EXTENDING THE DARTS SOFTWARE DESIGN METHOD TO DISTRIBUTED REAL TIME APPLICATIONS

Hassan Gomaa

George Mason University
School of Information Technology and Engineering
Fairfax, VA 22030

ABSTRACT

This paper describes a software design method for distributed real time applications which typically consist of several concurrent tasks executing on multiple nodes supported by a local area network. The design method is an extension of DARTS, the Design Approach for Real Time Systems, and is called DARTS/DA, DARTS for distributed applications. The method starts by developing a data flow model of the distributed application using Structured Analysis. The next stage involves decomposing the application into distributed subsystems based on a set of subsystem structuring criteria and defining the interfaces between them. Next each subsystem is structured into concurrent tasks using the DARTS task structuring criteria and the interfaces between tasks are defined. Finally, each task, which represents a sequential program, is structured into modules using the Structured Design Method. As an example, DARTS/DA is applied to the design of a distributed Factory Automation System.

1. INTRODUCTION

This paper describes a design method for distributed real time applications which execute on multiple nodes supported by a local area network. The design method is an extension of DARTS, the Design Approach for Real Time Systems [1,2] and is called DARTS for Distributed Applications, DARTS/DA. The method emphasizes concurrent processing and data abstraction.

DARTS/DA provides a methodical procedure for arriving at a design in which the application is structured into subsystems which consist of concurrent tasks. In particular, it provides a set of criteria for structuring an application into subsystems and a further set of criteria for structuring subsystems into concurrent tasks. A subsystem is defined as a collection of concurrent tasks executing on one node. Tasks within a subsystem may communicate by means of messages, event synchronization and Information Hiding Modules, which support concurrent access to data stores. However tasks which are in separate subsystems may only communicate via messages [11].

The DARTS/DA design method starts with a black box Requirements Specification [2] which defines what features the system will provide with no consideration of how they will be provided. DARTS/DA is an iterative design method and consists of the following steps:

(1) Perform Data Flow Analysis using Structured Analysis.

(2) Structure system into (potentially) distributed subsystems.

(3) Define Interfaces between subsystems

(4) Structure each subsystem into concurrent tasks.

(5) Define interfaces between tasks.

(6) Design each individual task using Structured Design.

The method is oriented towards applications which have a significant component of real time data collection and/or control. These applications are therefore categorized as operations systems rather than information systems [12]. No assumptions are made about the availability of a distributed data base. The application is distributed primarily on a functional basis. One of the objectives of the design method is to structure the distributed application so that relatively independent subsystems may use a local data base if they need to. Distributed access to the data is provided via a server.

The DARTS/DA design method is described in Sections 2 and 3. An example of using DARTS/DA in the design of a distributed Factory Automation System is described in Section 4. The paper concentrates on the structural aspects of a design and does not address the dynamic aspects of a design. This issue is addressed in a previous paper [2], where the use of event sequence diagrams can help the designer perform a timing analysis.

2. Structuring a Distributed Application into Subsystems

2.1. Data Flow Analysis

Data flow modelling is used as an analysis tool. Starting with the functional requirements of the system, the data flow through the system is analyzed and the major functions are determined. The data flow diagrams [3,4] are developed and hierarchically decomposed to sufficient depth to identify the component subsystems and their interfaces.

TH0212-1/88/0000/0252$01.00 © 1988 IEEE

Each data flow diagram (DFD) consists of:

(1) Transform bubbles representing functions carried out by the system.

(2) Arrows representing data flows between transforms.

(3) Data stores representing repositories of data.

Entity-Relationship diagrams are also developed to define the relationships between the data stores. A Data Dictionary is used to define the data items contained in the data flows and data stores.

2.2. Decomposition Into Subsystems

A subsystem consists of one or more concurrent tasks. By definition, all tasks that are part of one subsystem execute on the same node. For example, two subsystems would be designed such that they each could run on a separate node or alternatively they could both run on the same node.

The structuring of an application into subsystems is based on functional decomposition. Thus in analyzing the data flow model we look for transforms which display functional cohesion, i.e. a group of transforms which perform closely related functions. We also look for data stores, around which a service can be provided.

The following criteria are used for identifying subsystems. A subsystem may satisfy more than one of these criteria.

(1) Functionality. A subsystem performs a well defined function or closely related group of functions. The data traffic between these functions may be high, so that structuring them into separate subsystems would potentially increase system overhead.

(2) Server. The subsystem provides a service. It responds to requests from client subsystems. It does not initiate any requests. Frequently, the server provides services which are associated with a data store.

(3) Agent. An agent subsystem [13] provides an indirect service. In order to perform the service, it has to make requests of other subsystems. Thus it acts as an intermediary between a client and a server.

(4) Proximity to the source of physical data. This ensures fast access to the physical data and is particularly important if data access rates are high.

(5) Localized control. In some cases the subsystem performs a specific site related function. Often the same function is performed at multiple sites. Each instance of the subsystem resides on a separate node which provides greater autonomy and local control. Assuming a subsystem operates relatively independently of other nodes, then it can be operational even if other nodes are temporarily unavailable.

(6) Performance. By providing a time critical function within its own node, better and more predictable performance can often be achieved.

(7) User Interface. With the proliferation of workstations and personal computers, a subsystem providing a user interface may run on a separate node, interacting with subsystems on other nodes. A user interface subsystem performs an actor role [13].

The roles of server, actor and agent subsystems are analogous to those defined for objects by Booch [13].

2.3. Define Subsystem Interfaces

Each subsystem consists of one or more concurrent tasks. By definition all tasks within a given subsystem always execute on the same node. Thus tasks within the subsystem may use the DARTS inter-task communication/synchronization mechanisms. However, as subsystems can reside on separate nodes, all communication between tasks in different subsystems must be restricted to message communication.

Starting with the data flow diagram, subsystem interfaces will be in the form of data flows or data stores. A data flow that crosses subsystem boundaries needs to be designed as a message. As tasks on separate nodes may not access a shared data store directly, a data store that is accessed by more than one subsystem needs to be treated in one of two ways:

(1) Central Server. If the data store is updated relatively infrequently, and data is typically accessed by other subsystems more frequently than it is updated, then the data store is encapsulated in a Server subsystem. In this case the data is centralized and the access to is distributed.

(2) Distributed Server. It is sometimes the case in distributed applications that data collection is distributed because the data originates at several different locations. In a real-time data gathering environment, it is also often the case that the data is updated relatively frequently. In this situation, it is often more efficient to store the data at the source of data collection and let the data be accessed from remote locations. In this case, the data is distributed and the access to it is also distributed. To address this case, a Distributed Server capability is provided which is described next.

2.4. Distributed Application Services

2.4.1. Distributed Server

The Distributed Server provides the same services to client subsystems as the Central Server. However it hides the fact that the data that an application task needs to reference may be stored in several different locations. Thus the Distributed Server allows an application task to access distributed data for read or write without having to know where the data physically resides. The application refers to the data item by name. A layer of distributed services is provided at each node which maintains a Data Configuration Table which identifies where each data item is located. More information on servers and distributed systems is given in [14].

2.4.2. Distributed Task Management

It should be transparent to a given task as to where the destination task it is communicating with resides. Some commercial distributed operating systems, e.g. VAX/ELN, provide this capability. If this is not available, then a Distributed Task Manager (DTM) needs to be developed to provide this transparency. This layer of software sits above the operating system on each processor. It maintains a table which relates task name to node id. All inter-task communication is routed via DTM. More information on distributed operating systems is given in [15].

3. Subsystem Decomposition Into Tasks

After identifying the subsystems in the distributed application, the next step is to design each subsystem. Each subsystem consists of one or more concurrent tasks, which by definition execute on the same node. Thus, the DARTS design method may be used to design each subsystem.

3.1.1. Real Time Structured Analysis

If a given subsystem has a real time component, then the real time extensions to Structured Analysis may be used [7, 8]. Otherwise, traditional Structured Analysis [3, 4] is used. Depending on the size of the subsystem, a data flow diagram, or hierarchical set of data flow diagrams, are developed.

In Real Time Structured Analysis, the following extensions are provided in addition to those described in Section 2.1:

(1) Event flows (indicated by dashed lines) represent control flow. Three types of event flows are provided:

- Triggers which signal an event occurrence thereby triggering a transform to perform a specific action.

- Enable, which activates a transform. The transform remains enabled until it is disabled by a subsequent event flow.

- Disable, which de-activates a previously enabled transform.

(2) A control transformation (dashed circle) which may only receive trigger event flows as inputs. However it can generate trigger, enable or disable event flows as output. A control transformation is defined by means of a state transition diagram.

3.1.2. Task Structuring

The main consideration in decomposing a software system into concurrent tasks concerns the asynchronous nature of the functions within the system. The transforms in the DFDs are analyzed to identify which of these may run concurrently and which need to execute sequentially.

The criteria for deciding whether a transform should be a separate task or grouped with other transforms into one task are:

(1) Dependency on I/O.

A transform depending on input or output is often constrained to run at a speed dictated by the speed of the I/O device it is interacting with. In that case the transform needs to be a separate task. Typically there is one task per physical device.

Another type of dependency on I/O is user I/O. The speed of a task that interacts directly with a user is typically dependent on the speed of the user I/O.

(2) Time critical functions.

A time critical function needs to run at a high priority and therefore can be structured as a separate high priority task.

(3) Computational requirements.

A non-time critical computationally intensive function (or set of functions) may run as a lower priority task consuming spare CPU cycles.

(4) Functional cohesion.

Certain transforms may be grouped into a task, because they perform a set of closely related functions. The data traffic between these functions may be high, in which case having them as separate tasks could increase the system overhead. Instead, each function is implemented as a separate module within the same task. This ensures functional cohesion both at the module and task level.

(5) Temporal cohesion.

Certain transforms may perform functions which are carried out at the same time. Consequently, these functions may be grouped into a task so that they are executed each time the task receives a stimulus. Each function should be implemented as a separate module to achieve functional cohesion at the module level. These modules are grouped into a task thereby achieving temporal cohesion at the task level.

(6) Sequential cohesion.

Certain transforms may perform functions which must be carried out sequentially, even though their execution is separated in time. Each function should be implemented as a separate module to achieve functional cohesion at the module level. These modules are grouped into a task thereby achieving sequential cohesion at the task level.

(7) Periodic execution.

A transform that needs to be executed periodically may be structured as a separate task which is activated at regular intervals.

(8) Sequential Server.

A Server Task may provide a service for other tasks in the system.

(9) Concurrent Server.

To improve throughput, the services provided by a Server may be shared amongst several tasks.

3.2. Define Task Interfaces

We now need to consider the interfaces between tasks that are in the same subsystem. On the data flow diagrams, these interfaces are in the form of data flows, event flows or data stores. The next stage involves formalizing the task interfaces using the guidelines described in this section:

3.2.1. Data Flows

A data flow between two tasks is treated as one of the following:

(1) A loosely coupled message queue if one task needs to pass information to the other and the two tasks may proceed at different speeds.

(2) A tightly coupled message/reply, if information is passed from one task to the other, but the first task cannot proceed until it has received a reply from the other.

Figure 1 shows the message communication mechanisms supported. The graphical notation for loosely coupled and tightly coupled message communication are also shown.

3.2.2. Data Stores

A data store which needs to be accessed by two or more tasks within the same subsystem is encapsulated within an Information Hiding Module in which the data structure is defined as well as the access routines to the data structure. The IHMs are based on the concepts of Information Hiding and Data Abstraction [9, 10].

Figure 2 shows the graphical notation used in DARTS for an IHM. The data store is shown as a box. In this example, there are two access procedures, shown by the transforms A and B. The access procedures are executed by the task(s) which wish to access the data store. Since data in the store may be accessed by more than one task concurrently, it is necessary for the access procedures to provide the necessary synchronization of access to the data.

The case where the data store is to be accessed by tasks outside the subsystem is discussed in section 3.3.

3.2.3. Event Flows

An event flow is treated as one of the following:

(1) An event signal if only a notification of an event occurrence is required and no data transfer is required.

(2) The event flow is also treated as a message if the destination task can receive several event signals and the order in which they are received is important. Having a FIFO message queue ensures that event messages are received in the correct sequence.

The graphical notation used for task synchronization in DARTS is shown in Figure 3.

Events are used for synchronization purposes between tasks, where no actual information transfer is needed. A destination task may wait for an event occurrence. A source task may signal an event that activates the destination task.

Furthermore the synchronization mechanism is extended to allow one task to wait for any one of several events to be signaled. If any one event is signaled, the task is activated. A task may wait for events used for synchronization purposes only as well as events associated with message queues.

3.3. Structure of Server Subsystems

A Server frequently maintains a data store and responds to requests from other tasks to update or read data from the data store. One approach to providing the service is for the data store as well as the access routines to be encapsulated within an Information Hiding Server Task which receives messages from client tasks. This is the case of the Sequential Server and is shown in Figure 4a. There is one message type for each access procedure. The supervisory module of the Server Task interprets the message and based on message type calls the appropriate access procedure. The parameters of the message are used as the parameters of the access procedures.

An alternative approach is for the services to be provided by a Concurrent Server and hence shared amongst several tasks. Multiple readers may access a shared data store concurrently. However, only one writer may be allowed to update the shared data and only after the readers have finished. This is an example of the multiple readers and writers problem in concurrent processing. This case is shown in Figure 4b. Each service is performed by one task. There may be several instances of a reader task. The Server Supervisor allocates services to the appropriate service task.

3.4. Task Design

3.4.1. Structured Design

The next stage involves designing each individual task. Each task represents a sequential program. In this section we describe how the task could be designed using the Structured Design Method [5, 6]. Depending on the nature of the task, either Transform Centered or Transaction Centered Design is used. In addition, support for state transition diagrams is required for state dependent systems. Some other design methods focus almost exclusively on state transition diagrams, e.g. Statecharts [16]. However, in DARTS, the state transition diagram is one component of the design method. This stage is summarized here. More information is given in [1].

3.4.2. State Dependency In Transaction Processing

In Transaction Centered Design, the action taken on the incoming transaction depends only on the input data. In state dependent real time systems, the action to be taken depends not only on the incoming data but also on the current state of the system, i.e. it depends on what has happened before.

The solution to this problem is to introduce a module called the State Transition Manager (STM). The STM maintains the current state of the system. It also maintains a state transition table which defines all legal and illegal state transitions, as well as the actions to be performed.

In DARTS, the State Transition Manager is designed as an Information Hiding Module which hides the State Transition Table. The module also contains the access procedures which check the validity of requests and performs the state transitions.

3.4.3. Task Supervisory Module

A Task Supervisory Module is typically the main module of a task. In this module, the task waits for one or more events to occur. These may represent synchronizing events or message queuing events. Depending on circumstances, the task may wait for different events at different times.

When an event is triggered, the TSM calls a procedure to handle the event. The simplest case is for a task to wait on one message queue. When a message arrives it is read and the TSM determines the message type. For each message type there is a procedure. TSM calls the procedure and the parameters of the message becomes the parameters of the procedure.

4. Example of Using the DARTS/DA Design Method

4.1. Factory Automation System

As an example of using the DARTS/DA design method, a factory automation problem is considered. In a high volume low flexibility assembly plant, workstations are physically laid out in an assembly line (Figure 5). Parts are moved between workstations on a conveyor. A part is processed at each workstation in sequence. Since workstations are programmable, different variations on a given product may be handled. Typically, a number of parts of the same type are produced, followed by a number of parts of a different type.

Each workstation has a variety of automation equipment, some of which are microcomputer controlled. Typical components include a pick-and-place robot, for picking the part off and placing it on the conveyor, an assembly robot and a Programmable Logic Controller (PLC). A PLC typically controls an industrial process by executing a relay ladder logic program in which control decisions are made based on the values of sensor inputs and which results in output actuators being set. Sensors are used for monitoring operating conditions while actuators are used to switch automation equipment on and off.

The manufacturing steps required to manufacture a given part in the factory, from raw material to finished product, are defined in a process plan. The process plan defines the part type and the sequence of manufacturing operations. Each operation is carried out at a workstation.

The processing of new parts in the factory is initiated by the creation of a work order by a human production manager. The work order defines the quantity of parts required for a given part type.

4.2. Data Flow Analysis

Figures 6 and 7 show the data flow diagrams for the Factory Automation system. In Figure 6, the Process Planning transform accepts process engineer inputs and creates process plans.

There is a Co-ordinate Workstation transform for each workstation. The first and last workstations are unique. Figure 7 shows the Co-ordinate Workstation transform of Figure 6 decomposed into Co-ordinate Receiving Workstation, Co-ordinate Line Workstation (of which there are several connected in series, one per workstation, but only one is shown in figure 7) and Co-ordinate Shipping Workstation.

In Production Management, a work order is created to meet a customer need. When a work order is released to the factory by the production manager, a start part message identifying the part id and number of parts required is sent to Co-ordinate Receiving Workstation. Co-ordinate Receiving Workstation ensures that for each new part, a piece of raw material of the appropriate type is obtained and loaded onto the conveyor.

A just-in-time algorithm is used, which means that a workstation only requests a part when it needs it. When a workstation completes a part, it waits for a message from its successor workstation requesting the part. Only then will the part be sent. Following this, the workstation also sends a Part Coming message to the successor workstation and a part request message to the predecessor workstation. The receiving workstation maintains a count of the remaining number of parts for a given work order. At the shipping workstation, finished parts are removed from the conveyor and either shipped or placed in a finished goods inventory.

During part processing, the Workstation Status Data Store is updated. If an alarm condition is detected, an alarm is sent to Alarm Handling (Figure 6).

Other functions provided by the system are shown on Figure 6:

(1) Status Monitoring. An operator or supervisor may view current or historical status of one or more workstations, either in tabular or graphical form. The screens are dynamically updated to show changes in status. Current data is read from the workstation status data store, while historical data is read from the history store.

(2) A Statistical Quality Control function is provided to allow predefined algorithms to be executed as requested by the operator, e.g to compute mean and standard deviation of some process variable. Once again, either current and/or historical data may be used.

(3) Alarms are stored in the alarms data store by Alarm Handling. For each alarm generated, the list of operators to be notified of this alarm is scanned and the alarm is sent to them. Operators may view and acknowledge alarms.

(4) Selected historical data is collected and stored in a data base by Historical Analysis. This data may be used later by other transforms.

4.3. Structuring Application into Subsystems

The data flow diagrams in Figures 6 and 7 are analyzed to determine the subsystems, which are shown in Figure 8. In Figure 6, each transform corresponds to a subsystem. Each of the three classes of workstation (Figure 7) is also a subsystem. Based on the subsystem structuring criteria given in Section 2.2, the subystems are all functional subsystems. However some of the subsystems also demonstrate other subsystem structuring criteria. Process Planning, Alarm Handling and Historical Analysis are also server subsystems. They receive service requests from other subsystems and must respond to them. Statistical Quality Control and Status Monitoring are agent subsystems. They receive requests from the Operator Interface subsystem and must then make service requests to Workstation Status Server and/or Historical Analysis to obtain the necessary data. Co-ordinate Workstation is close to the source of physical data, exhibits localized control and performs a time-critical function. Operator Interface is a user interface subsystem.

Next, consider subsystem interfaces. As pointed out in Section 2.3, only message interfaces are allowed between subsystems. Data Stores must be local to a subsystem. Shared data stores must either be encapsulated within a central server or a distributed server. Figure 6 shows that the shared data stores are the Process Plans, Operations, Workstation Status, Alarms and History data stores. The Process Plans and Operations data stores are encapsulated within the Process Planning Server subsystem, as shown in Figure 8. Thus Production Management and Co-ordinate Workstation subsystems must now send message requests to the Process Planning subsystem to get process planning data or operations data. The alarms and history stores are encapsulated within the Alarm Handling and Historical Analysis subsystems respectively.

There is a Workstation Status Data Store for each workstation. In addition its users, e.g. Status Monitoring, may need to read from several of these concurrently. For example a factory status display needs status information from each workstation. Consequently a Workstation Status Distributed Server is used. Statistical Quality Control, Status Monitoring and Historical Analysis make requests for workstation status by sending messages to the Workstation Status Server task, which responds with workstation data. The data maintained by this server is updated via messages sent by Co-ordinate Workstation.

4.4. Subsystem Design

To design each subsystem, we apply the DARTS task structuring criteria. Two subsystems are considered. Co-ordinate Workstation is an example of a real-time subsystem, Alarm Handling is an example of a server subsystem. Other examples of using DARTS are given in [1,17].

4.4.1. Co-ordinate Workstation

As the Co-ordinate Workstation subsystem is a real-time subsystem, Real Time Structured Analysis is used. The data flow diagram for Co-ordinate Line Workstation is shown in Figure 9, while the state transition diagram for the control transformation, Control Workstation, is shown in Figure 10. In Figure 10, using the Real Time Structured Analysis conventions [7], states are represented by rectangles while labeled arrows represent state transitions. Above the horizontal line is the condition which causes the state transition. Below the line is the action which is performed when the transition occurs. The action is in the form of an event which triggers, enables or disables a transform on Figure 9.

Figure 10 shows that at workstation startup, the workstation transitions into the Awaiting Part from Predecessor Workstation (PWS). An action is also performed, namely the Request Next Part event is signaled which triggers the Send Request Next Part Message transform. When a Part left from PWS message is received, Receive Part Coming message reads the operation, which defines the PLC parameters and robot program id, from the operations data store and stores current part and operation data in the part/operation data store. It then signals the Part Coming event flow to Control Workstation. This causes the workstation to transition to Part Arriving state. A sensor attached to the PLC detects that the part has physically arrived at the workstation. This PLC status is sent to Monitor PLC which signals the Part Arrived event flow. The workstation transitions to Robot Picking state and triggers the Pick Part transform. Pick Part sends the Robot Pick command to the pick-and-place robot controller which removes the part from the conveyor and moves it into the workstation. On completion, it signals Part Ready. Control Workstation transitions to Part Processing and triggers the Send Operation to PLC transform.

During part processing, the PLC synchronizes the other automation equipment. Status information is passed up to Monitor PLC which updates the Workstation Status Data Store. Monitor PLC also signals Operation End when part processing has been completed. Control Workstation then waits for the Request Part signal from its successor workstation (SWS). If the signal has already been received, it transitions immediately to Robot Placing. On entering this state, it requests the robot to place the part onto the conveyor. When the part has departed, it sends the part left message to its SWS, and then sends a request next part message to its PWS.

We now apply the DARTS task structuring criteria. By definition, executing the state transition diagram is a purely sequential activity. The transforms which are associated with executing the state transition diagram are grouped into a task, Co-ordinate Part Processing, according to the sequential cohesion task structuring criterion. These transforms are 3.2.1, 3.2.2, 3.2.3, 3.2.7 and 3.2.9. The transforms Pick Part and Place Part interface directly with the robot controller and so are combined into an I/O dependent Robot Interface task. Similarly the Send Op to PLC transform and the Monitor PLC transform both interact directly with the PLC and so are structured according to the I/O task structuring criterion into the PLC Interface task. Both the Robot and PLC Interface tasks must be active concurrently with Co-ordinate Part Processing as they are continually monitoring the external environment and generating alarms if necessary. The task structure chart for this subsystem is shown in Figure 11.

Figure 11 shows that messages are used for communicating with tasks in other subsystems, namely predecessor and successor workstations as well as the Workstation Status Server and Process Planning Server. The Part/Operation data store is encapsulated in Co-ordinate Part Processing. Requests to PLC and Robot Interface include data from this store and are therefore also sent as messages. The Part Arrived and Operation End event flows are designed as event signals since they carry no data.

4.4.2. Alarm Handling

The Alarm Handling subsystem is a server subsystem. The three services provided are Read Alarm List, Add New Alarm and Acknowledge Alarm. The former is a read only service while the other two are write services. Alarm Handling is designed as a concurrent server subsystem (Figure 12) to provide improved throughput by supporting multiple reads. Each of the three services is implemented as a task. Receive Alarm Requests will create a new instance of Read Alarm List and send it a new request as long as no write request has been received. If one has, then it waits until all active read requests have been completed (and queues up any new read requests) before sending the write request to the writer task.

5. Conclusions

This paper has described a design method called DARTS/DA, the design approach for distributed applications. The method addresses the needs of distributed applications by providing criteria for structuring the application into distributed subsystems as well as criteria for structuring subsystems into concurrent tasks. The method leads to a design which is primarily message based. The use of DARTS/DA has been illustrated with the design of a distributed Factory Automation System.

6. References

(1) H. Gomaa, "A Software Design Method for Real Time Systems", Communications ACM, September, 1984.

(2) H. Gomaa, "Software Development of Real Time Systems", Communications ACM, July, 1986.

(3) T. DeMarco, "Structured Analysis and System Specification", Yourdon Press, 1978.

(4) C. Gane and T. Sarson, "Structured Systems Analysis", Prentice-Hall, 1979.

(5) E. Yourdon and L. Constantine, "Structured Design", 2nd. Edition, Yourdon Press, 1978.

(6) M. Page-Jones, "The Practical Guide to Structured Systems Design", Yourdon Press, 1980.

(7) P. Ward and S. Mellor, "Structured Development for Real-Time Systems", Vols. 1 & 2, Yourdon Press, 1985.

(8) P. Ward, "The Transformation Schema: An Extension of the Data Flow Diagram to Represent Control and Timing", IEEE Transactions of Software Engineering, Vol. SE-12, No. 2, February 1986.

(9) D.L. Parnas, "On the Criteria to be Used In Decomposing Systems into Modules", Communications of the ACM, December 1972.

(10) D.L. Parnas, "Designing Software for Ease of Extension and Contraction", IEEE Transactions on Software Engineering, March 1979.

(11) W.M. Gentleman, "Message Passing Between Sequential Processes", Software - Practice and Experience, Vol. 11, 1981

(12) J. Martin, "Design and Strategy for Distributed Systems", Prentice-Hall, 1981.

(13) G. Booch, "Object Oriented Development", IEEE Transactions on Software Engineering, February 1986.

(14) L. Svoboda, "File Servers for Network-Based Distributed Systems", ACM Computing Surveys, Vol.16, No.4, 1984.

(15) A. Tannenbaum and R. Van Renessee, "Distributed Operating Systems", ACM Computing Surveys, Vol.17, No.4, 1985.

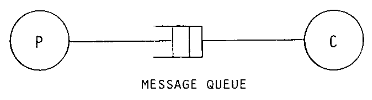
(16) D. Harel, "Statecharts: A Visual Approach to Complex Systems," Report No. CS86-02, Dept. of Applied Math.,Weizmann Institute, March 1986.

(17) H. Gomaa, "Using the DARTS Software Design Method for Real Time Systems", Proc. Twelfth Structured Methods Conference, Chicago, August 1987.

FIGURE 1. MESSAGE COMMUNICATION

- LOOSELY COUPLED MESSAGE COMMUNICATION

PRODUCER P:	CONSUMER C:
SEND MESSAGE (C,M)	RECEIVE MESSAGE (P,M)



- TIGHTLY COUPLED MESSAGE COMMUNICATION

PRODUCER P:	CONSUMER C:
SEND MESSAGE (C,M)	RECEIVE MESSAGE (P,M)
WAIT REPLY (C,R)	SEND REPLY (P,R)

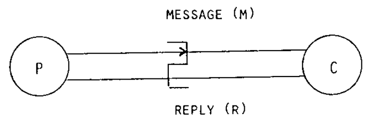


FIGURE 2. INFORMATION HIDING MODULE

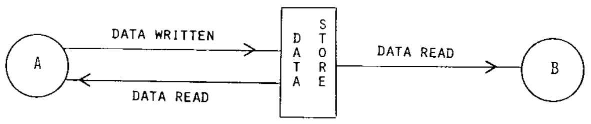
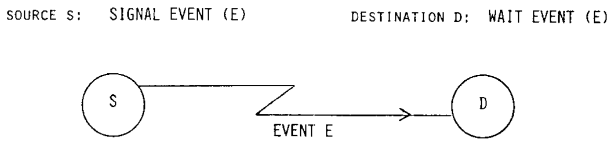


FIGURE 3. TASK SYNCHRONIZATION



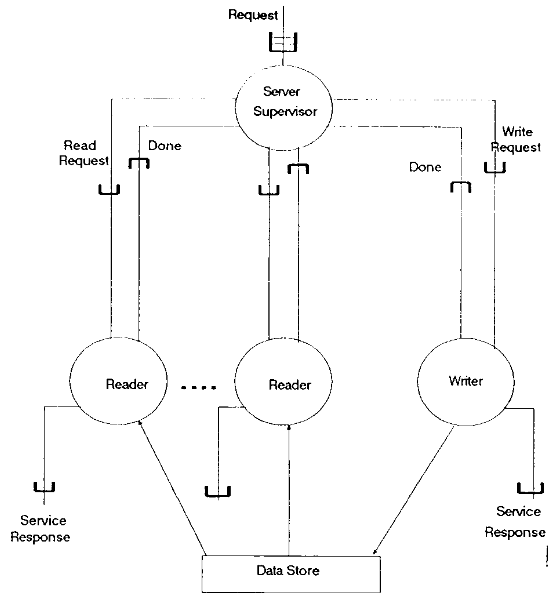
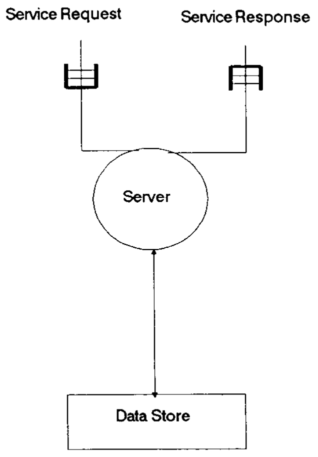


Figure 4b: Multiple Task Server

Figure 4a. Single Task Server



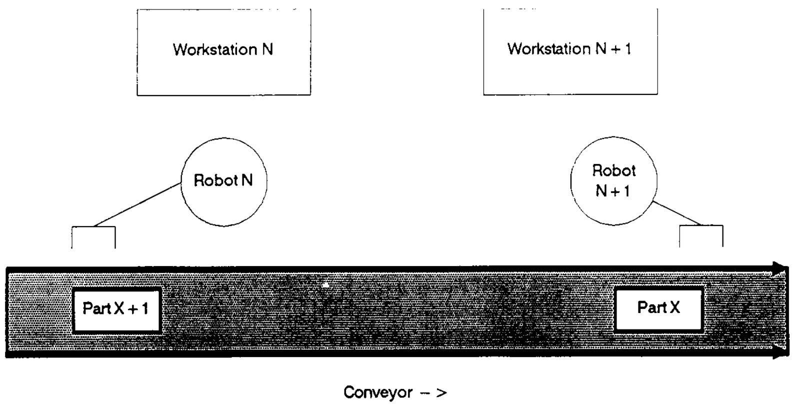


Figure 5: Factory Automation System

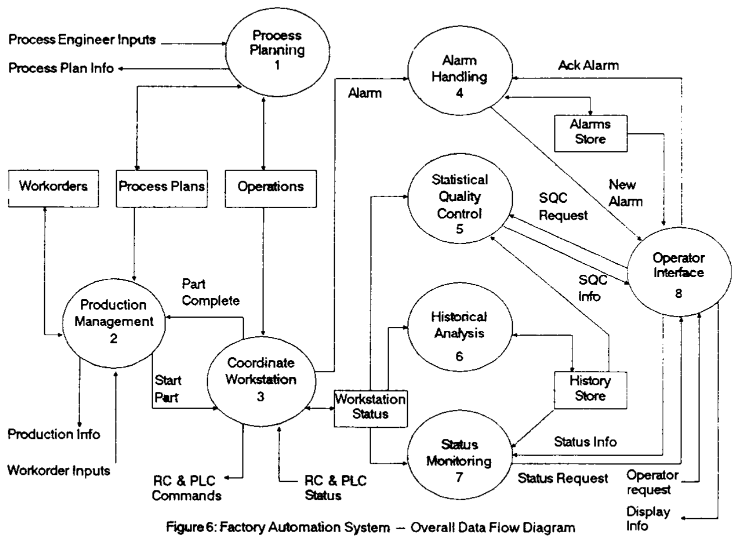


Figure 6: Factory Automation System – Overall Data Flow Diagram

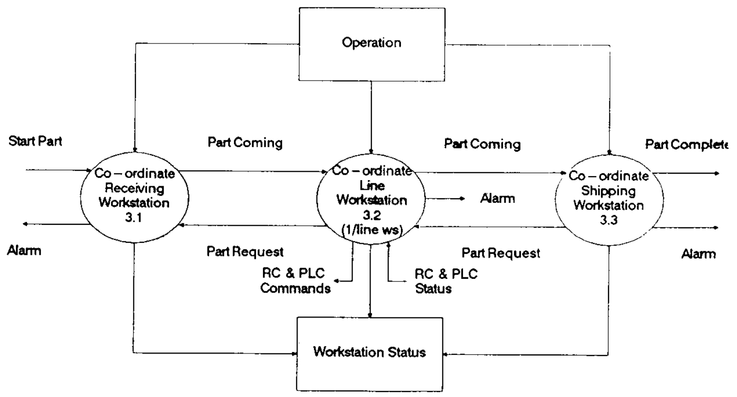


Figure 7: Co – ordinate Workstation Data Flow Diagram

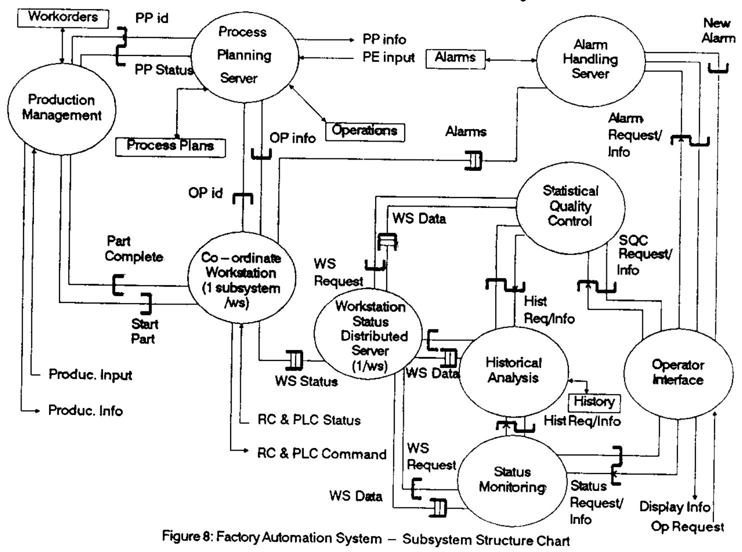


Figure 8: Factory Automation System – Subsystem Structure Chart

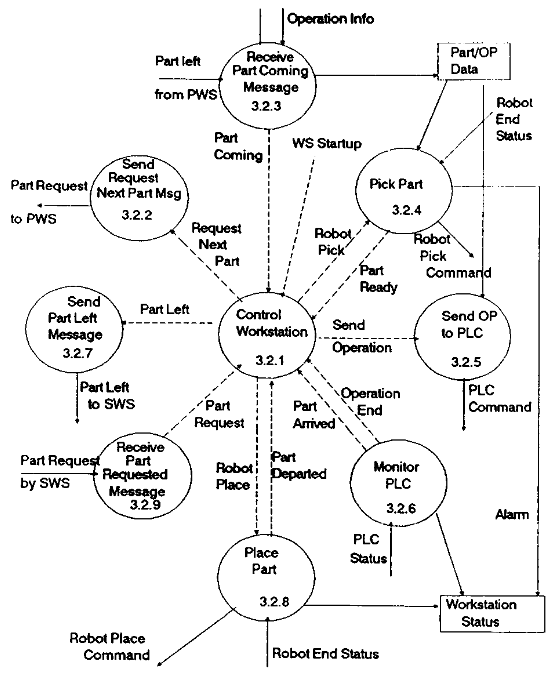
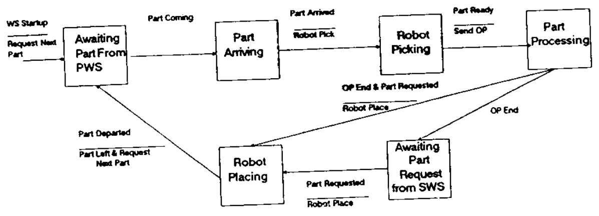


Figure 9: Co – Ordinate Workstation Data Flow Diagram

Figure 10: Control Workstation – State Transition Diagram



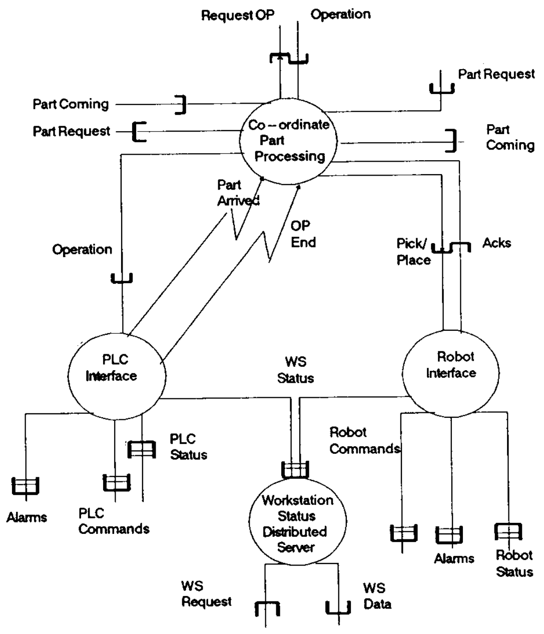


Figure 11: Line Workstation Controller: Task Structure Chart

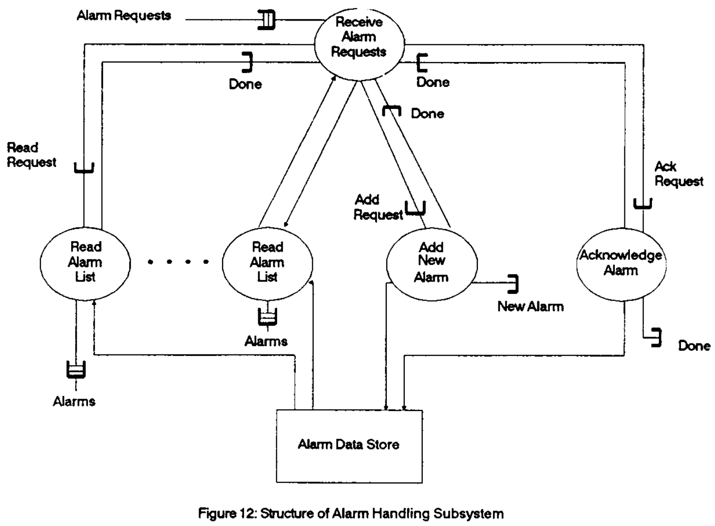


Figure 12: Structure of Alarm Handling Subsystem