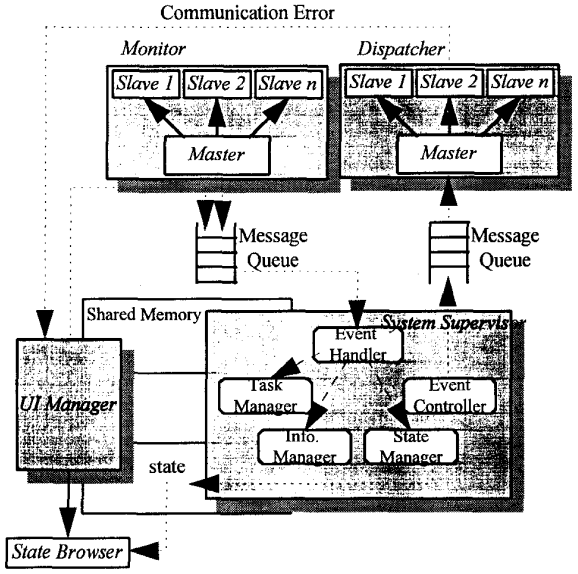

Fig. 4. Architecture of an object-based run-time executive

As mentioned above, dispatcher and monitor objects take care of sending and receiving IPC messages to and from managed FMC hosts respectively. They are concurrent servers which fork their slave processes whenever it is required to communicate with outer reporter and monitor processes in FMC hosts. Dispatcher and monitor objects are tightly coupled by message queue with *system supervisor* object [5]. The system supervisor object gets the message which contains significant FMS events (e.g. operation finished, resource failure, and so on) from incoming message queue and puts command messages (e.g. start transportation, start operation, and so on) to outgoing message queue for coordinating and synchronizing operations of its managed cells (FMCs).

The user interface process (UI Manager) displays interface window where an user can send commands such as start and stop of FMS operations to system supervisor (a dotted line from UI Manager to message queue in Figure 4). Also, if there is an error during communication with FMCs, it is reported to the user interface process. State browser is the child process of user interface process. It is tightly coupled with the system supervisor process to display current system state to user screen through shared memory.

The system supervisor contains several control objects as shown in Figure 5. The primary responsibility of event handler object is to dequeue event from message queue shown in Figure 4 and to invoke other control object's methods according to the event type dequeued. The *state manager* object has a method to invoke the state change of virtual cells and also notify this state change to the event controller. The term *virtual cell* is used to denote the software object which represents a physical cell in an FMS concerned. The primary advantage of using virtual cell object is the capability to encapsulate their own attributes and scheduling methods which could be different according to nature and configuration of the cells. The task manager manages the state of task and pallet objects, and issues the operation request to the event controller as the state of a task is changed. To be more precise, if a task has two consecutive operations (e.g. milling after turning) which must be done at two different cells (e.g. turning cell and milling cell), the task manager first requests turning operation to the event controller, and then requests milling operation after completion of turning successfully.

Finally, event controller has methods for controlling FMS activities; 1) a *request_resource* method where *enqueue_reqeust* method of virtual cells is invoked to make the task a candidate to be serviced by a resource of that cell whenever a task manager requests an operation, 2) a *command_operation* (or *command_mtl. handling*) method where *dequeue_request* method of virtual operation (or mtl.handling) cells is invoked to schedule and dispatch commands for tasks, waiting for its service whenever a resource in one cell becomes ready for operation, and 3) a *notify_event* method which is invoked to notify an event from one cell to the other cells.

4. THE RUN-TIME EXECUTIVE AT WORK

Figures 6 and 7 illustrate the cases when each of two important types of event is dequeued by the event handler object respectively. In these figures, vertical dotted lines represent each control object and arrows denote messages passing among these objects. Whenever the event message *operation finish* is dequeued from message queue, the *manage_task* method of the task manager is invoked at first. After identifying the next operation required for that task, the task manager invokes *request_resource* method of the event controller object.

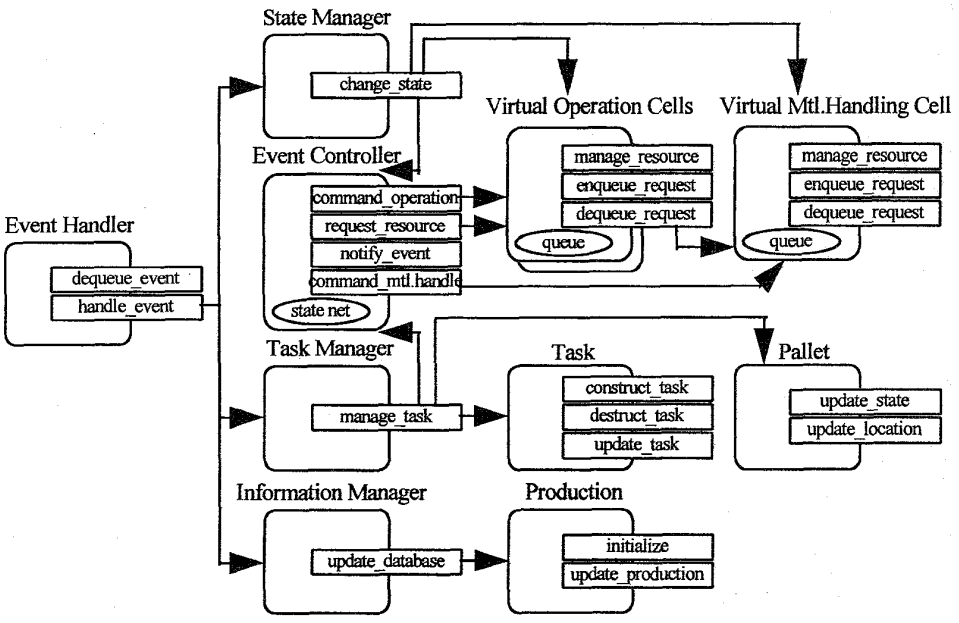

Fig. 5. Control objects in system supervisor process

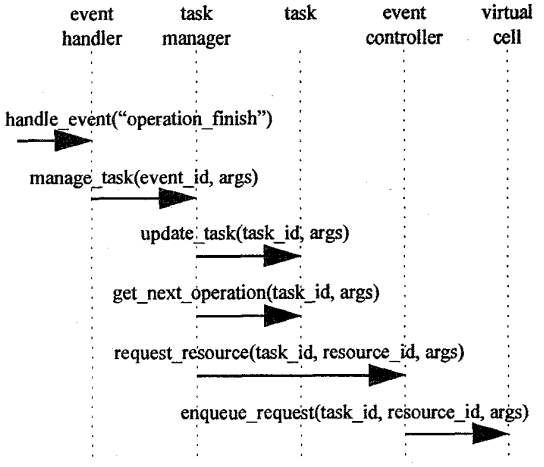

Fig. 6. An example of message passing among control objects when an "operation_finish" event is received

Since the event controller object has a state net for the FMS concerned as a private member, it could generate appropriate resource request to the virtual operation cell or buffering request to the virtual material handling cell without incurring deadlock situation [7]. We have adopted the deadlock detection and recovery policy, presented by Kumaran et al. [7], with slight modification.

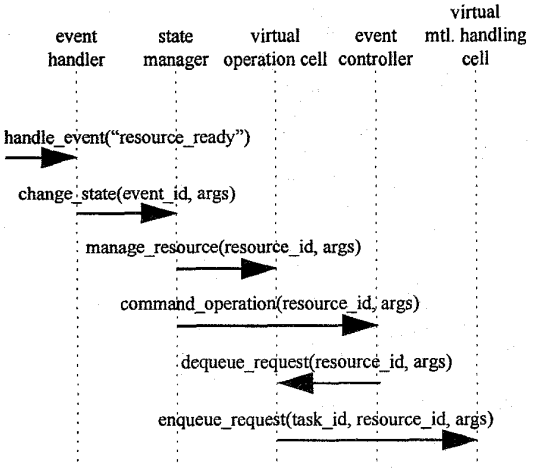

Fig. 7. An example of message passing among control objects when an "resource_ready" event is received

The situation when the event message *resource ready* (which implies that resource is idle and there is no pallet on one of its local buffer where an operation can be performed) is dequeued from message queue, is illustrated in Figure 7. The *change_state* method of the state manager updates the system state in shared memory and then invokes *manage_resource* method of virtual operation cell to update the local state of virtual cell. The *dequeue_request* method of the virtual cell actually dispatches commands into outgoing message queue after

selecting one from its internal request queue by its own look-ahead scheduling rule.

Note that actual dispatching is deferred to the time when the corresponding resource becomes ready, not to the time when the operation of a task is finished for endowing look-ahead property to the scheduling method of virtual cell object.

A simple case example is given with an illustration in Figure 8, where three cells are shown. The cells A and B are assumed to be virtual operation cells and the cell C be virtual material handling cell. Each cell is assumed to have only one resource respectively. There are three tasks *a*, *b*, and *c* in our example. Task *a* consists of operations which require the resources in cell A and cell B in turn. To transport a pallet from cell A to cell B, the material handling resource in cell C is used.

At stage (1) in Figure 8, task *a* is being processed in cell A, where task *b* is in queue waiting for its operation to be processed by cell A. Cells B and C are in ready state which means that the resources are in idle state and no pallet is on it. If the operation of task *a* is finished by cell A, then the resource of cell A changes its state to *idle* and the request of task *a* for cell B (for its next operation) is enqueued into the queue (Stage (2)). Because the cell B is in ready state, it can dequeue one request from request queue. There are two requests currently ; one is from cell A for task *a* and the other is from other operation cell for task *c*. If we assume that the request of task *a* is dequeued rather than *c* according the cell B's own scheduling rule (e.g. SPT : Shortest Processing Time [10]), cell B changes its state to *reserved* and enqueues material handling request for task *a* to cell C. Since cell C is ready at now, it immediately changes its state to *busy* and starts transportation (Stage (3)). It should be noted that the state of cell A is also changed into *reserved* because not only cell A has become *ready* when the pallet for task *a* is loaded on the material handling resource but also it has dequeued task *b* .

Finally at stage (4), the pallet for task *a* is transported to cell B. Now cell B starts the operation of task *a* and cell C starts transportation for task *b*.

Figure 9 presents a situation where dispatcher and monitor processes in our run-time executive and monitor and reporter processes in an operation cell (mc_monitor and mc_reporter) communicate following predefined synchronization message primitives. Thick lines indicate that the process is blocked until the next event or expected acknowledge message is received. Our strategy is command-driven rather than material handling-driven because the actual operation start is not possible until the required pallet is arrived.

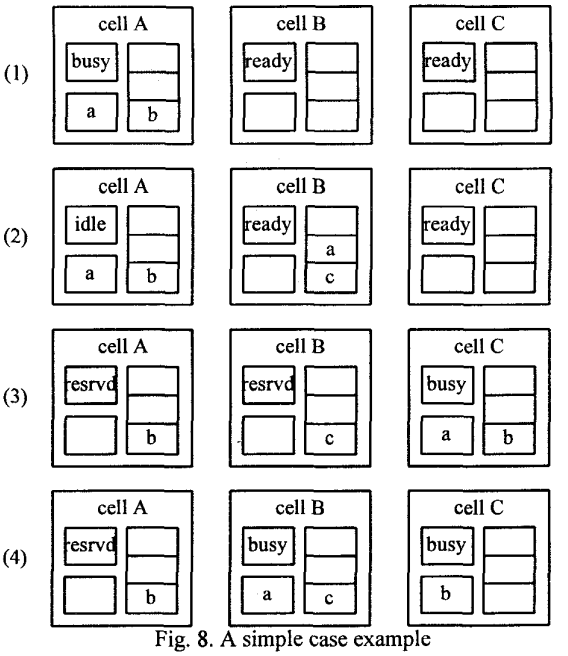

Fig. 8. A simple case example

To achieve better automatic failure solutions, the synchronization message primitives are designed in such a way that run-time executive could find out the source of errors (e.g. communication error or resource error), as pointed out by Adlemo [1]. For example, we have designed protocol in which the reporter process in a material handling cell (e.g. AGV : Automated Guided Vehicle) sends multiple responses for single transportation command (See Figure 10). The reporter process sends *inbound_start (inbound_finish)* message when the transporter starts (finishes) moving from its current location to the source station where the pallet is, and *outbound_start (outound_finish)* message when the transporter starts (finishes) transportation (moving with the pallet loaded) from the source station to the destination station where the pallet should be unloaded from the transporter. Therefore, if there is an error message from the reporter process during command execution, the run-time executive can identify the context of the error, which makes it possible to perform more intelligent failure recovery actions.

5. CONCLUSION

We have focused on the design specification of a run-time executive for control of operations in flexible manufacturing systems (FMSs). By applying the object-based design methodology, several control objects along with physical and informational ones, which could be served as reusable software components in designing

and implementing case-specific FMS controllers, have been defined. This design strategy provides more modularity among the software components, hence guarantees dynamic reconfiguration capability to the changes of physical, informational and control-related FMS components. We have used C and C++ as base languages under UNIX environment for implementation of our run-time executive and X11 / Motif library for operator interface. This control software has been implemented in our pilot FMS plant of Seoul National University successfully and now under running.

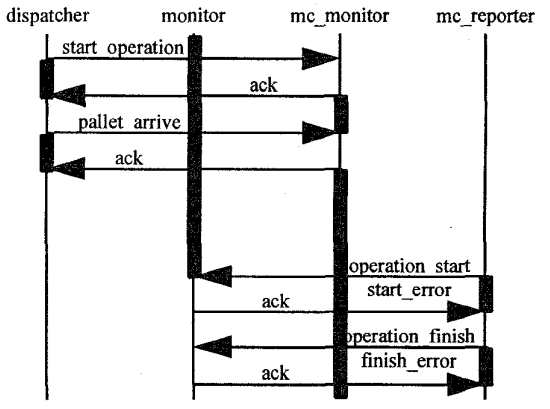

Fig. 9. Command-driven operation strategy

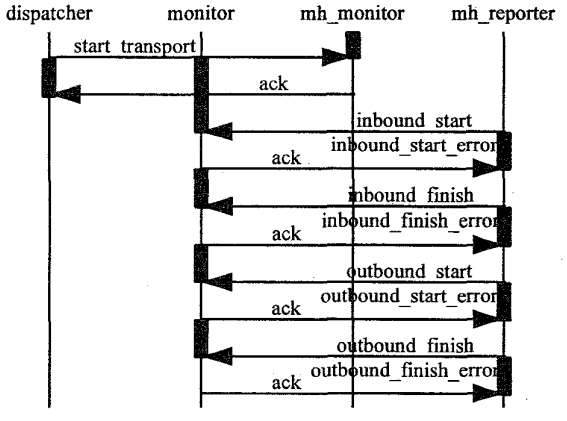

Fig. 10. Multi-level communication protocol

6. REFERENCES

[1] A. Adlemo and S-A. Andreasson, "Communication Protocols and Failure Semantics in Intelligent Manufac turing Systems", *Proceedings of the 1994 IEEE International Conference on Robotics and Automation*, 1994, pp.3453-3458

[2] S. Aggarwal, S.Mitra and S. S. Jagdale, "Specification and Automated Implementation of Coordination Protocols in Distributed Controls for Flexible Manufacturing Cells", *Proceedings of the 1994 IEEE International Conference on Robotics and Automation*, 1994, pp.2877-2882

[3] G. Booch, *Object Oriented Design with Applications*, The Benjamin / Cummings Pub. Co., 1991

[4] M. Fabian and B. Lennartson, "Object-Oriented Structuring of Real Time Systems", *Proceedings of the 31st Conference on Decision and Control*, 1992, pp.2529-2530

[5] D. Gauthier, et al, "Interprocess Communication for Distributed Robotics", *IEEE Journal of Robotics and Automation*, Vol. RA-3, No.6, 1987, pp.493-504

[6] A. Kemper, and G. Moerkotte, *Object-Oriented Database Management : Applications in Engineering and Computer Science*, Prentice Hall, 1994

[7] T. K. Kumaran, W. Chang, H. Cho and R. A. Wysk, "A structured approach to deadlock detection, avoidance and resolution in flexible manfuacturing systems", *International Journal of Production Research*, Vol. 32, No. 10, 1994, pp.2361-2397

[8] L. Lin, M. Wakabayashi, and S. Adiga, "Object-oriented modeling and implementation of control software for a robotic flexible manufacturing cell", *Robotics & Computer-Integrated Manufacturing*, Vol.11, No.1, 1994, pp.1-12

[9] J.H. Mize, et al, "Modeling of integrated manfuacturing systems using an object-oriented approach", *IIE Transactions*, Vol.24, No.3, 1992, pp.14-26

[10] M. Montazeri and L. N. Van Wassenhove, "Analysis of scheduling rules for an FMS", *International Journal of Production Research*, Vol. 28, No. 4, 1990, pp.785-802

[11] S. Y. Nof, "Critiquing the potential of object orientation in manfuacturing", *International Journal of Computer Integrated Manufacturing*, Vol. 7, No. 1, 1994, pp.3-16

[12] S. M. Shatz, *Development of Distributed Software : Concepts and Tools*, Macmillan Pub. Co., 1993

[13] J. S. Smith and S. B. Joshi, "Reusable software concepts applied to the development of FMS control software", *International Journal of Computer Integrated Manufacturing*, Vol. 5, No. 3, 1992, pp.182-196