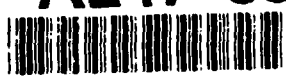


Fault-Tolerant Parallel Computer Systems for Real-Time Applications

Final Report
Contract No. N00014-88-K-0622
Office of Naval Research

S.M. Yang and Bill D. Carroll
Computer Science Engineering Department
The University of Texas at Arlington
Arlington, TX 76019

February 1992

Table of Contents

CHAPTER I

INTRODUCTION

Statement A per telecom
Dr. Keith Bromley
ONR/Code 126A-2
Arlington, VA 22217-5000
NWW 3/24/92

This is the final report that summarizes the research results obtained under the contract N00014-88-K-0622. The objective of our research was to investigate techniques for designing fault-tolerant parallel computer systems for critical real-time applications. The focus of our research was to develop the practical fault tolerance design, implementation and analysis technology with the considerations of real-time recovery, structuring of recoverable interactions, and handling of software as well as hardware failure in distributed/parallel computing environments. We also investigate techniques for scheduling of real-time messages as welll as real-time tasks in fault-tolerant distributed systems.

The specific tasks carried out are as follows:

(1) A Parallel computer architecture and software structures for ultra reliable, real-time applications are proposed. The proposed architecture is a modified hypercube called the Hybrid Higher Radix Hypercube (HHRH). Processor nodes are partitioned into clusters. Nodes in a common cluster communicate through shared memory but communicate with nodes in other clusters by point-to-point connections. A possible implementation of the architecture along with the system components are investigated. The conversation scheme is adopted as the basic fault tolerance scheme to be used with the HHRH architecture. Along with the conversation scheme three difficult concepts of error detection and recovery schemes, namely, filtration, consensus and approval, are incorporated at the various stages of computation and/or communication.

(2) Several different approaches for implementing conversations in message-based distributed computer systems are investigated. Important implementation factors to be considered include the control of exits of processes upon completion of their conversation tasks and the approach to execution of the conversation acceptance test. As a case study, an unmanned vehicle system is used to illustrate how the identified approaches to implementation of the conversation scheme can be used in a realistic real-time application.

(3) Queueing network models are developed for both the system operating under the basic conversation execution scheme and the system operating under the execution scheme extended with the look-ahead capability. Based on the models, various performance indicators such as the system throughput, the average number of the processors idling inside a conversation due to the synchronization required, and the average time spent in a conversation, are evaluated numerically for different application environments. The performance under the lookahead scheme are compared against these under the basic conversation execution scheme.

(4) An execution time estimation techniques is proposed for efficient and safe task scheduling of critical embedded real-time applications. Execution time estimation is one of the most important issues in real-time systems since etermination of task schedulability depends primarily on the exact knowledge of the execution time behavior of a task. However, input-dependent branches or loops and intertask communications make the program's timing behavior hard to predict. We introduces a Dynamic Execution Time (DET) estimation technique for determining the worst-case execution time behavior of a task. DET estimation comprise two concepts: (1) compile time estimation based on semantic as well as syntactic analysis and (2) runtime estimation based on Execution Time functions. We demonstrate the usefulness of DET estimation for real-time task scheduling, especially under time-value-function based (or best effort) scheduling strategy.

(5) A protocol, named token passing with concession (TP/C), is proposed for real-time messages and communication channel scheduling in distributed computer systems. The protocol can be implemented on top of the existing contention-free protocols such as token bus, token ring and FDDI ring protocol. The protocol is useful for real-time control applications in distributed computing environments as well as for voice and data communication in LAN/MAN environments. The performance matrics such as percentage of message lost and effective channel utilization were obtained by simulation under various network and protocol parameters and compared with CSMA/CD, Virtual Time CSMA, window and token bus protocols.

(6) A distributed real-time system testbed to support experimental research has been established. The testbed, named SUVS (Simplified Unmanned Vehicle System), is being used to conduct experimental evaluation of system level fault tolerance techniques in distributed/parallel real-time systems. Current version of SUVS is written in C and is running on a network of eight SUN Workstations. The SUVS testbed is planned for clinical study of specification, design and implementation methods studied by the investigators for real-time distributed/parallel systems.

The following 6 chapters, Chapter II to Chapter VII, describe the details of the tasks discussed above. Attempts were made to organize the chapters to be self-contained. Chapter VIII is the summary. We also propose a new system development model for large complex real-time embedded systems along with the current status of our work.

CHAPTER II

A PARALLEL ARCHITECTURE AND PROCESS CONFIGURATION WITH CONVERSATIONS FOR ULTRA RELIABLE REAL-TIME COMPUTING

1. INTRODUCTION

The use of computers as embedded components in critical real-time systems has led to a need for ultra reliable, fault-tolerant, real-time computers. For the past two decades various fault tolerance mechanisms have been proposed and implemeted at different levels, e.g., hardware level, operating system level, application software level, etc., within computer systems [Ran78,Ren80,Ser84,Kim82]. Although many systems containing different sets of mechanisms are now available, they do not satisfy the requirements for reliable operations of many critical real-time applications. Furthermore, many current fault tolerance mechanisms cannot be implemented for practical use. This is partly because of high implementation cost due to redundancy required in software as well as in hardware and partly because of lack of practical design technologies to implement the mechanisms in a parallel-processing, real-time computing environment [Kim85].

Recently, the development of newer, faster, and cheaper microprocessors has led to their incorporation into a variety of parallel processing systems [Aga86,Bon87]. The primary reason for adopting parallel processor systems is their computing power, especially for problems with inherent parallelism either in data or in processing operations. Another attractive reason is the redundancy which is essential for fault-tolerant operation of the computer system. That is, in a parallel processor organization, a failure in one unit need not interfere with the continuing operation of other parallel units. Hence, a parallel organization can increase the system reliability, particularly if software redundancy is also employed.

In designing fault tolerant capabilities into real-time computer systems in distributed/parallel environments, the following characteristics must be carefully considered. First, processing nodes exchange information among themselves. This interaction may cause fault propagation through the network. That is, faulty behavior of one node may cause the failure of other nodes. Therefore, cooperation among nodes for fault detection and recovery is needed. Second, timely recovery (or real-time recovery) is an important aspect to be considered in designing a fault-tolerant real-time system. The system should produce correct results within a specified time limit in the presence of failure in hardware and/or software. Finally, the network structure of a system has a significant effect upon the design of the operating system as well as the fault tolerance mechanisms incorporated in it. It also affects the system performance in terms of communication overhead, recovery time, and reconfiguration

after failure.

In this chapter a modified hypercube architecture, called the Hybrid Higher Radix Hypercube (HHRH), is proposed for use in reliable, real-time environments. Processor nodes are partitioned into clusters. Nodes in a common cluster communicate through shared memory but communicate with nodes in other clusters by point-to-point connections. This approach provides (1) short diameter and average node distance which makes communication between nodes fast and predictable (speed and predictability are the most important factors in real-time systems), (2) redundancy which is essential for fault-tolerant computing, and (3) reconfigurability as required by many applications.

The conversation scheme has been adopted as the basic fault tolerance mechanism to be used with the HHRH structure. This choice was made because (1) the scheme deals with recovery actions of interacting processes, (2) software redundancy is provided, (3) real-time recovery is achievable, and (4) the scheme is relatively easy to implement. Various process configuration strategies with conversations are illustrated and compared using a real-time control system scenario, called the Simplified Unmanned Vehicle System (SUVS). SUVS consists of a set of real-time interacting processes. Software redundancy is thus incorporated in two forms in SUVS: (1) multiple versions in the form of alternate try blocks in conversations and (2) multiple identical copies of processes. Three different concepts of error detection and recovery schemes, namely filtration, consensus, and approval are incorporated along with the conversation scheme. Filtration is implemented in the form of voting among multiple (identical) copies of a process, consensus in the form of comparison among multiple versions of a process, and approval in the form of conversation acceptance test. These are not exclusive but are incorporated into the system at various stages.

In the next section the HHRH architecture is described and its characteristics are discussed. A possible implementation of the architecture with the description of the system components is also discussed. Section 3 presents a real-time control system scenario, the SUVS, which is used to illustrate the software structuring strategies. The conversation scheme along with other error detection and recovery schemes are incorporated in the SUVS. In Section 4 three different process configuration and mapping strategies to the HHRH architecture are proposed and compared. Section 5 presents the preliminary result on the reliability and the communication overhead of various fault-tolerant configurations. Section 5

is the summary section.

2. THE HYBRID HIGHER RADIX HYPERCUBE (HHRH) ARCHITECTURE

In this section the basic characteristics of the binary hypercube (or simply hypercube in this chapter) architecture and the higher radix hypercube architecture are reviewed. Then the Hybrid Higher Radix Hypercube (HHRH) architecture is proposed. The architecture and a message routing scheme are described and the performance of the HHRH architecture is analyzed. A possible implementation of the architecture with a description of the system components is also given.

2.1 The Higher Radix Hypercube Architecture

2.1.1 Binary vs. Higher Radix Hypercube. The Hyperube structure can be visualized as a cube of any dimension with a node at each corner. Each node typically has its own processing unit, local memory, a communication processor, an optional floating point processor, a kernel of the operating system and the application program. At the lowest level of the hypercube family is the binary hypercube, for which the relation:

$N = 2^n$ where N = number of nodes, and
n = dimension of the hypercube.

The success of the hypercube in parallel systems is a result of a number of suitable features, some of which are unique to it [Sei84]: the ease of expansion by increasing the dimensionality, the absence of memory contention, the flexibility of its interconnection scheme allows embedding of other topologies, and the simplicity of routing messages. Moreover, the fault tolerance capability in terms of the number of disjoint paths between any two nodes increases as does the ratio of the communication to computation capability, with increase in the network size.

As the proposed hybrid architecture is partially obtained by increasing the radix beyond 2, examined below is the resulting topology and the benefits obtained if any, as a result of radix enhancement. Figure 1 shows the higher radix hypercube (HRH) [Bhu84] for N = 27, r = 3. For this structure the following relation is true:

$N = r^n$ or $Log_r N = n$ where N = number of nodes,
n = demension of the hypercube, and
r = radix

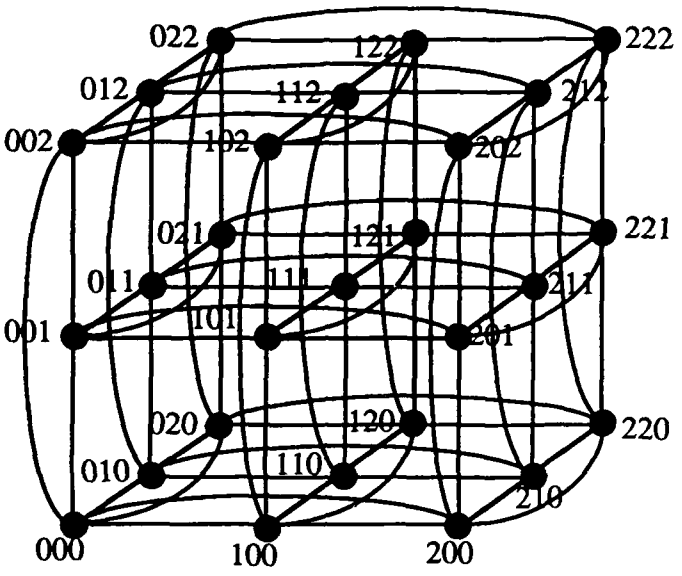


Figure 1. The HRH with N=27 and r=3.

The interconnection scheme between the nodes is as follows: Given nodes

$$X = X_{n-1}X_{n-2}.....X_i.....X_1X_0$$

and

$$Y = Y_{n-1}Y_{n-2}.....Y_i.....Y_1Y_0$$

then node X connects to node Y iff there exists a k such that

$$X_i <> Y_i \text{ for } i = k \text{ and } X_i = Y_i \text{ for all other } i <> k$$

where X_i, Y_i $\in$ {0,1,2,...(r-1)} and i, k $\in$ {0,1,2,...(n-1)}.

2.1.2 Effect of the Radix on the Hypercube Performance. The radix of the hypercube affects the parallel system characteristic parameters, such as diameter, degree of node, average node distance, and message traffic density.

(1) Diameter: The diameter, which is defined to be the maximum number of links that must be traversed between any two nodes in the network provided that the shortest possible route is chosen, can be shown to be:

$$\text{Diameter} = \log_r N$$

It should be noted that for a binary hypercube, the radix, r, is equal to two. It follows from the above equation that by increasing the radix reduces the diameter for the same sized network. This translates into a vital reduction in the maximum value for the message latency that is brought about by reduced store-and-forward operations.

(2) Degree of node: The degree of a node is given by the following relation:

$$\text{Degree} = n*(r - 1)$$

As current technology supports more than an order of magnitude higher than the degree of nodes in highly parallel systems, the increase with the radix does not pose a problem.

(3) Average Node Distance (AND): This parameter [Aga86] is given by the relation:

$$\text{AND} = \{d * (r-1)^d * C(^n_d)\}/ (N - 1)$$

From the above equation, it can be shown that for the same sized network, an increase in the radix reduces the average node distance.

(4) Message Traffic Density (MTD): By proper substitution in the definition, the MTD [Aga86] for an HRH can be shown to be given by the following relation:

$$\text{MTD} = 2 * (\text{AND})/ \{n*(r - 1)\}$$

The above relation confirms the decrease in the traffic density obtained by increasing the radix. Considering the fault tolerance of the HRH, it is found that the number of alternate disjoint paths is n*(r -1). Hence, more node and link failures can be tolerated in a network with a higher radix.

2.2 The Hybrid Higher Radix Hypercube (HHRH) Architecture

The major advantage of using the hypercube structure for designing fault-tolerant real-time systems is the inherent redundancy provided in processors and communication links. As the number of nodes increases, however, the distance between nodes (i.e., the number of intermediate nodes between two communicating nodes) becomes longer. This long distance not only increases the message traffic among the nodes but, more importantly, makes the accurate prediction of the system behavior and performance difficult. The HRH architecture remedies these weaknesses to some degree, there is yet a limitation.

The shared memory with bus architecture, on the other hand, seems more predictable in its behavior and performance. However, it suffers from two problems: (1) the bus is a single failure point, and (2) the bus may become a communication bottleneck of the system. Therefore, a shared memory architecture is not suitable for some real-time applications which requires ultra reliable computing and/or high communication traffic among nodes.

The Hybrid Higher Radix Hypercube (HHRH) architecture remedies these problems by combining the hypercube structure and shared memory units. Processor nodes are partitioned into clusters. Nodes in a common cluster communicate through shared memory but communicate with nodes in other clusters by point-to-point connection. The shared memory units not only provide high bandwidth to nodes within clusters but reduce the distance between nodes in different clusters. Characteristics of the HHRH architecture are presented in the following subsections. The discussion includes implementation of message routing and performance analysis of the HHRH architecture.

2.2.1 The HHRH Architecture. Figure 2 illustrates a simplified diagram for the hybrid network obtained by collapsing the least significant bits of an HRH with N =16, r = 4. The components of Figure 2 are the nodes (00, 01, ... 32, 33), the cluster memory units (CM0, CM1, CM2, CM3), and the links between the nodes and memory units. Nodes that are directly connected to the same cluster memory unit are grouped as PC0, PC1, PC2 or PC3. The overall topology between the processor clusters is that of an HRH, while the grouping of the processor nodes of the HRH architecture into clusters makes it hybrid. The grouping is accomplished by collapsing relevant bit positions. The nodes that have identical non-collapsed

bits, irrespective of the collapsed bit values, belong to the same processor cluster and share the same cluster memory. In Figure 2, for example, nodes 00, 01, 02, and 03 belong to the processor cluster PC0 and share the cluster memory CM0. Hence the collapsed bit positions, i.e., the least significant digit in the case, can be thought of as "don't care" positions.

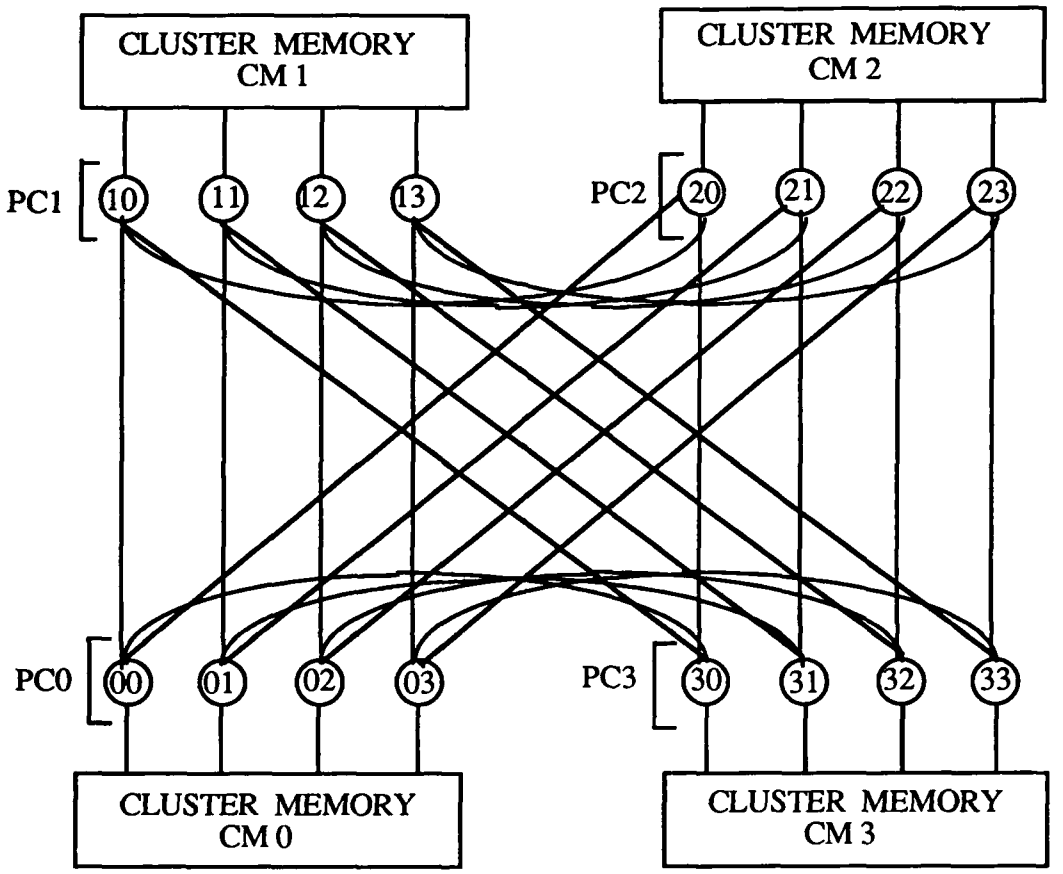


Figure 2. The HHRH with N=16 and r=4.

The hybrid system in addition to the underlying network includes the processor node, cluster memory and the system manager.

(1) Processor Node: As mentioned earlier, the node at each corner of the cube consists of the main CPU, a communication processor, a floating point processor and local shared memory. An internal bus interconnects these devices. The communication processor handles all message requests that require routing to other nodes as well as the cluster memory. The I/O channels

are connected via links to neighboring nodes in other processor cluster groups, and one channel connects to the memory unit shared by all processor nodes in its cluster. Each node also has storage for message requests in a buffer that is accessed on a FIFO basis by the communication processor. Figure 3 is a typical node processor layout.

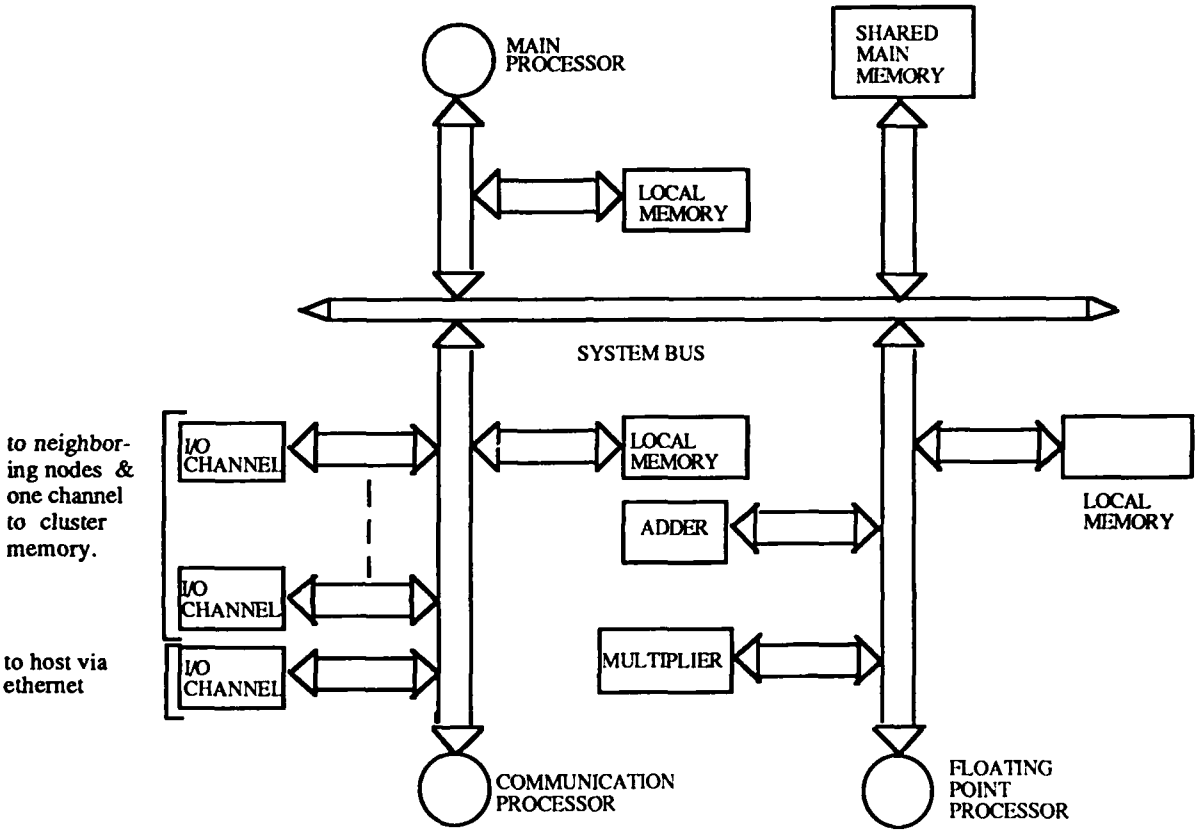


Figure 3. Node Processor Layout.

(2) Cluster Memory: The cluster memory is a high speed memory unit that consists of 2 sub-devices: the message handler and the memory mailbox. Figure 4 is a block diagram of the cluster memory shared by p node processors. The memory mailbox is a passive device that stores messages at appropriate locations. These predesignated location addresses are stored in a lookup table in the message handler. As each node has with it (p-1) slots to post messages to all possible nodes shared by the memory unit, a total of p*(p-1) slots exist. The message handler is a dedicated unit for handling message requests for the cluster memory. Its hardware

includes a processing unit, a message request buffer and I/O channels that connect to the member nodes. Figure 5 shows the main components in the message handler unit. An additional lookup table provides information on the read/write status for the message slots. The message request buffer is accessed by the processing unit on a FIFO basis. The detailed execution of the message handling is discussed in a later section.

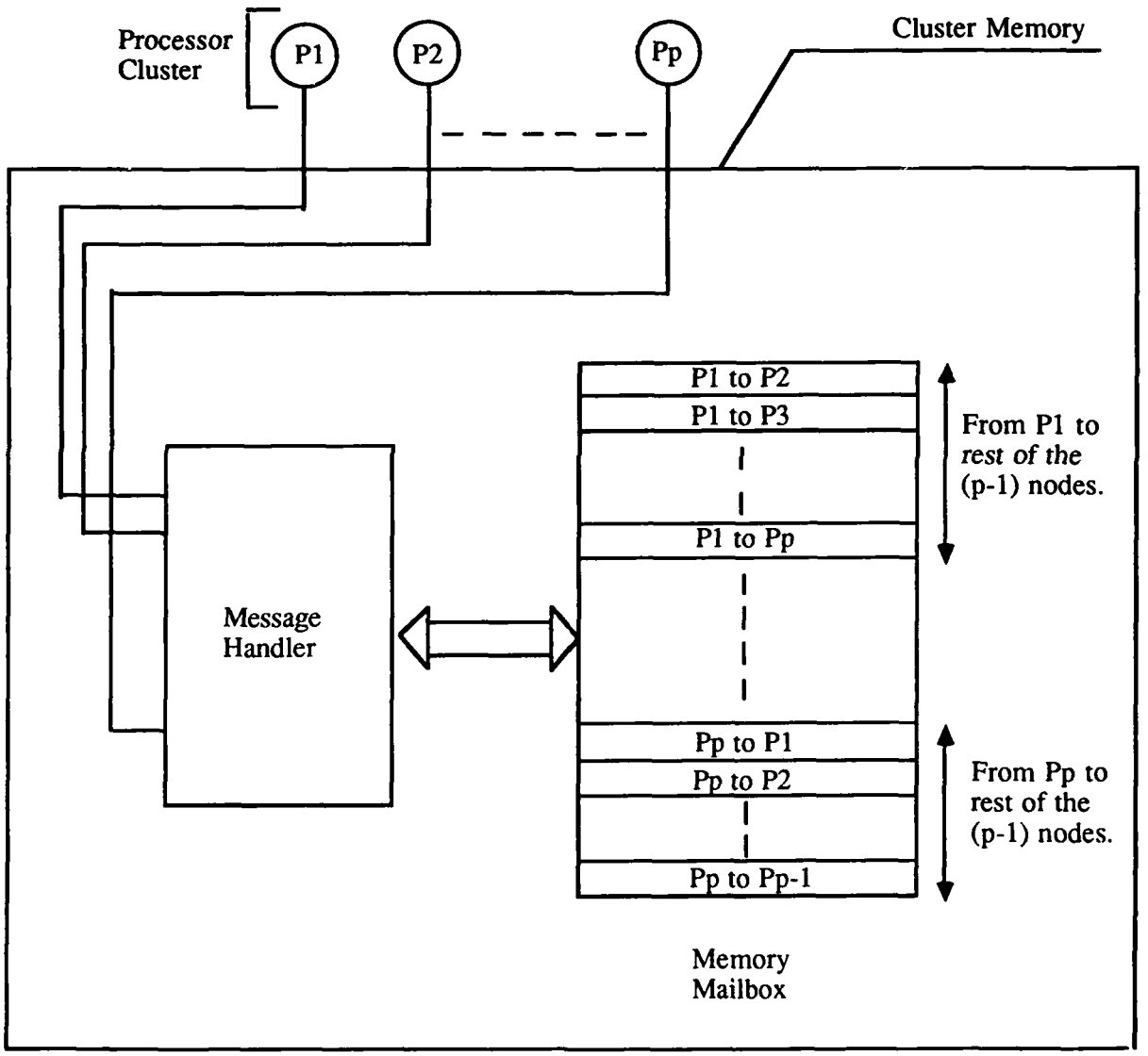


Figure 4. Cluster Memory Layout.

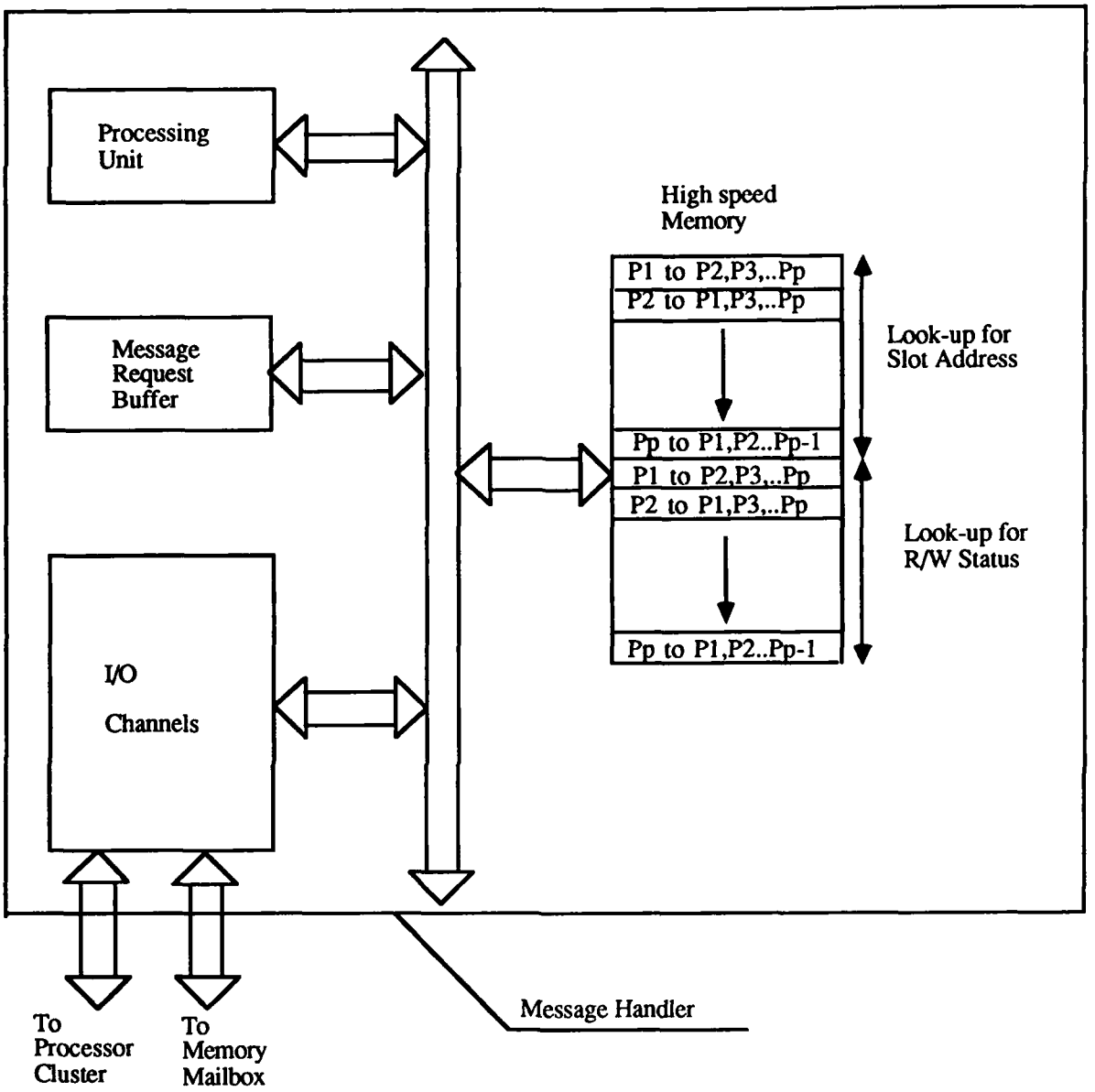


Figure 5. Memory Handler Layout.

(3) System Manager: This device is connected to the nodes in the network via a global communication channel such as the Ethernet. The system manager has varied functions. It serves as the administrative console and also acts as the gateway to the hypercube for the users. It supports a program development system that includes compilers, simulators and vector tools that users can access. In addition, the system manager is able to download data/instructions to

the processors in the network in a short span of time.

2.2.2 Implementation of Message Routing. Prior to an in-depth discussion, it is important to consider the assumptions made with respect to the routing procedure. It is assumed that the mode of communication is packet switched and is further assumed that each message packet has a message header. The message header has fields to hold attributes such as source and destination addresses, source and destination process id numbers, message length and type etc.

The basic primitives employed are send and receive, and in addition some variation of these. As the routing hardware is different for the links and the cluster memory, the implementation also differs and is hence presented separately. The routing of messages over the links is handled by the communication processor. It accesses the message request buffer on a FIFO basis, and for each request obtains routing information from the header. It should be noted that the necessity of global communication (via communication links) arises only when the non collapsed bit positions of the source and destination do not match. The communication processor at each intermediate node determines at least one non-matching bit position and to reduce the disparity, routes the message via the appropriate channel to an address that resembles the destination. Thus the same message is stored and forwarded at several processor nodes. This sequence of events continues until all non-collapsed bits are matched. Following this, the matchability of the collapsed bits is determined. If all these bits match then the destination is the current node. Otherwise, a local transfer via cluster memory will be necessary to complete the routing procedure. Message routing via cluster memory may be initiated due to any of the following: (a) a node in the cluster has to send a message to another node belonging to the same cluster, and (b) a node in the cluster, which is an intermediary node in the complete route path has to forward a message to the destination node.

Consider the case when a node wishes to send a message via the cluster memory. The communication processor at the sender node initiates handshaking with the message handler unit in the cluster memory and upon success, the message is buffered in the request buffer. The processing unit in the handler, like the communication processor at each processor node accesses the buffer on a FIFO basis. Processing the request involves determining the address slot for the source destination pair, and obtaining the Read/Write status for that slot. Depending on the message type, the Read/Write status is interpreted to enable or postpone message transfer. It should be noted that for each send request, a corresponding receive

primitive must be inserted in the source code. It is important to also note that to prevent queueing at the slots, a node does not initiate a second message to the same destination unless the acknowledgement for the previous message is received.

2.2.3 Performance Analysis of the HHRH. The analysis of the HHRH necessitates the redefining of certain terms, as the introduction of memory units must also be taken into account. In the network analysis the following is obtained [Bha88].

(1) Hybrid Diameter: This parameter includes traversals via the memory units, while the rest of the definition is identical to that of the diameter. For an HHRH, it is given as follows :

$$\text{Hybrid diameter} = (\log_r N) - x + 1 \text{ memory traversal}$$

where x = number of collapsed bits

The hybrid diameter is further reduced as the size of the processor cluster is increased. There will however be a performance as well as a physical restriction on this size.

(2) Degree of node: The presence of shared memory units that interconnect all nodes in a particular cluster eliminates the need for any direct connection among them. This reduction increases with the cluster size and hence the number of collapsed bits, and is given as:

$$\text{Degree} = n*(r-1) - (r-1)^x + 1$$

(3) Hybrid Average Node Distance (HAND): This parameter is also modified to accommodate memory traversals as was the case for the diameter. Equating a memory traversal to a link traversal, then for an HHRH the following is derived:

$$\begin{aligned}\text{HAND} = [&1*\{(r^x-1) + C(^{n-x}1)*(r-1)^1\} + \\ &2*\{(r^x-1) + C(^{n-x}2)*(r-1)^1 + C(^{n-x}2)*(r-1)^2\} + \\ &\ldots. \\ &(n-x)*\{ \ldots\ldots\ldots\ldots\ldots\ldots\ldots\} + \\ &(n-x+1)*\{(r^x-1) + C(^{n-x}_{n-x})*(r-1)^{n-x}\}]/(N-1)\end{aligned}$$

Upon substitution in the above equation, it is found that for x > 1, the average node distance

for the HHRH is lower than that for an HRH with the same radix, and even greater difference is found when a comparison is made to the binary hypercube.

(4) Hybrid Message Traffic Density (HMTD): The total number of paths in this definition of the message density accommodates paths via links as well as cluster memory, and for the hybrid HRH it is found to be:

$$HMTD = \{2*HAND\}/\{(r-1)*(n-x)+(r^x - 1)\}$$

From the equation above, it is obvious that the traffic density is improved (i.e., lessened). If the higher bandwidth of the memory were accounted for, the hybrid hypercube traffic density would further reduce.

The fault tolerance capability, in terms of the number of disjoint paths, is found to be n*(r-1). This is the same as that obtained for a regular HRH, but is an improvement over the binary hypercube. It should be noted that as the reliability of the paths is now greatly dependent on the reliability of the memory units.

3. A REAL-TIME CONTROL SYSTEM

This section presents a real-time control system scenario, called the Simplified Unmanned Vehicle System (SUVS). The system consists of interacting real-time processes. The conversation scheme is incorporated in SUVS for error detection and recovery among the interacting processes. The scenario is used to illustrate various process configurations and mapping strategies to the HHRH architecture proposed in Section 4. The SUVS will be implemented to be used as a testbed for experimental evaluation on the performance and time complexity of process configurations and error detection schemes.

3.1 Simplified Unmanned Vehicle System (SUVS)

The SUVS consists of three different sets of tasks, i.e., sensor tasks (or sensors), analyzer tasks (or analyzers), and actuator tasks (or actuators). (Note that we use the term "task" here as a sequence of actions defined at the software specification level. A "process" is an implementation of a "task" using a programming language.)

(1) Sensors are input devices such as speedometer, engine temperature meter, direction indicator, vision sensor, and road surface sensor. They periodically provide data to the analyzer tasks.
(2) Analyzers process sensor data and include speed analyzer, direction analyzer, vision analyzer, and surface analyzer. They make decisions which are forwarded to the actuators. They also exchange information among themselves.
(3) Actuators are output devices such as brake, accelerator, handle, and camera handlers. They receive commands from the analyzers and provide control to the system.

The information flow among these tasks is shown in Figure 6. In the following the basic conversation scheme is briefly reviewed. Then the strategies of incorporation of the conversation scheme into the SUVS are illustrated. Our discussion focuses on the implementation of the analyzer tasks.

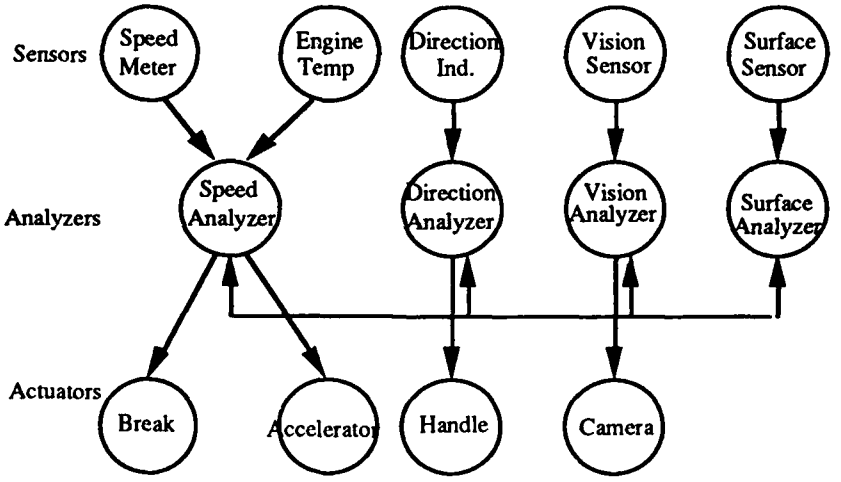


Figure 6. Information Flow of the Simplified Unmanned Vehicle System (SUVS).

3.2 Incorporation of the Conversation Scheme

3.2.1 The Conversation Scheme. The conversation is a two-dimensional enclosure of recoverable activities of multiple interacting processes, in short, a recoverable interacting session [Kim82,Ran75]. It creates a "boundary" which process interactions may not cross. The boundary of a conversation consists of a recovery line, a test line and the walls defining membership as shown in Figure 7. Each participant process contains one or more try blocks

designed to produce the same or similar computational results as well as an acceptance test which is a logical expression representing the criterion for determining the acceptability of the execution results of the try blocks. A recovery line is a coordinated set of the recovery points of interacting processes that are established (possibly at different times) before interactions begin. A test line is a correlated set of the acceptance tests of interacting processes.

A conversation is successful only if all the interacting processes pass their acceptance tests forming the test line. Therefore, the participants are allowed to leave the conversation when all the participants have passed their acceptance tests. If any of the acceptance test fails, all the processes roll back to the recovery line and retry with their alternate try blocks. These alternate try blocks collectively define an alternate interacting session (AIS) where as the set of primary try blocks executed first after the processes enter the conversation define the primary interacting session (PIS). A process that has executed its try block and passed its acceptance test is said to have finished its conversation task.

A process which is inside a conversation cannot interact with a process which is not in the conversation. Conversations must be strictly nested in two dimensions. That is, when conversation C.nest is nested within conversation C, the set of processes that participate in nested conversation C.nest must be a subset of the processes that participate in C, the entire recovery line of C.nest must be established after the entire recovery line of C, and the entire test line of C.nest must be set before the entire test line of C. Mechanization schemes and implementation strategies of the conversation scheme are discussed in detail in [Kim82,Yan89].

The conversation scheme has been adopted as the basic fault tolerance mechanism to be used with the HHRH structure. This choice was made because (1) the scheme deals with recovery actions of interacting processes, (2) software redundancy is provided, (3) real-time recovery is achievable, and (4) the scheme is relatively easy to implement.

3.2.2 Incorporation of Conversations into SUVS. In order to incorporate conversations into a real system, the characteristics of the system (in terms of interaction among tasks) should be carefully analyzed. This vehicle control system has the following characteristics. First, the decision made by one analyzer affects the decision of other analyzer(s). For example, before we change the direction of a car, we may have to reduce the speed, if the current speed is too

fast. This means that the direction analyzer has to cooperate with the speed analyzer. Second, the actions made by the system are not revocable. There may be cases where we cannot compensate or change, even if we find a mistake immediately after an action was done. Therefore, the output actions should be very carefully taken. Finally, any decision should be made within a specified time under all circumstances. (Otherwise, the effect is the same as driving a car while sleeping.)

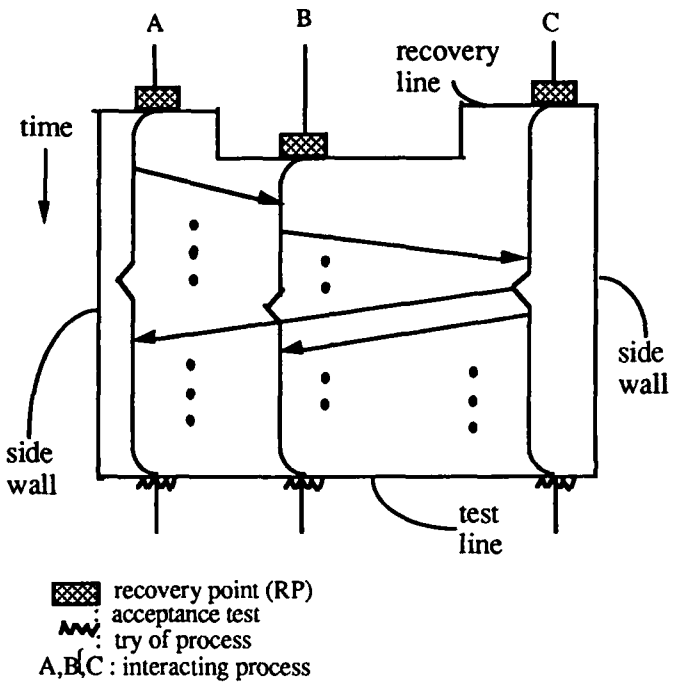


Figure 7. Conversation.

The above three characteristics requires cooperation among the analyzer tasks to properly control the real-time response of the system. Therefore, the following conversation structure is proposed.

(1) Four analyzer tasks, i.e., speed, direction, vision, and surface analyzers, cooperate (i.e.,

exchange information and preliminary decisions) to make decisions on any action.
(2) Output (to actuators) is made only when all of the analyzer tasks agree.
(3) Upon disagreement, the alternate algorithms (or alternate interacting session) are executed.
(4) If they don't reach any final agreement within a specified time or they have failed in all alternate algorithms, the system goes into emergency mode and tries to stop the car in the safest and fastest way.

The conceptual conversation structure and possible information exchanged among tasks are shown in Figure 8. Note that the preliminary decisions made by the tasks are broadcast so that this information may be used for the conversation acceptance test. The acceptability criteria may include (1) whether the decisions made by the speed analyzer and the direction analyzer conflict with each other, (2) whether the decision made by the vision analyzer conflicts with the request made by the speed and/or direction analyzer, and (3) whether the analyzers have made decisions based on correct information.

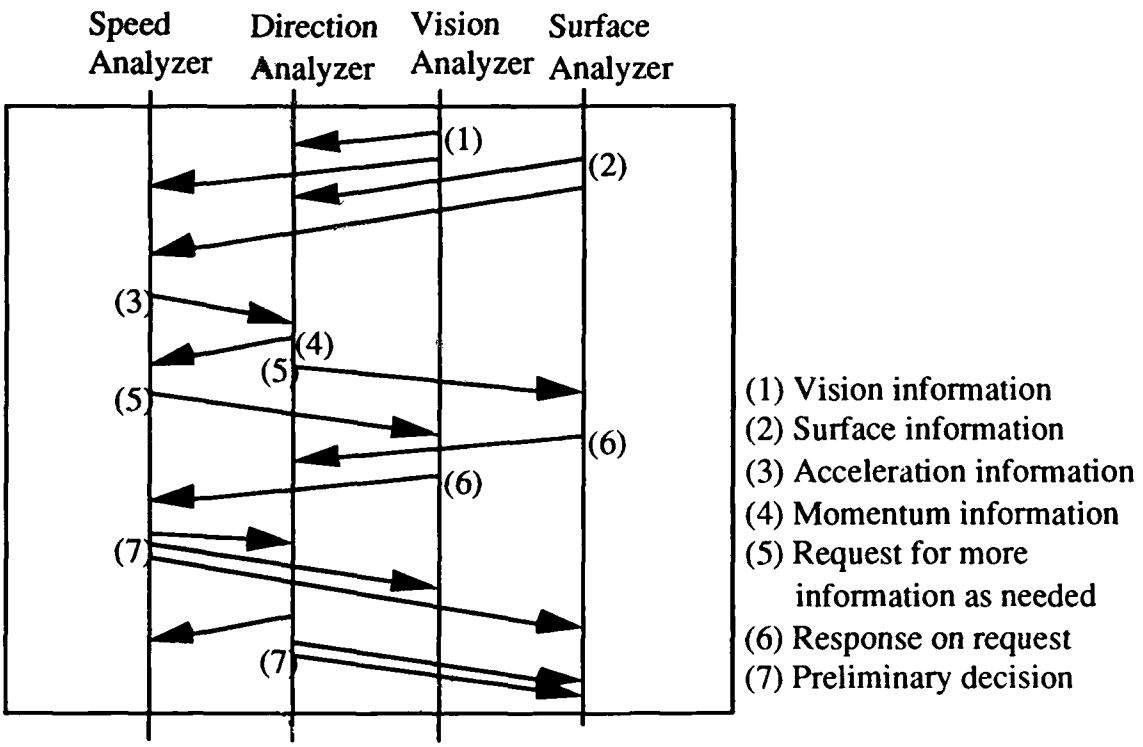


Figure 8. Conceptual conversation structure and information excahnge among four analyzer tasks.

4. PROCESS CONFIGURATION WITH CONVERSATIONS

This section investigates some possible process configuration and mapping strategies relative to the HHRH architecture. The SUVS described in the previous section is used to illustrate these strategies. Two forms of software redundancy and three concepts of error detection and recovery schemes are incorporated in SUVS. Our objective is to achieve system level fault-tolerance, both hardware and software, in consideration of communication and synchronization overhead, reconfigurability after failure and real-time recovery.

4.1 System Configuration

As mentioned in the previous section, four analyzer tasks in SUVS run` concurrently and participate in the same conversation every cycle of executions. They receive inputs from the sensor tasks periodically and make decisions based on current inputs and status under strict timing constraints. These four tasks are to be implemented on the HHRH architecture described in Figure 2. Since there are sixteen nodes in the system, quadruple redundancy can be achieved if each node runs one task. Software redundancy is thus incorporated in two different forms.

(1) Multiple identical copies of processes: Since quadruple redundancy is provided, we may have up to four identical copies of each process running on different nodes.

(2) Multiple versions of processes: Different versions of processes are provided in the form of alternate try blocks in conversations.

Three different concepts of error detection and recovery schemes, namely filtration, consensus, and approval are also incorporated. Filtration is implemented in the form of voting among multiple copies of a process, consensus in the form of comparison among multiple versions of a process, and approval in the form of conversation acceptance test. These are not exclusive but incorporated into the system at various stages. In the following subsections we present three different process configuration and mapping strategies of the SUVS. Advantages and drawbacks of each strategy are also discussed.

4.2 Configuration I: Clusters of Conversations

With this configuration each cluster (i.e., four nodes with shared memory) runs a complete set of processes of the system. As shown in Figure 9, four clusters run identical software simultaneously. Therefore, we may view the system as the collection of clusters, each cluster performs the complete functions.

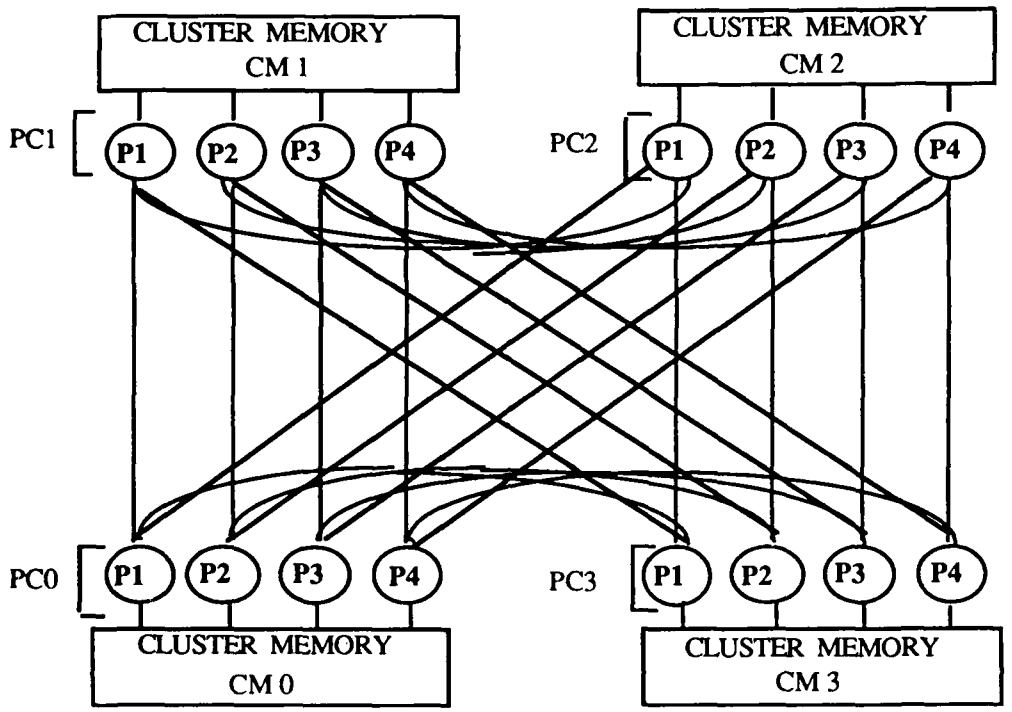


Figure 9. Configuration I: Clusters of conversations.

Since we have quadruple redundancy in software as well as hardware it is possible to incorporate the voting mechanism into various stages of computations and/or communications. (Note that the voting is done independently from the conversation acceptance test. Therefore, these different types of checking mechanisms can improve the system reliability significantly.) For example, we may design the system such that four identical copies of a process take a vote for every message from one process to another. Or we may design the system such that four identical copies of a process take a vote only for the output to the actuator tasks. It is hoped that errors due to hardware failure are masked by voting among multiple (four in this case) copies of a process. Software fault, on the other hand, is assumed to be detected by the conversation acceptance test (CAT). As mentioned in the previous section the CAT is done for the decisions made by the four analyzer tasks before the decisions are output to the actuator

tasks. Upon failure of the CAT, alternate try blocks provided in each process are executed.

The major advantage of this configuration is the modularity of each cluster. Since each cluster performs the complete functions little modification is required when clusters are added into the system or removed from the system . (The only change expected is the voting mechanism.) Therefore, reconfiguration after failure is easy. Another advantage is the fast communication among a set of processes within the cluster through the shared memory. That is the interactions among processes within conversations are done through the shared memory. However, the cost of communication among the identical processes for voting is high. This is because each message is passed through a point-to-point connection among four identical processes. Therefore, the communication overhead would be significant if we want to incorporate voting logic frequently (e.g., voting for every message from one process to another).

4.3 Configuration II: Clusters of Processes

Another way to configure the system is to have each cluster run multiple identical copies of a process. As shown in Figure 10, each cluster runs four identical copies of a process in this configuration. Therefore, the number of clusters should be the same as the number of different processes in the system.

The same error detection and recovery schemes (i.e., the voting among multiple identical copies and the CAT) can be incorporated in this configuration. Since the multiple copies of a process reside in a cluster and communicate through shared memory, the cost of voting among those processes is relatively small compared with the previous configuration. However, communication cost among the four analyzer tasks is high. Also crash of shared memory leads to very clumsy communication among multiple identical copies of a process.

4.4 Configuration III: Parallel Executions of Conversations

With the previous two configurations it is inevitable that all processes roll back and execute the alternate interacting session if CAT fails. (Voting among identical copies of a process cannot detect or mask the failure due to errors in software design and/or implementation.) In some applications, however, because of strict response time requirement

rollback and retry may not be tolerable. One way to handle this problem is to execute the different versions of software (i.e., alternate interacting sessions in this case) simultaneously and take the result of the alternate version if the primary version fails to produce the acceptable result.

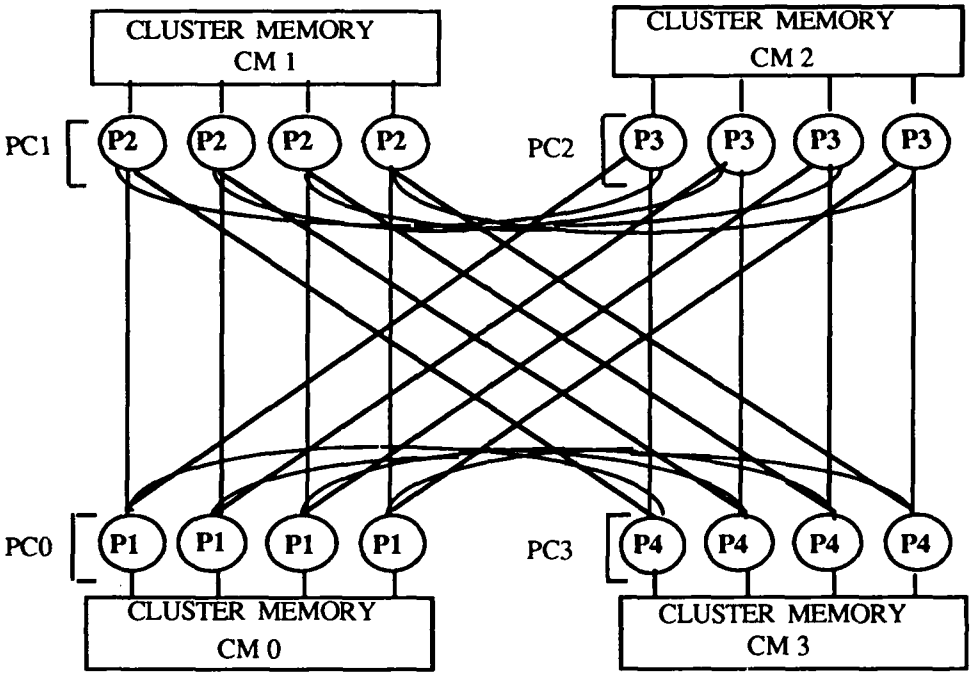


Figure 10. Configuration II: Clusters of processors.

Figure 11 shows one possible configuration of parallel execution of conversations. (Assume that each process is designed with one primary try block and one alternate try block.) As shown in the figure, Clusters 0 and 1 run primary try blocks (i.e., primary interacting session) of the four analyzers, whereas Clusters 2 and 3 run the alternate try block (i.e., alternate interacting session). Should none of the primary try blocks (running on Clusters 0 and 1) succeeds in producing the acceptable result, one of the results produced by the alternate try blocks will be output to the actuators. Since rollback and retry is not required even if there are errors in the primary try blocks this configuration is especially good for time critical applications. It is also possible to incorporate other error detection and recovery schemes, i.e., voting and comparison. Figure 12 shows one possible process configuration with three error detection and recovery schemes. In the upper half of the figure there are four analyzer

processes, each with four identical copies. These four copies of a process take a vote for a message to be transmitted to other analyzer process(es). At the end of every conversation the CAT is performed for outputs to the actuators. In the bottom half of the figure there is another set of processes with the same configuration but run different versions of processes. The results of the two sets (i.e., the set of primary versions and the set of alternate versions), if both have passed in their CAT's, are compared before the results are finally output to the actuators. By comparing these results we can possibly detect errors that have not been detected by the CAT. If one of them fails in the CAT or does not produce the results within a specified time (or timeout) the results produced by the other set are immediately output.

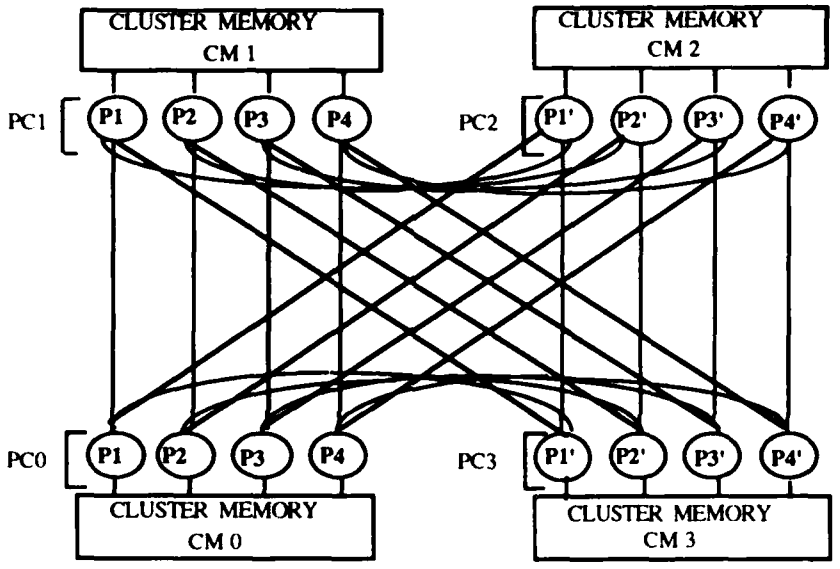


Figure 11. Configuration III: Parallel execution of conversations.

4.5 Discussion

This section described three possible process configuration and mapping strategies to the HHRH architecture shown in Figure 2. Two forms of software redundancy and three different concepts of error detection and recovery schemes, namely filtration, consensus, and approval are incorporated within each configuration. In the first configuration each cluster runs a complete set of processes of the system. In the second configuration each cluster runs multiple identical copies of a process. In the third configuration each cluster runs a complete set of processes, but different versions simultaneously. Although each configuration has its

own merits and drawbacks the third configuration seems the most attractive approach (especially for time critical applications) because the rollback and retry is not required. This is yet determined based on careful analysis of timing aspects, i.e., worst-case execution time and the response time limit (or deadline).

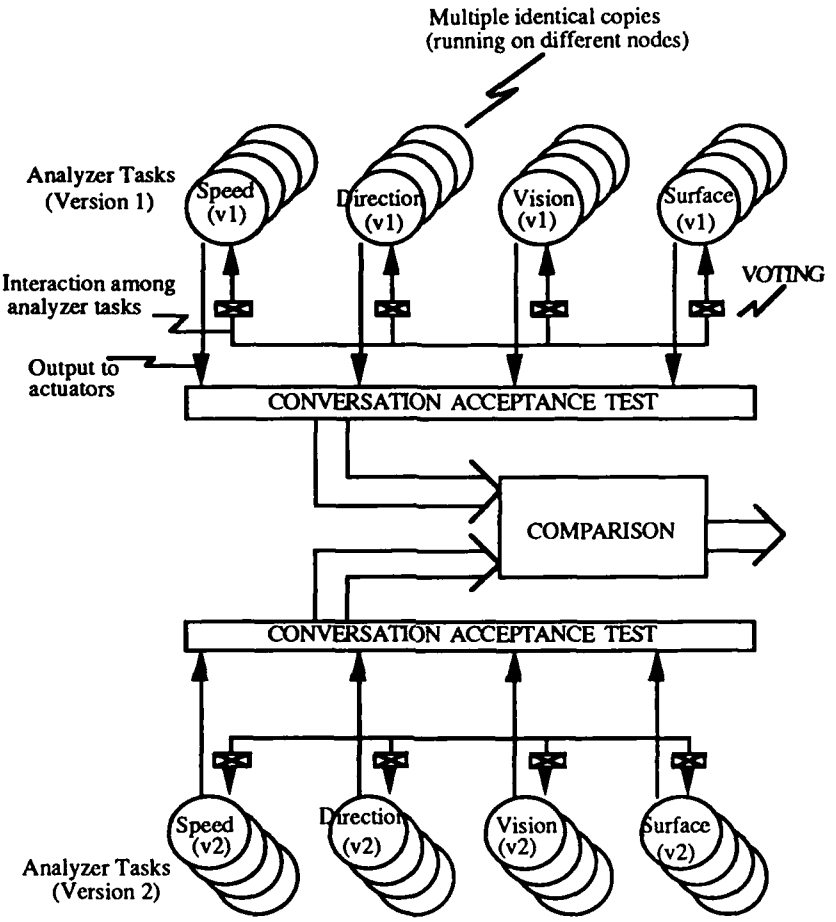


Figure 12. Three error detection schemes with Configuration III.

One of the major factors to be considered for incorporating these approaches is communication and synchronization overhead. Since the HHRH architecture provides two different communication paths, i.e., shared memory and point-to-point connection, the communication cost should be carefully analyzed. Another design consideration is to incorporation of the timeout mechanism. The timeout mechanism is required for every synchronization stage to avoid lockup of other processes. The proper timeout period should be determined based on the analysis of the system behaviour. It is also important to consider the

cases where no agreement is reached among processes (in the case of voting or comparison) or no acceptable result is produced (in the case of CAT). Analyses on the reliability and communication overhead of on different configurations and implementation strategies (in terms of design and implementation cost, runtime cost, timing aspects, and reliability) are under study. Some preliminary result is discussed in the next section.

5. Analysis of Reliability and Communication Overhead

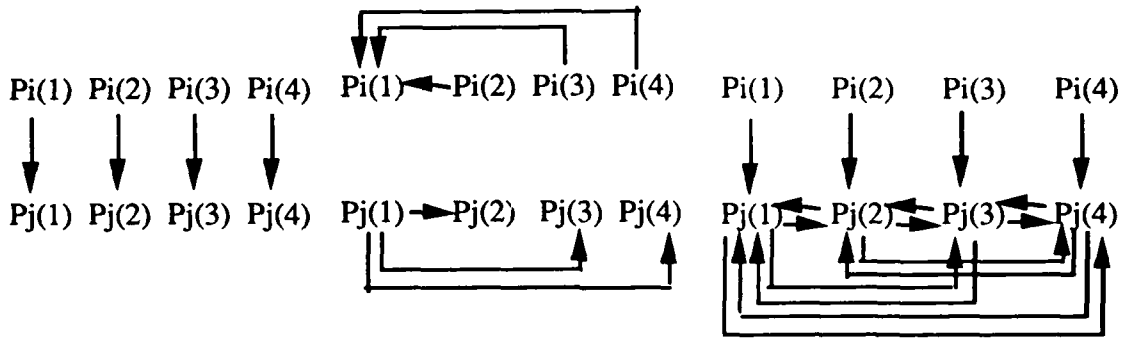
In this section, the reliabilities and the communication overhead are analyzed under the six cases, i.e. three message passing structures for two process configurations, Configuration I and Configuration II, described in Section 4.

5.1 Message Passing Methods

We considered three different message passing structures. Under Structure 1 as shown in Figure 13(a) a message is forwarded without voting to the directly connected receiving node. The structure is very simple to implement, but fault is propagated.

A message is forwarded after a voting under Structure 2 as shown in Figure 13(b). One node (among four sending nodes) is designated as a voter which takes a majority vote against the four outputs. The voted message is forwarded to one of the four receiving nodes, which forwards the message to the remaining three other receiving nodes. Under this structure it is necessary to provide the mechanism for the redesignation of the voter when the original voter fails.

Under Structure 3 shown in Figure 13(c) voting is done by the four receiving nodes. Each sending nodes forwards a message to the directly connected receiving node. Then the receiving nodes exchange the received message among themselves and take a majority vote. This message passing structure can mask fault made during the communication as well as faults made by the sending node. However, the structure suffers from the communication overhead.



(a) Message passing without voting (b) Voting before message passing (c) Voting after message receiving

Figure 13. Message Passing Structures.

5.2 Analysis

Failure rates of node, link, and cluster memory are assumed to be 10^{-5}, 10^{-6}, and 10^{-7} per hour, respectively. In order to a message to be reliable, it is assumed that at least two nodes among four receiving nodes must receive a correct message from the sending node. Let R be the reliability of a message passing at a mission time t. Then the combinatorial reliability models [Joh88] for the three structures are as follows:

Message Passing Structure 1:

Configuration I:

$$R = R_{2\text{-of-}4}\ (R_{node} * R_{CM})$$
$$= 3\exp(-4.04t * 10^{-5}) - 8\exp(-3.03t * 10^{-5}) - 6\exp(-2.02t * 10^{-5})$$

Configuration II:

$$R = R_{2\text{-of-}4}\ (R_{node} * R_{link})$$
$$= 3\exp(-4.4t * 10^{-5}) - 8\exp(-3.3t * 10^{-5}) - 6\exp(-2.2t * 10^{-5})$$

The fault propagation should be considered in this structure. For example, the reliabilities in second step can be changed as follows:

Configuration I:

$$R = R_{2\text{-of-}4}\ (R_{node} * R_{CM} * R_{node} * R_{CM})$$
$$= 3\exp(-8.08t * 10^{-5}) - 8\exp(-6.06t * 10^{-5}) + 6\exp(-4.04t * 10^{-5})$$

Configuration II:

$$R = R_{2\text{-of-}4}\ (R_{node} * R_{link} * R_{node} * R_{link})$$
$$= 3\exp(-8.8t * 10^{-5}) - 8\exp(-6.6t * 10^{-5}) + 6\exp(-4.4t * 10^{-5})$$

Message Passing Structure 2

Configuration I:

$$R = R_{A2} * R_{CM} * R_{1\text{-of-}3}(R_{link})$$
$$= R_{1\text{-of-}3}(R_{node} * R_{link}) * R_{node} * R_{CM} * R_{1\text{-of-}3}(R_{link})$$
$$= (\exp(-4.31t * 10^{-5}) - 3\exp(-3.21t * 10^{-5}) + 3\exp(-2.11t * 10^{-5})) * (\exp(-3t * 10^{-6}) - 3\exp(-2t * 10^{-6}) + 3\exp(-t * 10^{-6}))$$

Congiguration II:

$$R = R_{A1} * R_{link} * R_{1\text{-of-}3}(R_{CM})$$
$$= R_{1\text{-of-}3}(R_{node} * R_{CM}) * R_{node} * R_{link} * R_{1\text{-of-}3}(R_{CM})$$
$$= (\exp(-4.13t * 10^{-5}) - 3\exp(-3.12t * 10^{-5}) + 3\exp(-2.11t * 10^{-5})) * (\exp(-3t * 10^{-7}) - 3\exp(-2t * 10^{-7}) + 3\exp(-t * 10^{-7}))$$

Message Passing Structure 3

Configuration I:

$$R = R_{2\text{-of-}4}(R_{CM} * R_{A2})$$
$$= R_{2\text{-of-}4}(R_{CM} * R_{1\text{-of-}3}(R_{node} * R_{link}) * R_{node})$$
$$= 3D - 8D + 6B$$

where

$$D = \exp(-4.31t * 10^{-5}) - 3\exp(-3.21t * 10^{-5}) + 3\exp(2.11t * 10^{-5})$$

Configuration II:

$$R = R_{2\text{-of-}4}(R_{link} * R_{A1})$$
$$= R_{2\text{-of-}4}(R_{link} * R_{1\text{-of-}3}(R_{node} * R_{CM}) * R_{node})$$
$$= 3B^4 - 8B^3 - 6B^2$$

where

$$B = \exp(-4.13t * 10^{-5}) - 3\exp(-3.12t * 10^{-5}) + 3\exp(-2.11t * 10^{-5})$$

The result confirms that Message Passing Structure 3 is the best in terms of reliability. We also analyze the communication overhead assuming that the message passing time in point-to-point connection is 600 μsec (this is the average latency of iPSC/2 [Aga86]) whereas that via cluster memory is 50 μsec. From the result we can make the following conclusions: Configuration I is generally more reliable than Configuration II. However, communication overhead of Configuration I is generally higher than that of Configuration II. More detail analysis on reliability and communication overhead is reported in [Yoo89].

6. Summary

This chapter reports part of our effort to develop practical design and implemntation approaches for ultra reliable, fault-tolerant computers for critical real-time applications. The proposed appoaches are intended for system-level fault-tolerance, both hardware and software, in parallel and distributed systems. We proposed a hybrid parallel architecture, called Hybrid Higher Radix Hypercube (HHRH) architecture, and process configuration and mapping strategies relative to the HHRH architecture. The HHRH architecture is attractive because it is well suited for ultra reliable, real-time applications due to (1) high bandwidth and predictability in communication between nodes (speed and predictability are the most important factors in real-time systems), (2) inherent redundancy which is essential for fault-tolerant computing, and (3) reconfigurability as required by many applications. We are investigating efficient and reliable message routing algorithms on HHRH architecture. The algorithm should find the optimal path between the sender and receiver in the presence of failures in nodes, links, and/or cluster (or shared) memories. The timing aspects of the proposed algorithm will also be analyzed.

The conversation scheme has been adopted as the basic fault tolerance to be used with the HHRH MPS. A real-time control system scenario, the SUVS, is used to illustrate process configuration with conversations. (As part of the experimental study, C version of SUVS is being implemented on a network of six SUN 3/60 workstations at the University of Texas at Arlington.) Along with the conversation scheme three different concepts of error detection and recovery schemes are incoporated at the various stages of computations and/or communications. An important design consideration is how to coordinate these schemes within the system to maximize the reliability in consideration of implementation and runtime cost.

Although our discussion was based on the SUVS, we believe that the architecture, process configuration strategies, and error detection and recovery schemes are applicable for the design of most fault-tolerant real-time applications in distributed/parallel environments. Experimental as well as analytical evaluation is yet required on the performance (in terms of reliability increase) and time complexity of various process configurations and error detection schemes. We leave these problems for future research.

7. REFERENCES

[Aga86] Agarwal, D.P., Janakiram, V.K., and Pathak, G.C., "Evaluating the Performance of Multicomputer Configuration", IEEE Computer, May 1986, pp. 23-37.

[Bha88] Bhargava, 'Implementation and Analysis of a Hybrid Higher Radix Hypercube Architecture', MS Thesis, The University of Texas at Arlington, April 1988.

[Bhu84] Bhuyan, L. and Agrawal, D.P., "Generalized Hypercube and Hyperbus Structures for a Computer Network", IEEE Trans. on Computers, Vol. 34, April 1984, pp. 323-333.

[Bon87] Bond, J., "Parallel Processing Concepts Finally Come Together in Real Systems", Computer Design, June 1987, pp. 51-74.

[Joh88] Johnson, A.M., "Survey of Software Tools for Evaluating Reliability, Availability, and Seviceability," ACM Computing Survey, Vol. 20, No. 4, December 1988, pp. 227-269.

[Kim82] Kim, K.H., "Approaches to Mechanization of the Conversation Scheme Based on Monitor," IEEE Trans. on Software Eng., May 1982, pp. 189-197.

[Kim85] Kim, K.H. and Yang, S.M., "Fault Tolerance Mechanisms in Real-Time Distributed Operating Systems: An Overview," Proc. PCCS 85, Oct. 1985, pp. 220-229.

[Ran75] Randell, B., "System Structure for Software Fault Tolerance," *IEEE, Trans. on* Software Eng., June 1975, pp. 220-232.

[Ran78] Randell, B., Lee, P.A., and Treleaven, P.D., "Reliability Issues in Computing System Design," Computing Surveys, June 1978, pp. 1123-1165.

[Ren80] Rennels, D.A., "Distributed Fault-Tolerant Computer Systems," IEEE Computer, March 1980, pp. 55-65.

[Sei84] Seitz, C.L., "The Cosmic Cube," CACM, January 1985, pp. 22-33.

[Ser84] Serlin, O., "Fault-Tolerant Systems in Commercial Application," IEEE Computer, Aug. 1984, pp. 19-30.

[Yan89] Yang, S.M. and Kim, K.H., "Implementation of the Conversation Scheme in Loosely Coupled Distributed Computer Systems", Proc. Int'l on Distributed Computing Systems, June 1989, pp. 570-578.

[Yoo89] Yoo, S.M., "Reliability Analysis of the Hybrid Higher Radix Hypercube Parallel Architecture with Conversation Scheme," Master's thesis, the Univ. of Texas at Arlington, May 1989.

CHAPTER III

IMPLEMENTATION OF THE CONVERSATION SCHEME IN MESSAGE-BASED DISTRIBUTED COMPUTER SYSTEMS

1. INTRODUCTION

Since the concept of conversation structuring was introduced in an abstract form as an approach to facilitating cooperative recovery in systems of interacting processes [Ran75], continuous research efforts have been invested to convert the concept into a practical technology. For example, several mechanized structuring schemes containing practical language syntax and associated precise semantics have been formulated [Cam83,Gre85,Kim82,Rus79]. Subsequently, some practical language processing systems have been produced [Kim85], a "recovery metaprogram" has been proposed for efficient design of conversations [Oza88] and the execution costs of conversations have been studied [Kim89]. Conversation design schemes using Petri Nets have also been studied [Tyr86, Wu89]. In [Man89] an attempt is made to utilize some well-known object replication techniques in design of communicating processes with conversation structures.

However, experimental studies of the scheme have been scarce and have not yet progressed to the point of producing concrete results. Moreover, techniques for efficient implementation, especially in distributed computer systems (DCS's), have not been much studied either. This chapter presents issues in implementing conversations in DCS's together with some efficient implementation approaches. While some of the issues discussed in this chapter are common to both tightly coupled parallel processing networks and loosely coupled DCS's, others are unique to the latter types of systems due to the major difference in the inter-process communication costs between the two types of systems. The following three cost factors should be carefully considered in such an implementation.

The first factor is the cost of communication between remote processes. In selecting an efficient implementation strategy for the conversation scheme, the interprocess communication cost plays an important role as will be elaborated in this chapter. The communication cost is primarily determined by the network structure adopted, i.e., network topology, communication medium, and protocol used. In this chapter "message-based DCS's", in which computing nodes communicate with each other via message passing, are considered.

The second factor is the cost of synchronizing processes at the exit of a conversation. As shown in [Kim89], this cost can have significant impact on the performance of the conversation scheme. To reduce this cost, the approach of conversation lookahead, (allowing asynchronous exits from the conversation) was studied in [Kim89] for its potential performance impacts based on an analytic model. The conversation scheme extended with the lookahead capability is called the asynchronously exited conversation scheme whereas the conversation scheme with no lookahead is called the synchronously exited conversation scheme. Under the synchronously exited

conversation scheme, all processes are synchronized at the end of each conversation, i.e., no process is allowed to exit from the conversation earlier than other processes even if the process has passed its acceptance test. On the other hand, under the asynchronously exited conversation scheme processes are allowed to exit from the conversation as they finish their tasks, but the exiting processes maintain the capabilities for their rollback to the beginning of the conversation until all processes have passed their acceptance tests and exited from the conversation.

The third factor is the cost of designing and executing the conversation acceptance test (CAT). In a single-node multiprocess system, the choice between the centralized CAT (i.e., one participant takes care of the non-local acceptance test routine) and the decentralized CAT (i.e., each participant has its own acceptance test routine), has relatively insignificant effect on the system performance. However, in a message-based DCS, the CAT choice affects the system performance to a significant extent. In this chapter, three different approaches to design and execution of CAT's, centralized, decentralized, and semi-centralized, are discussed.

In the next section, approaches to the implementation of synchronous and asynchronous exit control strategies are introduced and the pros and cons of each approach are discussed. An efficient approach to run-time management of recovery information based on an extension of the recovery cache scheme is also discussed. In Section 3, three different CAT execution approaches are presented. The effectiveness of different execution approaches also depends on the way conversations are structured by program designers. The cases of using two different basic types of conversation structures, Name-Linked Recovery Block (NLRB) and Abstract Data Type (ADT) Conversation, are examined in Sections 4 and 5 respectively, in order to analyze the effectiveness of the execution approaches discussed earlier. In spite of the passage of a considerable amount of time since the publication of the first paper that proposed the concept of conversation structuring [Ran75], practical illustrations have been scarce. In Section 6, an unmanned vehicle system is used to illustrate how the approaches discussed in earlier sections for implementation of the conversation scheme can be used in a realistic real-time application. Section 7 is the summary section.

2. SYNCHRONOUSLY EXITED AND SYNCHRONOUSLY EXITED CONVERSATIONS

This section starts with a brief description of the conversation structure. Then the characteristics of synchronously exited and asynchronously exited conversations are discussed. These two approaches are quite different in terms of their impacts on system performance and complexity. A brief comparison is made at the end of this section.

2.1 Basic logical structure of the conversation

A conversation is a two-dimensional enclosure of recoverable activities of multiple interacting processes, in short, a recoverable interacting session [Kim82, Ran75]. It creates a "boundary" which process interactions may not cross. The boundary of a conversation consists of a recovery line, a test line and the walls defining exclusive membership as shown in Figure 1. Each participant process contains one or more try blocks, i.e., program blocks designed to produce the same or similar computational results, as well as an acceptance test which is a logical expression representing the criterion for determining the acceptability of the execution results of the try blocks. A recovery line is a coordinated set of the recovery points of interacting processes that are established (possibly at different times) before interactions begin. A test line is a correlated set of the acceptance test excution-points of interacting processes.

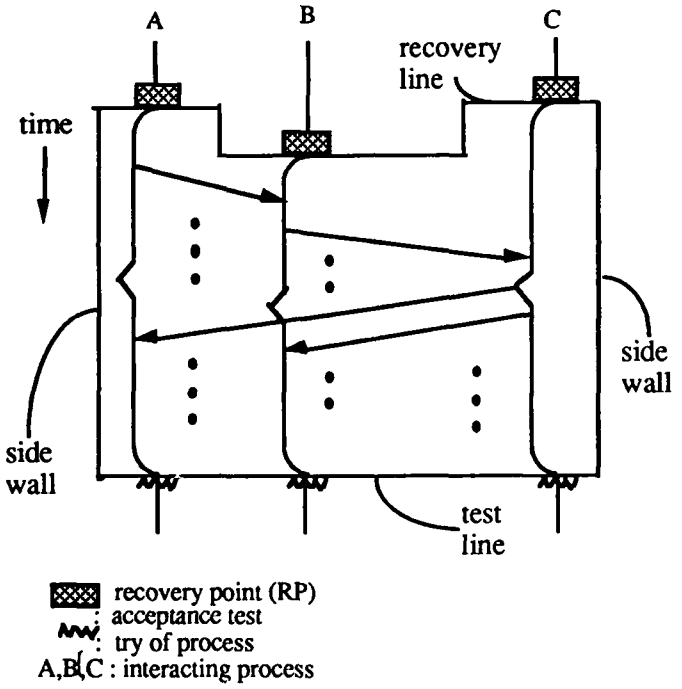


Figure 1. Conversation (adapted from [Kim82]).

A conversation is successful only if all the interacting processes pass their acceptance tests forming the test line. Therefore, the participants are allowed to leave the conversation when all the participants have passed their acceptance tests. If any of the acceptance test fails, all the processes roll back to the recovery line and retry with their alternate try blocks. These alternate try blocks collectively define an alternate interacting session (AIS), whereas the set of primary try blocks

executed first after the processes enter the conversation define the primary interacting session (PIS). A process that has executed its try block and passed its acceptance test is said to have finished its conversation task. A process which is inside a conversation cannot interact with a process which is not in the conversation. Conversations must be strictly nested in two dimensions. That is, when a conversation C_{nest} is nested within another conversation C, the following conditions must be satisfied:
(1) the set of processes that participate in nested conversation C_{nest} must be a subset of the processes that participate in C,
(2) for each process the point of entry into C_{nest} must be after the point of entry into C and thus in a sense the recovery line of C_{nest} is always established "after" the recovery line of C, and
(3) for each process the acceptance test execution-point for C_{nest} must precede that for C and thus the test line of C_{nest} is always set before the test line of C.

2.2 Exit control

In the basic conversation scheme sketched above, processes enter a conversation asynchronously but synchronize themselves before exiting from it. The synchronization considered here is of a special kind specifically required by the conversation scheme and thus is different from the application-dependent synchronization required between cooperating processes. The synchronization can add significantly to the time cost of the conversation scheme. A fundamental approach to reducing the synchronization overhead is the lookahead. Under the lookahead approach each participant process leaves the conversation as soon as it passes its own acceptance test. It does so with the awareness of the possibility that another participant may execute an acceptance test later and fail in the test thus making it necessary for the former to roll back to the recovery line of the conversation. Therefore, the lookahead approach here is an optimistic approach and is aimed at trading increase in recovery costs for reduction of synchronization overhead.

Although it is logically feasible to make provisions for a participant process to execute beyond the unfinished conversation to an unlimited extent, it is a sensible choice to limit the extent of the lookahead with respect to controlling the implementation complexity. In addition, the lookahead should not be allowed to go past the points where critical irreversible actions, e.g., certain critical output actions, are taken. In this chapter, the cases where lookahead is allowed to limited extents are considered.

The conversations which are executed under the lookahead permitting strategy are called the asynchronously exited conversations whereas those executed where the lookahead is not permitted

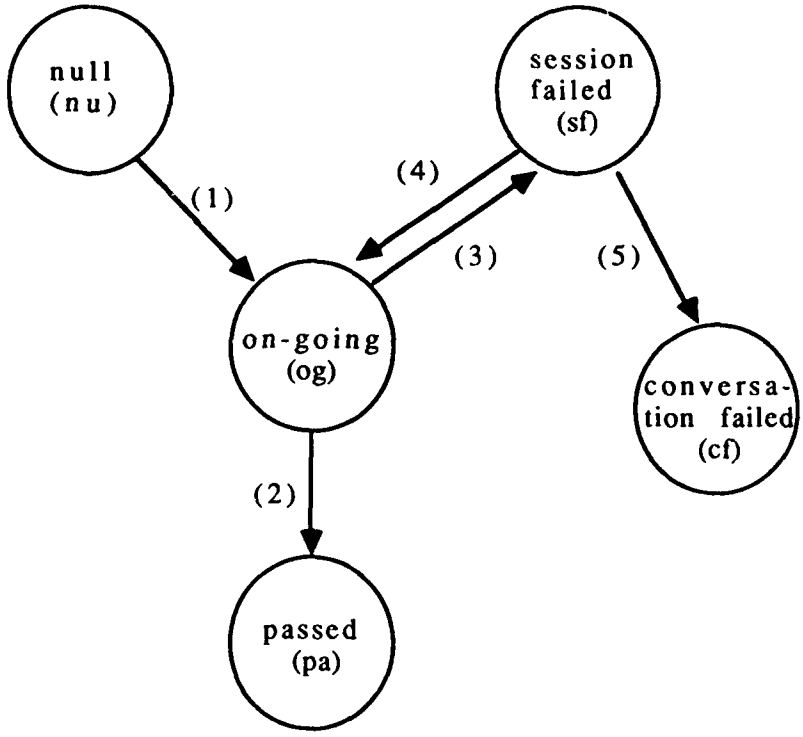
are called the synchronously exited conversations. Although the concepts of synchronously exited and asynchronously exited conversations were introduced in earlier papers [Kim76,Kim89,Rus79], their implementation techniques were not studied in depth. Finite state machine representations of both conversations that can be useful in implementing the conversations are introduced below.

A synchronously exited conversation may transit among the following five different states:
(S1) *null*: None of the participants have entered into the conversation.
(S2) *on-going*: There is at least one participant which is executing its conversation task.
(S3) *passed*: All the participants have passed their acceptance tests.
(S4) *session failed*: At least one participant has failed in its acceptance test.
(S5) *conversation failed*: The conversation has failed due to the failures of all alternate interacting sessions.

Figure 2 shows possible changes in the state of a synchronously exited conversation. Once a participant process fails in its acceptance test (transition (3) in the figure), all the participant processes abandon the try with the current interacting session and later start a retry with the next alternate interacting session.

In the case of an asynchronously exited conversation, a process that has exited from a conversation C1 via lookahead may enter another conversation C2. If some slow progressing participants of C1 are not participants of C2, then it is possible that C2 activities including acceptance tests are completed while C1 remains unfinished. In such a case, C2 should be treated as an unfinished conversation until C1 becomes completed. This is because if C1 fails, then all C2 activities that have taken place must be nullified as a part of the rollback to the recovery line of C1.

Definition 1: A conversation is validated if all the participants of the conversation have passed their acceptance tests.
Definition 2: When a process exits from a conversation and enters another conversation, the latter (the former) conversation is said to be logically after (before) the former (the latter) conversation. On the other hand, when a process in a conversation enters another (nested) conversation (without exiting from the former conversation), the newly entered nested conversation has no logical ordering relationship with any order earlier entered conversations.
Definition 3: A conversation CONV is completely validated (1) if CONV has been validated and there is no other conversation which is not only logically before CONV but also in the on-going state, or (2) if CONV has been validated and all other conversations which are logically before CONV have been completely validated.



(1) Start execution of the primary interacting session.
(2) All the participants have passed their acceptance tests.
(3) A participant has failed in its acceptance test.
(4) Start execution of the next alternate interacting session.
(5) A conversation has failed due to the failures of all interacting sessions.

Figure 2. Change in the state of a synchronously exited conversation.

Definition 4: A conversation CONV is partially validated if CONV is neither in the on-going state nor in the completely validated state. To be more specific, a conversation CONV is partially validated if CONV has been validated and there is at least one conversation which is not only logically before CONV but also in the on-going state.

For example, in Figure 3 conversations CONV1, CONV2 and CONV3 are logically before CONV4. Therefore, CONV4 cannot be completely validated (even if it has been validated) until all of its logically earlier conversations CONV1, CONV2 and CONV3 become completely validated. Also, CONV5 is logically before CONV6. However, CONV5 and CONV6 have no ordering relationship with any other earlier entered conversations (i.e., CONV1, CONV2, CONV3, and CONV4 in this example). Therefore, CONV6 becomes completely validated if it has been validated and CONV5 has been completely validated.

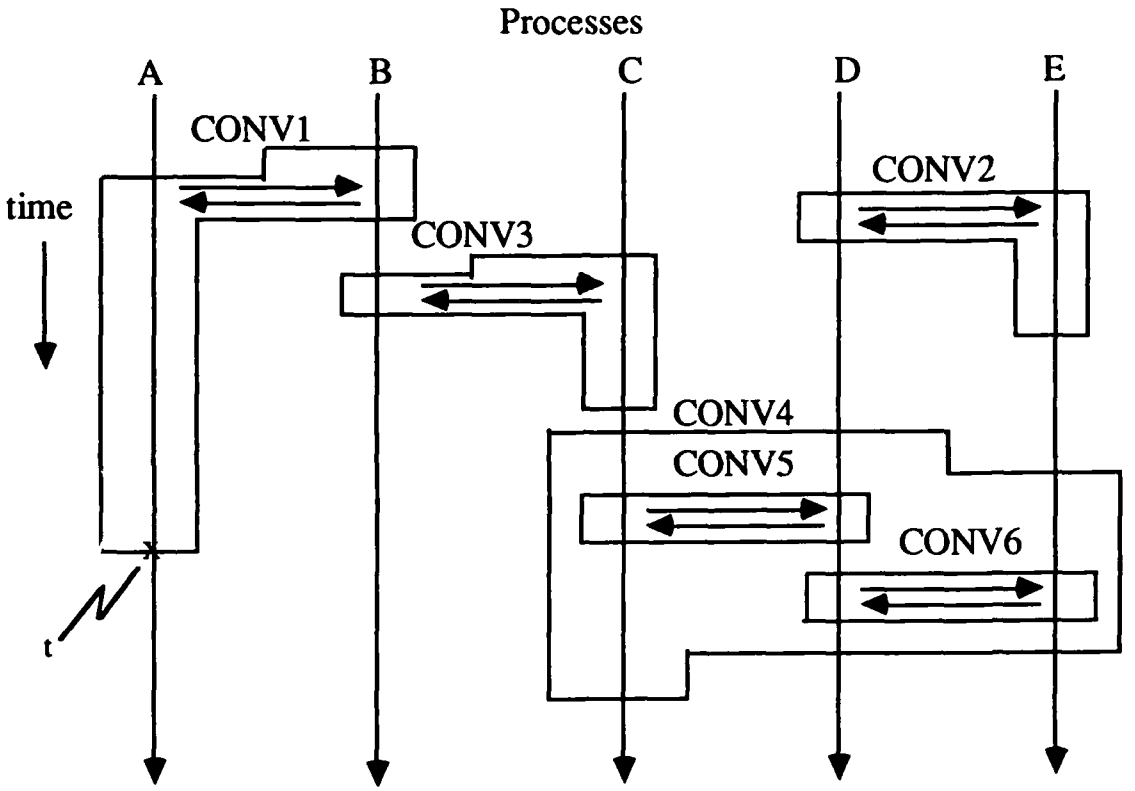


Figure 3. An example of asynchronously exited conversation.

An asynchronously exited conversation may, therefore, transit among the following six different states as shown in Figure 4.

(S1) *null*: None of the participants have entered into the conversation.

(S2) *on-going*: There is at least one participant which is executing the conversation task.

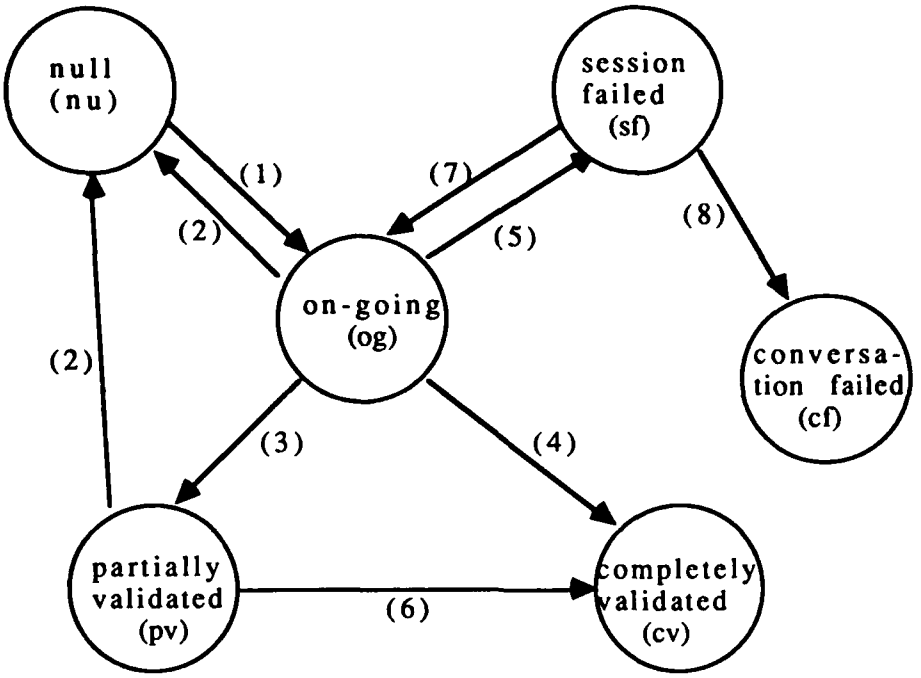
(S3) *partially validated*:

(S4) *completely validated*:

(S5) *session failed*: At least one participant has failed in its acceptance test.

(S6) *conversation failed*: The conversation has failed due to the failures of all interacting sessions.

In Figure 4 the transition (2) means that the on-going or partially validated conversation is nullified when one of the participants must roll back to an older conversation. For example, in Figure 3 Processes A and B participate in CONV1. Then Process B participates in CONV3. Suppose Process A fails at t after Processes B and C completed CONV3 (that is, CONV3 has been partially validated). This results in CONV3 being nullified since Process B has to roll back to the recovery line of CONV1. If a retry of the older conversation (i.e., CONV1) is successful, Process B and C may again execute the conversation (i.e., CONV3) with their primary try blocks (transition (1) in Figure 4).



(1) Start execution of the primary interacting session.
(2) Conversation execution is nullified due to the rollback of a participant to an older conversation.
(3) All the participants have passed their acceptance tests but there is at least one logically earlier conversation which is in the on-going state.
(4) All the participants have passed their acceptance tests and there is no logically earlier conversation which is in the on-going state.
(5) A participant has failed in its acceptance test.
(6) All logically earlier conversations have become completely validated.
(7) Start execution of the next alternate interacting session.
(8) A conversation has failed due to the failures of all interacting sessions.

Figure 4. Change in the state of an asynchronously exited conversation.

Intuitively, the system performance would increase if asynchronous exits of the participants are allowed. The performance improvement would be particularly conspicuous when the acceptance test failure probability is very low (making the rollback infrequent) and the number of participants is large (making the synchronization overhead substantial). A previous analytic study on the performance of the conversation scheme based on a queueing network model [Kim89] confirmed this property. It also showed that the performance of a system would be significantly affected by the synchronization required of the processes in exiting from a conversation, not by the failure probability of each process. For example, suppose a process is allowed to make "one conversation lookahead", which means that a process can continue lookahead as long as it has not

exited from more than one unfinished (i.e., on-going or partially validated) conversation. The analytic study based on a queueing network model showed that by allowing one conversation lookahead the performance could increase about 46% when the number of participants is six, the failure probability is almost zero, and the amount of computation that a process performs inside a conversation is on the average about 10% of the total computation the process performs. On the other hand, the performance would decrease only 2% when the failure probability increases from zero to 0.05 (which is higher than that of most practical systems) and other parameters including synchronous exit remain unchanged. (Further details are referred to [Kim89].)

2.3 Management of recovery information

Under the asynchronous exit approach, the information needed for rollback, e.g., try block entry-points, prior values of the non-local variables which have been changed after the process entered the conversation, etc., should be kept until the conversation is completely validated. To keep this information, each process maintains a table called the "conversation record" for each unfinished conversation. The entries of the conversation record are shown in Figure 5. The table is created when the process initially enters the conversation and removed when the conversation becomes completely validated.

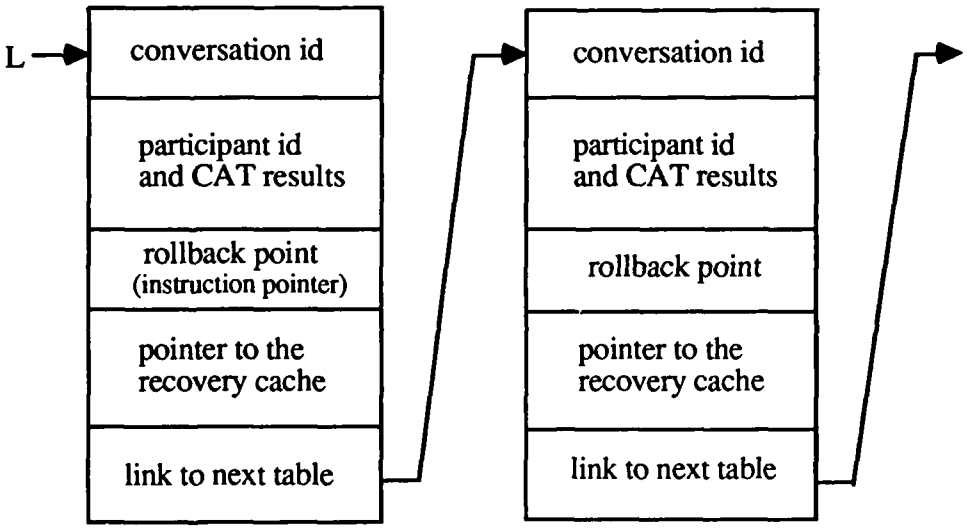


Figure 5. Conversation record.

The conversation records are linked hierarchically as they are created. All partially validated or on-going level-0 conversations (i.e., conversations with no parent conversations) are linked linearly. The conversation record of a nested conversation (i.e., level-1 conversation which is

immediately nested within a level-0 conversation or lower level conversation) is linked under its parent record. Therefore, records of various unfinished conversations (including partially validated conversations) are in general related in the form of a tree. Figure 6.a illustrates nested conversations and Figure 6.b shows how the conversation records are managed in Process B under the following scenario: (In Figure 6.b a solid box represents a conversation record for an on-going conversation, a dotted box for a partially validated conversation, and a shaded box for a completely validated conversation.)

(a) Process B is executing a level-1 conversation CONV2, i.e., CONV1 and CONV2 are on-going conversations.
(b) Process B completed CONV2 and then CONV1 while Process A has completed CONV2. Therefore, CONV2 has been completed validated within CONV1.
(c) Process B has entered another level-0 conversation CONV3 while Process A is still executing CONV1.

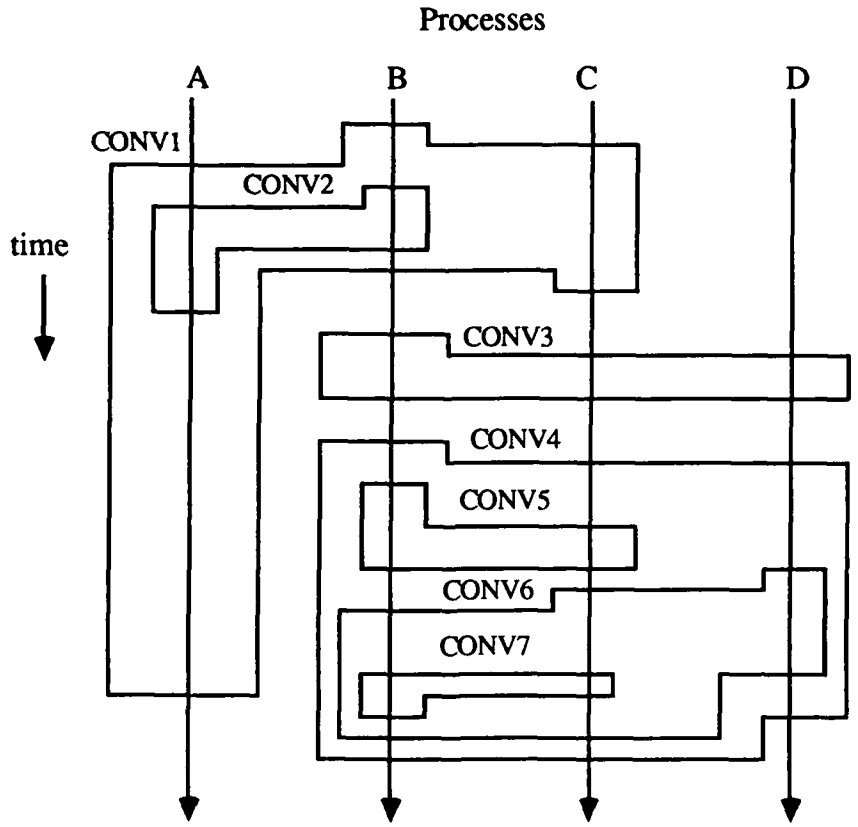


Figure 6.a. An illustration of nested conversation.

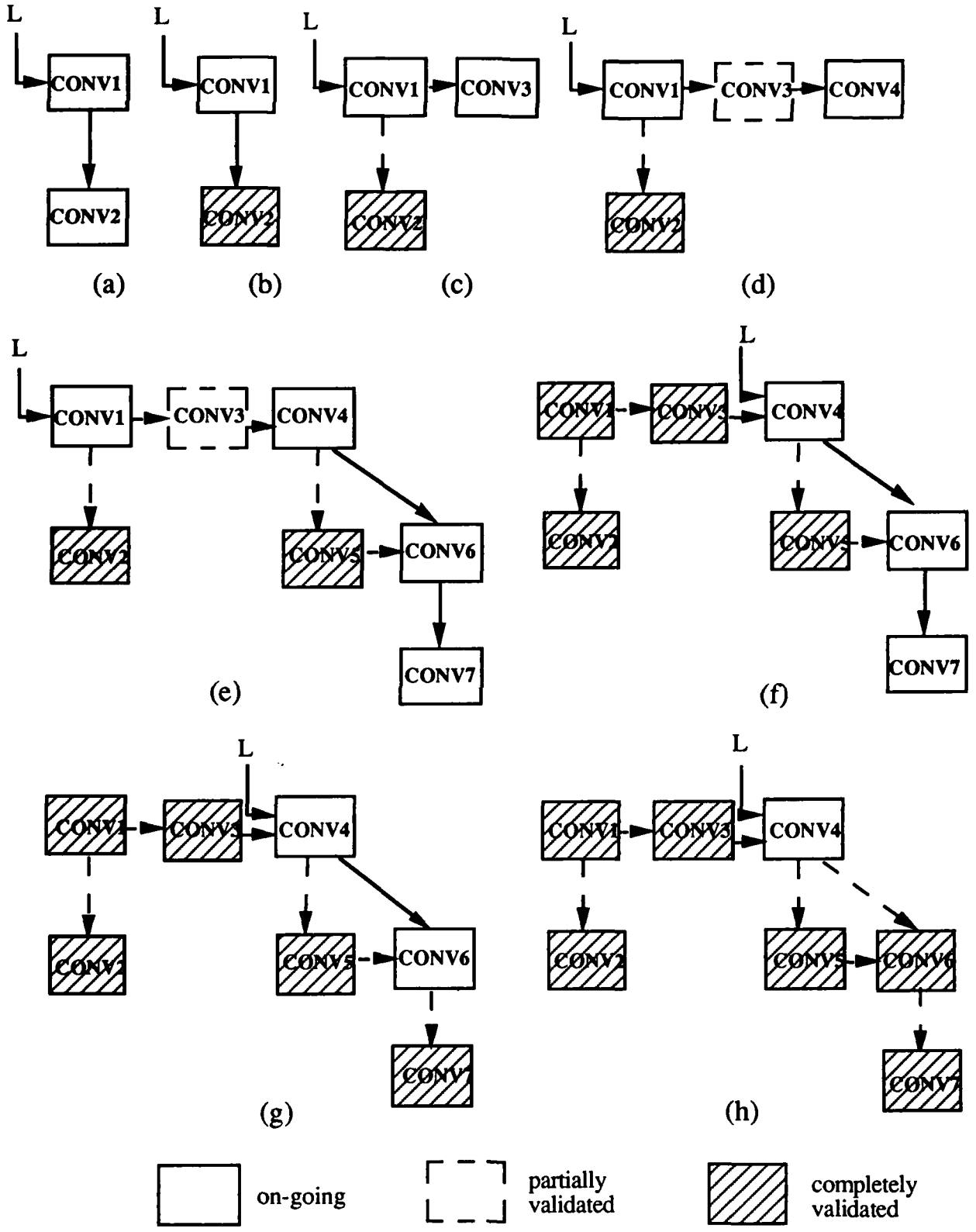


Figure 6.b. Conversation record structure for nested conversations.

(d) Processes B, C and D completed CONV3 and have entered another level-0 conversation CONV4. CONV3 has been partially validated since CONV1 which is logically before CONV3 is still on-going.
(e) Processes B and C entered and completed CONV5, have entered CONV6, and then CONV7 (a level-2 conversation). Therefore, CONV5 was completely validated within CONV4 and CONV6 and CONV7 are on-going. CONV1 has not been completely validated yet, because Process A is still in CONV1.
(f) Process A has finally completed CONV1. CONV3 becomes completely validated since its logically earlier conversation, CONV1, has been completely validated. The conversation records of CONV1 and CONV3 are removed from the link.
(g) Processes B and C have completed CONV7. Therefore, CONV7 is completely validated within CONV6.
(h) Processes B, C and D have completed CONV6. CONV4 is now the only on-going conversation.

Another implementation consideration is how to keep the original values of non-local variables that have been modified after the process entered a conversation. An approach, which is an extension of the recovery cache scheme developed in [Hor74, And76] to facilitate efficient execution of recovery blocks, is discussed here. Under the recovery cache scheme a special run-time stack, named a cache stack, is used to save the old values of non-local variables. When a non-local variable is assigned a value for the first time after the process execution has gone past a recovery point, the existing value together with the variable name is saved into the top region of the cache stack. Therefore, using the values in the top region of the cache stack, the process can roll back to the recent recovery point. This also means that whenever a non-local variable is to be assigned a value, it must be checked whether it is the first assignment after a recovery point . In the rest of this subsection we briefly discuss how the recovery cache can be extended to facilitate conversation execution.

The cache stack of an ordinary stack structure for each process is sufficient for facilitating synchronously exited conversations. However, that is not the case for facilitating asynchronously exited conversations. The reason is because the stack needs to shrink in two directions: (1) to shrink from the top when a process has to roll back, which requires restoration of the modified non-local variables, and (2) to shrink from the bottom when the oldest conversation has been completely validated, which requires discard of the old values. Assume that no nested conversations are used for the time being. In a normal situation (i.e., when there is no fault) the stack grows in one direction (when the old values of non-local variables are saved in the stack) and

shrinks from the other direction (when the saved old values are discarded from the stack). Therefore, this type of data structure can be implemented as a stack with the "leaking" base and the two pointers associated with the stack, "stack_top" and "stack_base", are dynamically changing. The stack consists of stack-segments, each created when the host process enters a new conversation.

When a process enters a conversation the current stack_top is saved in the conversation record of the newly entered conversation, more specifically, the "pointer to the recovery cache" field shown in Figure 5. Suppose the process needs to roll back to the beginning of the conversation which has failed or been nullified. The values of non-local variables saved in the segment(s) of the cache stack delimited by the current stack_top and the recovery cache pointer in the conversation record, are restored. (If the same variables appear more than once in the segment(s), then the oldest values should be restored.)

In the cases of nested conversations (i.e., level-1 or lower level conversations), even after a nested conversation is completely validated within its parent conversation, the values of the non-local variables which have been modified within the nested conversation should be kept until its parent conversation becomes completely validated. In other words, the segment which belongs to the nested conversation in the recovery cache stack gets merged into the segment belonging to the parent conversation. When two stack segments get merged into one, the same variables may appear twice, one appearance in the segment of the nested conversation and the other appearance in the segment of the parent conversation. There are two approaches to handling this problem: (1) discard the useless values (the younger ones) immediately, or (2) keep the younger values as they are in the stack until the entire merged segment becomes useless. Under the second approach if the restoration of the variable becomes necessary (due to rollback), the younger values are simply discarded without being used. Although the first approach saves some memory space, the run-time overhead under the approach is higher than under the second approach. Therefore, the second approach seems more suitable for real-time applications. (In fact, if the first approach is to be pursued, then it is generally more storage-efficient to use a tree of cache stacks in which each stack is used in support of one conversation only.)

In Figure 7.a, CONV1 and CONV3 are level-0 conversations and CONV2 is a nested (level-1) conversation within CONV1. Suppose that in Process B
(1) non-local variables p, q, and s are assigned new values within CONV1;
(2) non-local variables q and r are assigned new values within CONV2;

(3) non-local variables r and t are assigned new values after completion of CONV1 and before entry into CONV3; and
(4) non-local variables q and u are assigned new values within CONV3.

Figure 7.b shows how the recovery cache stack is managed at run-time in Process B. The stack grows in the upward direction. The left column is for variable names and the right column is for saved values.
(b1) Initially both the stack_top and the stack_base point to the bottom of the stack.
(b2) Process B is about to enter CONV2.
(b3) Process B is about to leave CONV1 (assuming that CONV2 has not been completely validated.)
(b4) Process B is about to enter CONV3. (CONV2 has not yet been completely validated.)
(b5) Process A has completed CONV2. Therefore, CONV2 has become completely validated within CONV1. (If Process A has failed in CONV2 then the values of q, r, s and t should be restored. Since r appears twice, the older value 4 is restored.)
(b6) Process A has completed CONV1, thereby making CONV1 completely validated. (Variables q and u have been modified within CONV3.)
(b7) Processes B and C have completed CONV3, thereby making it completely validated.

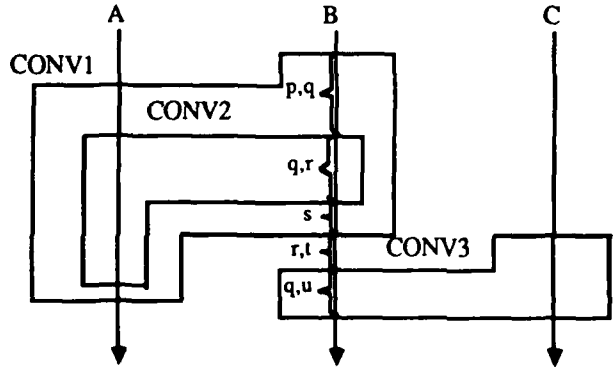


Figure 7.a. An example of an asynchronously exited conversation.

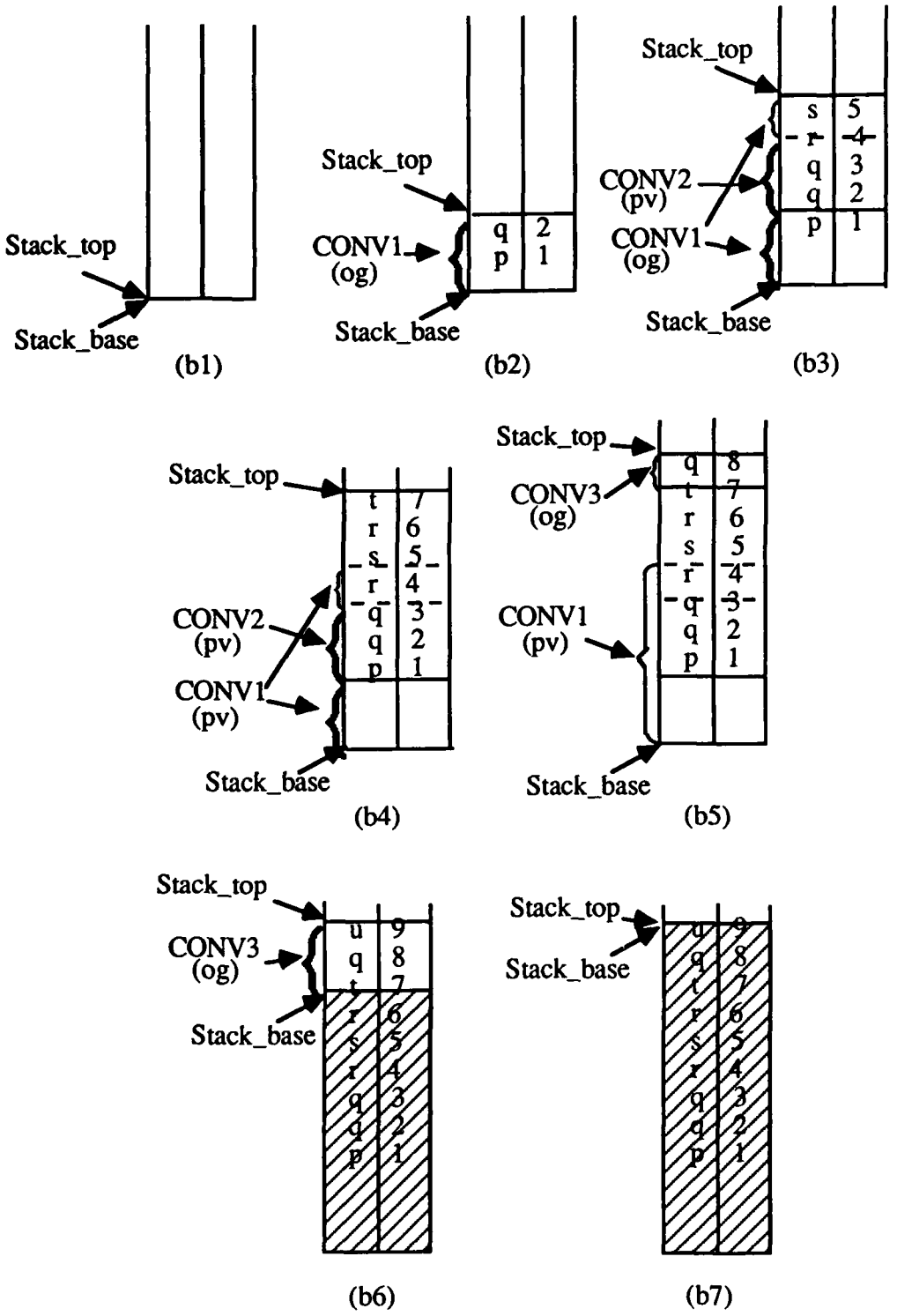


Figure 7.b. Snap shots of the recovery cache stack in Process B of Figure 7.a.

2.4 Discussion

The asynchronous exit approach increases the performance at the cost of increase in implementation complexity. The process must keep the history information until the conversation becomes completely validated. As discussed in the previous section, run-time overhead due to handling of conversation records and recovery cache stacks would increase rapidly as the lookahead level increases. Therefore, the one- or two-conversation lookahead strategy seems the most practical for a wide range of applications. (Also, analytic study results showed that the incremental performance gain expected when changing from the one-conversation lookahead strategy to the two-conversation lookahead strategy would be relatively insignificant in comparison to the incremental gain expected from the change from the synchronous exit strategy to the one-conversation lookahead strategy [Kim89].) In addition, in order to support fast rollback, it is necessary to incorporate the interrupt mechanism in the system. That is, when the IO handler receives a failure or rollback message, it immediately interrupts the corresponding process to minimize the waste of time.

3. CONVERSATION ACCEPTANCE TEST

This section describes three different major approaches to execution of the conversation acceptance test (CAT): centralized CAT, decentralized CAT, and semi-centralized CAT. Advantages and disadvantages of each approach are discussed.

3.1 Centralized CAT

In this approach only one designated participant, named "head" participant, contains the complete CAT routine. Therefore, the head participant executes the CAT when all the participants have finished their execution of try blocks, and thereafter it broadcasts the CAT result to other participants. Since the code that implements the CAT is not scattered among the participant processes this approach has an advantage of not requiring the decomposition of the CAT routine. This property is valuable where the CAT is designed as a single function.

In the centralized CAT approach, it is possible to designate the head participant in two ways: statically and dynamically. With static designation, the predetermined head participant executes the CAT. With dynamic designation, on the other hand, the last participant that completes the conversation task executes the CAT. Under the synchronous exit approach the effect of a choice between static and dynamic designations is relatively insignificant since all other participants must wait until the last participant completes the conversation task. However, in the cases of asynchronous exits, the dynamic designation approach generally performs better than the static

designation approach. This is because under the static approach some extra work is required for the last participant to inform the head participant of completion of the conversation task. Nevertheless, the static approach seems more practical in the sense that it makes it easier to debug and monitor the system.

The total number of messages needed for CAT under this approach is N, the number of participants in the conversation. (In fact, this number is the same for all three CAT execution approaches as will be shown in the following sections. However, the messages communicated under the three approaches are different in length and number of destination processes.) In this approach, N-1 of the N messages would be of a relatively long type because the messages include the values of the variables needed for CAT. Therefore, the centralized approach may lead to relatively large communication overhead due to these sizable messages from the participants to the head participant. Another factor to consider in using this approach is that the malfunctioning of the head participant process causes the loss of the entire CAT function. Various ways to avoid such a loss of the CAT function are conceivable but they all represent additional costs.

3.2 Decentralized CAT

In this approach each participant performs its own acceptance test, and the participants exchange their results with each other. Therefore, every process receives the results of other participants and figures out by itself the total result of the CAT execution.

One of the advantages of this approach is the symmetry that exists among all participants with respect to CAT execution. This makes it simple to implement. However, decomposition of the CAT function is sometimes a costly burden on the programmer. Although automated decomposition is conceivable, its practicality requires further study. Also, all N messages needed for CAT are broadcast messages and thus each message has N destination processes including the sender itself. Therefore, if an efficient broadcasting channel is not available, the number of CAT-related messages exchanged among participants may become very large, although each message will be short. (That is, the messages include "pass" or "fail" information only.) Therefore, in such an environment the communication cost becomes very high.

3.3 Semi-centralized CAT

This approach compromises the above two approaches in such a way that the "local" acceptance test is done by each participant and the total CAT result is determined by the head participant. That is, each participant performs its own acceptance test and sends the result to the head participant. The head participant judges the success or failure of the CAT depending upon

whether all the reports received are success reports or not, and then broadcasts the CAT result. In order to facilitate speedy recovery this approach should be implemented whenever possible in such a way that a failure report of a participant is made in a broadcast form so that all participants may begin their rollback actions immediately. The main advantage of this approach is the small communication overhead; all N messages are short messages and among them N-1 messages are one-to-one messages. The semi-centralized approach, however, shares one deficiency with the decentralized CAT approach: it is necessary to decompose the CAT function.

3.4 Discussion

Table 1 summarizes the number of messages needed for CAT under each approach. As discussed in the previous sections the total number of messages is the same for all three cases, but the messages communicated under the three approaches are different in length and number of destination processes. For example, as mentioned earlier, the messages from the participants to the head participant under the centralized CAT approach are of the relatively long type because the messages include the values of the variables needed for CAT . On the other hand, the CAT-related messages required in the semi-centralized and decentralized CAT approaches include "pass" or "fail" information only. Therefore, the size of each message is very short and fixed. Nevertheless, all the messages in the decentralized CAT approach need to be broadcast whereas in the centralized and semi-centralized CAT approaches only one message (the total CAT result message broadcast by the head participant) needs to be. It seems reasonable to conclude that the semi-centralized CAT approach is the best in terms of message traffic.

In contrast to the decentralized or semi-centralized CAT approach, the centralized CAT approach does not require any local acceptance test to be done by each participant. This possibly degrades detection and recovery performance because the CAT is not evaluated until all the participants complete the conversation tasks. Under the decentralized or semi-centralized CAT approach, it is possible to have participants abandon the conversation tasks if one of the participants has failed in its local acceptance test. This reduces unnecessary computation time although the amount of gain is highly application-dependent. The relatively high fault latency associated with the centralized CAT approach can be a serious drawback in many safety-critical applications such as that to be discussed in Section 6.

The choice should also be made based on the following two factors: (1) characteristics of the application program such as process structure and reliability consideration, and (2) characteristics of the communication system such as network topology, communication medium,

and protocol used. Also, the conversation structuring scheme used by the program designer favors a certain CAT execution approach. This will be further elaborated in Sections 4 and 5.

Message type \ Policy	Centralized	Decentralized	Semi-centralized
Total number of messages for CAT	(N - 1)(l) + 1(sn)	N(sn)	(N - 1)(s) + 1(sn)
one-to-one large* messages (l)	N - 1		N - 1
one-to-one small messages (s)	1	N	1
one-to-n small messages (sn)			
Maximum number of CAT-related messages from/to a participant	2	N	2
Head participant case	N		N

Number of participant = N
l: long message
s: short message
sn: short, broadcast message

* This message contains the values of the variables needed for CAT.

Table 1. Comparison in number of messages for CAT.

4. IMPLEMENTATION OF THE NLRB SCHEME IN MESSAGE-BASED DCS'S

This and the next sections describe how some structuring schemes used by the program designer and the execution approaches discussed in preceding sections can be used in combination in implementing conversations in message-based DCS's. The structuring schemes selected for discussion in this chapter are the Name-Linked Recovery Block (NLRB) scheme and the Abstract Data Type (ADT-) Conversation scheme described in [Kim82,Rus79].

Message-based DCS's considered in these two sections have the following characteristics:
(1) Each node contains a processor, a local memory, and an IO handler.
(2) Each node runs one or more software processes.

(3) Communication between processes in different nodes is done through a bus with the aid of IO handlers. (Broadcasting is facilitated).
(4) The IO handler in the destination node receives and puts a message into the mailbox of the destination process.

4.1 NLRB scheme

The basic idea of the NLRB scheme is to extend the recovery block (RB) construct with a conversation identifier field as follows [Kim82,Rus79]: [*conv* C:] *ensure* T *by* B1 *else by* B2 ... *else by* Bn *else error* where C is the conversation identifier, T the acceptance test, and Bi, $1 \leq i \leq n$, the try blocks. The set of name-linked RB's, each executed by a different process but having the same conversation identifier, compose a conversation construct.

Although this scheme has several deficiencies as pointed out in [Kim82], the scheme is believed to be suitable in some distributed environments for the following reason. In some message-based DCS's, nodes are geographically dispersed. It is very likely that in such application environments processes on different nodes are designed largely independently. In such an environment the NLRB scheme is more natural for use than other schemes such as the ADT-Conversation scheme [Kim82].

4.2 Syntax adopted and message format

The following syntax is an extension of the original NLRB syntax and the extended part is the "participant" field: [*conv* C:] *participants* PROCA, PROCB, ... *ensure* T *by* B1 *else by* B2 ... *else by* Bn *else error* where PROCA, PROCB, ... are the process ids of the conversation participants. (Each process has a unique process id in the system.)

In the following subsections, execution of NLRB's under the two different exit control approaches and the two different CAT execution approaches, semi-centralized and decentralized, are considered. The centralized CAT approach was not considered because each participant of the NLRB conversation is designed to have its own acceptance test routine and placing all the participants' acceptance test routines in one node did not seem competitive with other two approaches in terms of resulting execution performance.

The generic message format adopted is shown in Figure 8. Note that it is of a multicast type sent to more than one destination process. By doing this the number of messages communicated among the processes can be reduced. Messages are classified largely into two types: normal message and CAT result message. In the case of a CAT result message, the data

field is empty. As will be shown in the following subsections, different types of CAT result messages are required for different implementations.

ML: message length in number of bytes (2 bytes)
T: message type (1 byte)
N: number of destination process (1 byte)
D: destination process id's (1 byte per destination)
S: source process id (1 byte)
Data: contents (m bytes)
EOM: end of message (2 bytes)

Figure 8. Message format.

4.3 Approach N1 for execution of NLRB's: Synchronous exit and semi-centralized CAT execution

Under the synchronous exit approach, history logging is not necessary because a participant can exit the conversation only when all the participants pass their acceptance tests. Therefore, its implementation is relatively simple.

The types of CAT result messages used in this execution approach are as follows:
(1) local success notice (LS): This message is sent to the head participant when the participant has passed its local acceptance test.
(2) failure notice (FA): This message is sent to the head participant when the participant has failed in its local acceptance test.
(3) conversation success notice (CS): This message is broadcast when the head participant has concluded the CAT to be a success.
(4) conversation failure notice (CF): This message is broadcast when the head participant has concluded the CAT to be a failure.

In normal cases (i.e., when there is no fault) N messages are needed for each conversation where the number of the participants is N; those are N-1 LS messages from the participants to the head participant and one CS message from the head participant to others. For example, if there are six participants in a conversation construct, six messages are needed for each execution of the conversation: five one-to-one messages are 8 byte-long each and one broadcast message is 12 byte-long (because this broadcast message has five destinations). If the head participant receives an FA message, it immediately notifies other participants by broadcasting the CF message and then all

participants should roll back. Therefore, in order to minimize the recovery time it is desirable to incorporate an interrupt mechanism for receiving an FA or CF message.

4.4 Approach N2 for execution of NLRB's: Synchronous exit and decentralized CAT execution

The execution approach is almost the same as the previous one (Approach N1). As mentioned earlier, however, there is no head participant with the decentralized CAT approach. The success or failure notice from each participant is broadcast to all other participant processes. Therefore, in normal cases N LS messages are needed for each conversation. For example, with six participants six broadcast messages are needed and each message is 12 byte-long. Upon receiving an FA message the participant rolls back for retry. Besides the LS and FA messages, no other types of CAT result messages are required.

4.5 Approach N3 for execution of NLRB's: Asynchronous exit and semi-centralized CAT execution

As discussed in Section 2, implementation of asynchronously exited conversations is much more costly. Also, many more types of CAT result messages are needed than in the case of the synchronous exit:

(1) complete local validation notice (CV): This message is sent to the head participant when a process has passed its acceptance test and all the conversations which the process had participated is prior to entering the conversation currently being exited has been completely validated.

(2) partial local validation notice (PV): This message is sent to the head participant when a process has passed its acceptance test and there is a conversation which the process had participated in prior to entering the conversation currently being exited but has not been completely validated.

(3) failure notice (FA): This message is sent to the head participant when a process has failed in its acceptance test.

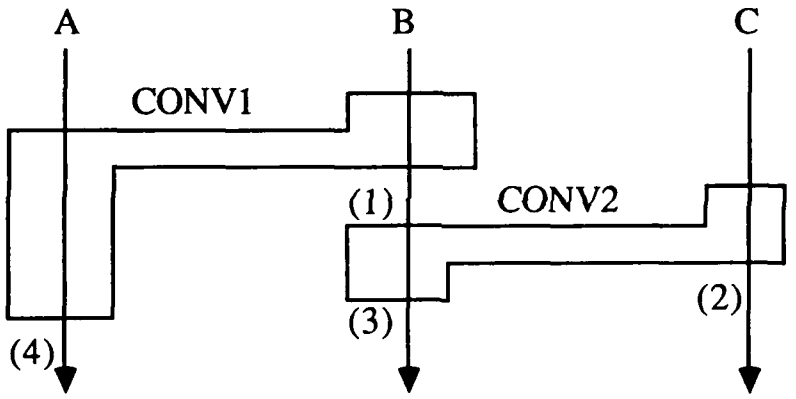
(4) conversation success notice (CS): This message is broadcast when the head participant has concluded the CAT to be a success. That is, the conversation is completely validated.

(5) conversation failure notice (CF): This message is broadcast when the head participant has concluded the CAT to be a failure.

(6) rollback notice (RO): This message is sent to/from the head participant of a conversation when a participant process learns that the conversation which it participated in earlier has become a failure.

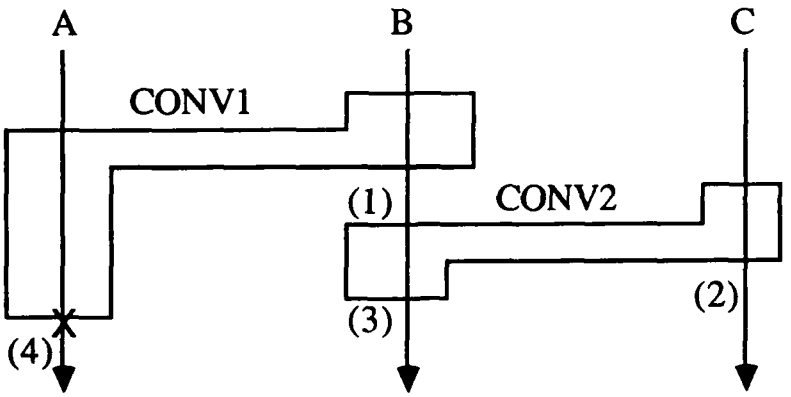
(Note: The PV message is not an absolute necessity. However, it is believed that the PV message is helpful in implementing and testing the system.)

Figure 9 shows two different execution scenarios under N3. Process A is the head participant of CONV1 and Process C is the head participant of CONV2. In both scenarios Processes B and C complete CONV2 with no failure before Process A completes CONV1. Therefore, CONV2 is not completely validated until CONV1 is since CONV1 is logically before CONV2. In the first scenario (shown in Figure 9.a), by the time Process A passes CAT of CONV1 both CONV1 and CONV2 become completely validated. Consequently, in Process B the conversation records for CONV1 and CONV2 are removed. In the second scenario (shown in Figure 9.b), on the other hand, Process A fails at (4), which results in CONV2 being nullified since Process B has to roll back to the recovery line of CONV1. Process B sends the RO message to Process C (which is the head participant of CONV2) and retries CONV1 with the next alternate try block. Later, Processes B and C are allowed to use their primary try blocks when they reenter CONV2.



History	Message	Conversation State	
		CONV1	CONV2
(1) B passes CAT of CONV1	B-->A: CV(CONV1)	og	--
(2) C passes CAT of CONV2		og	og
(3) B passes CAT of CONV2	B-->C: PV(CONV2)	og	pv
(4) A passes CAT of CONV1	A-->B: CS(CONV1)	cv	pv
	B-->C: CV(CONV2)	cv	cv
	C-->B: CS(CONV2)	cv	cv

Figure 9.a. Asynchronously exited conversation: Scenario I.



History	Message	Conversation State CONV1	CONV2
(1) B passes CAT of CONV1	B-->A: CV(CONV1)	og	--
(2) C passes CAT of CONV2		og	og
(3) B passes CAT of CONV2	B-->C: PV(CONV2)	og	pv
(4) A fails CAT of CONV1	A-->B: FA(CONV1)	sf	pv
	B-->C: RO(CONV2)	sf	nu
	C-->B: RO(CONV2)	sf	nu

Figure 9.b. Asynchronously exited conversation: Scenario II.

By allowing asynchronous exit the number of messages needed for each conversation may increase. That is, in normal cases the participants exchange N-1 CV messages, one CS message, and up to N-1 PV messages. For example, if there are six participants, five CV messages are 8 byte-long each, one CS message is 12 byte-long, and each PV message is 8 byte-long. The number of PV messages needed may become a non-negligible source of overhead as the scope of the lookahead increases.

4.6 Approach N4 for execution of NLRB's: Asynchronous exit and decentralized CAT execution

This execution approach is almost the same as the previous case. Since there is no head participant under the decentralized CAT approach, the result of the non-local acceptance test is not broadcast. Consequently, CS and CF messages are not required. In normal cases N CV messages (and up to N PV messages) are broadcast. Therefore, with six participants all six CV messages are 12 byte-long. Each PV or RO message is also 12 byte-long.

4.7 Discussion

As expected, asynchronously exited conversations require more messages than synchronously exited conversations, although the overhead does not seem serious with one- or

two-conversation lookahead. The message traffic incurred under the semi-centralized CAT approach and that under the decentralized CAT approach are almost the same. However, the load on each participant (due to incoming and outgoing messages) in the decentralized CAT approach is higher than that in the semi-centralized CAT approach as shown in Table 1. The difference becomes larger as the number of participants increases.

We did not examine the centralized CAT execution cases closely in this section since the NLRB scheme is most likely applicable to "loosely coupled" network environments where processes on different nodes are designed largely independently. Consequently, the CAT execution by one process/node is not natural in such environments. Moreover, the centralized CAT approach requires long messages which may result in too much communication overhead in loosely coupled network environments. Nevertheless, it is possible to apply the implementation strategies of the centralized CAT approach which will be discussed for the case of ADT-Conversation in Section 5 to the NLRB scheme.

5. IMPLEMENTATION OF THE ADT-CONVERSATION SCHEME IN MESSAGE-BASED DCS'S

5.1 ADT-Conversation scheme

The Abstract Data Type (ADT-) Conversation scheme was proposed to remedy the shortcomings of the NLRB scheme [Kim82], which stems from the scattered appearance of the constituent RB's of a conversation structure in the program text. In the ADT-Conversation scheme, the conversation construct is structured in the form of an abstract data type.

A possible syntactic structure is shown in Figure 10. The participant enters a conversation by calling a procedure, say CONV.PROCA, of which the name and the formal parameters are listed in the participant area. The role of CO (conversation object) in the figure is to facilitate interprocess communication within the associated conversation. In other words, CO is accessible only within the conversation. This prevents information smuggling by a process participating in the conversation.

Basically, there are two options in executing the CAT. One is to decompose the CAT into segments, each executed by a different participating process. The CAT execution can be done with either the decentralized or the semi-centralized approach. The other is to have one process execute the entire CAT after all the participating processes have completed their executions of conversation

tasks. In the case of the first option, the implementation strategies should be the same as those discussed in the previous section (i.e., NLRB scheme cases). Therefore, in the following subsections 5.3 and 5.4 strategies for implementing the ADT-Conversation scheme with the centralized CAT approach in combination with either the synchronous or the asynchronous exit approach are discussed.

5.2 Syntax adopted and message format

The syntax given in Figure 10 can be incorporated into most of the target implementation languages, although some variations are possible. The ADT-Conversation incorporated into Path Pascal [Kol80] was implemented by the investigators [Kim85]. The same message format given in the previous section (Figure 8) is used here again.

```
type C = conversation
  <const & type declaration>                          "can declare nested conversation"
  participants
        PROCA ( "formal parameters" );
        PROCB (        .....                  );
        ..........
  var
        CO: c-object-type;                            "conversation object declaration"
        ..........
  <CAT function declaration>                          "conversation acceptance test"
  <procedures & functions declaration>
  ensure CAT
  by begin                                            "primary interacting session"
                PROCA: begin ..... end;
                PROCB: begin ..... end;
                ..........
                end;
  elseby begin                                        "alternate interacting session"
                PROCA: begin ..... end;
                PROCB: begin ..... end;
                ..............
                end;
        ............
  elseerror
endconversation;
```

Figure 10. ADT-Conversation.

5.3 Approach A1 for execution of ADT-Conversation: Synchronous exit and centralized CAT execution

In the centralized CAT approach, only one participant executes the acceptance test and broadcasts the result. Therefore, only two types of CAT result messages are needed.
(1) conversation success notice (CS): This message is broadcast when the CAT has succeeded.
(2) conversation failure notice (CF): This message is broadcast when the CAT has failed.

However, as discussed in Section 3, the participants must send the values of the variables needed for CAT to the head participant. Since these messages are longer than simple "pass" or "fail" messages the communication overhead of this approach is higher than those of other approaches. For example, suppose that there are six participants, including the head participant which executes CAT, and each participant provides 50 bytes data for CAT execution. Then each conversation requires five 58 byte-long messages and one 12 byte-long message. Moreover, in general, recovery time under this approach is longer than under the others because CAT cannot be performed until all participants complete the conversation tasks and provide information needed for CAT. (Under the decentralized or the semi-centralized CAT approach it is possible to have the participants abandon their conversation tasks if one participant has failed in its local acceptance test.)

5.4 Approach A2 for execution of ADT-Conversation: Asynchronous exit and centralized CAT execution

With asynchronous exit the pass of the CAT does not necessarily make the conversation completely validated. The conversation success notice is deferred until the conversation becomes completely validated. Three types of CAT result messages are needed.
(1) conversation success notice (CS): This message is broadcast when the CAT has succeeded and all the participants are irrevocable.
(2) conversation failure notice (CF): This message is broadcast when the CAT has failed.
(3) rollback notice (RO): This message is sent to the head participant of a conversation when a participant process learns that the other conversation which it participated earlier has become a failure.
The messages needed for CAT in this approach are basically the same as those in the previous approach (Approach A1). That is, in normal cases five 58 byte-long messages and one 12 byte-long message are needed assuming that there are six participants and each participant provides 50 bytes data for CAT execution.

5.5 Discussion

The ADT-Conversation scheme has a number of advantages over the NLRB scheme. Among others, (1) each interacting session is presented as a single unit in the program text and thus easier to read and (2) it is not necessary to decompose the CAT into distributed routines of which collective effect is generally harder to comprehend. All six implementation approaches (combinations of three different CAT approaches and both synchronous and asynchronous exit cases) can be applied to ADT-Conversations. However, it is the easiest to apply the centralized CAT approach to ADT-Conversations because decomposition of CAT's could be a burden on the program designer. Table 2 summarizes how three CAT execution approaches are applicable to NLRB's and ADT-Conversations.

	Centralized CAT	Deentralized CAT	Semi-centralized CAT
NLNB	less applicable	most applicable	applicable
ADT-Conversation	most applicable	applicable	applicable

Table 2. Applicability of three CAT execution approaches.

Table 3 illustrates the communication costs incurred under the six different implementation approaches (combination of three different CAT approaches and both synchronous and asynchronous exit cases). We assume here that there are six participants and each participant provides 50 bytes data to the head participant in the case of the centralized CAT approach. We also assume that transmission speed is 1 μsec per bit (i.e., 1 Mbps transmission). Two different amounts of fixed protocol overhead incurred in each message transmission are considered: 10 μsec in one case and 100 μsec in the other case. As shown in the table the communication time cost under the centralized CAT approach is almost three times of that under the decentralized or the semi-centralized CAT approach even with the 100 μsec fixed protocol overhead. As this fixed protocol overhead decreases, the ratio will increase. On the other hand, as the transmission speed becomes faster the difference becomes smaller. It also shows that the decentralized CAT approach requires a little more communication cost than the semi-centralized CAT approach.

Communication Cost / Approach	Type of messages	Comm. Time Cost (10 μsec fixed protocal overhead)	Comm. Time Cost (100 μsec fixed protocal overhead)
Centralized CAT		2476 μsec	3016 μsec
Synchronous Exit	5(1-to-1) + 1(broadcast)	2476 μsec	3016 μsec
Asynchronous Exit	5(1-to-1) + 1(broadcast)		
Decentralized CAT		636 μsec	1176 μsec
Synchronous Exit	6(broadcast)	954 μsec	1764 μsec
Asynchronous Exit	9(broadcast)		
Semi-entralized CAT		476 μsec	1016 μsec
Synchronous Exit	5(1-to-1) + 1(broadcast)	698 μsec	1508 μsec
Asynchronous Exit	8(1-to-1) + 1(broadcast)		

* 3 PV messages are assumed in the asynchronous exit case.

Table 3. Communication costs incurred under six implementation approaches.

6. SIMPLIFIED UNMANNED VEHICLE SYSTEM: AN EXAMPLE

The previous sections dealt with general strategies for implementing the conversation scheme into message-based DCS's. In this section, a simplified unmanned vehicle system (SUVS) is used to illustrate the conversation implementation strategies.

6.1 Scenario

The SUVS consists of three different sets of tasks, i.e., sensor tasks (or sensors), analyzer tasks (or analyzers), and actuator tasks (or actuators).

(1) Sensors are input devices such as speed meter, engine thermometer, direction indicator, vision sensor, and road surface sensor. They periodically provide data to the analyzer tasks.

(2) Analyzers process sensor data and include speed analyzer, direction analyzer, vision analyzer, and surface analyzer. They make decisions which are forwarded to the actuators. They also exchange information among themselves.

(3) Actuators are output devices such as brake, accelerator, handle, and camera handlers. They receive commands from the analyzers and cause the controlled objects to change their states.

The information flow among these tasks is shown in Figure 11. In the following subsection we illustrate the strategies for incorporation of the conversation scheme into the system. Our discussion focuses on the implementation of the analyzer tasks.

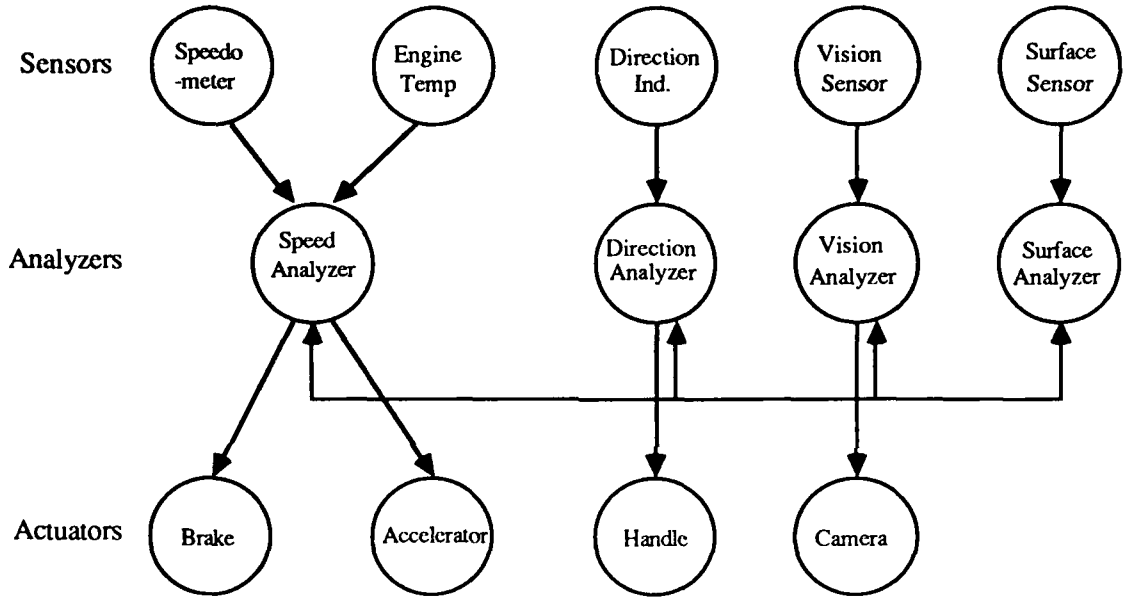


Figure 11. Information flow of Simplified Unmanned Vehicle System (SUVS).

6.2 Implementation strategies

6.2.1 System architecture

A bus-structured computer network, as shown in Figure 12, is assumed for this implementation. Each node has its own local memory and database and runs a single analyzer task. Tasks communicate with each other via message passing. Nodes are also connected to sensors and actuators.

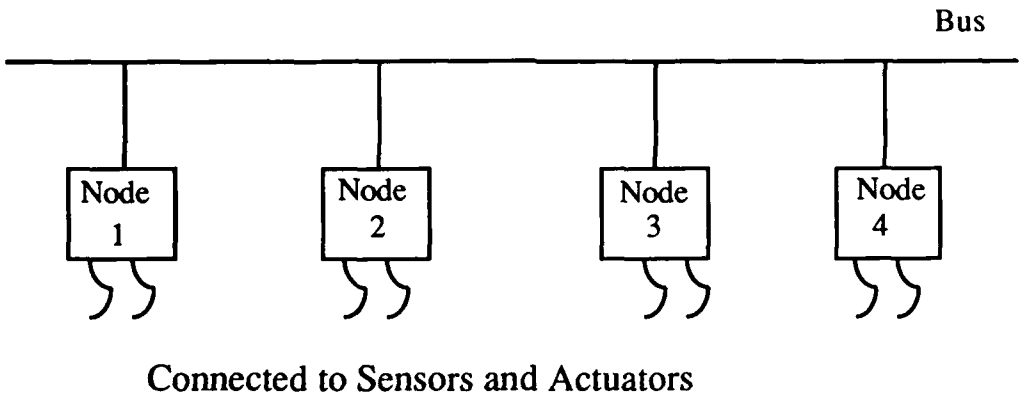


Figure 12. Computer network to implement the SUVS.

6.2.2 Incorporation of the conversation scheme

In order to incorporate conversations into a real system, the characteristics of the system (in terms of interaction among tasks) should be carefully analyzed. This SUVS has the following characteristics. First, the decision made by one analyzer affects the decision of other analyzer(s). For example, before the direction of a car is changed, the speed may have to be reduced, if the current speed is too fast. This means that the direction analyzer has to cooperate with the speed analyzer. Second, the actions taken by the actuators are not revocable. There may be cases where we cannot compensate or change, even if we find a mistake immediately after an action was done. Therefore, the output actions should be very carefully taken. Finally, any decision should be made within a specified time under all circumstances. (Otherwise, the effect is the same as driving a car while sleeping.)

The above three characteristics cause it to be a requirement for the analyzer tasks to cooperate in order to properly control the real-time response of the system. Therefore, the following conversation structure seems useful.

(1) Four analyzer tasks, i.e., speed, direction, vision, and surface analyzers, cooperate (i.e., exchange information and preliminary decisions) to make decisions on any action.

(2) Output (to actuators) is made only when all of the analyzer tasks agree that the system is in a safe state.

(3) Upon disagreement, try again with the alternate algorithms provided.

(4) If they don't reach any final agreement within a specified time or they have failed in all alternate algorithms, the system goes into an emergency mode and tries to stop the car in the safest and fastest way.

The conceptual conversation structure and possible information exchanged among tasks are shown in Figure 13. Note that the preliminary decisions made by the tasks are broadcast so that this information may be used for the conversation acceptance test. The acceptability criteria may include

(1) whether the decisions made by the speed analyzer and the direction analyzer conflict with each other,

(2) whether the decision made by the vision analyzer conflicts with the request made by the speed and/or direction analyzer, and

(3) whether the analyzers have made decisions based on correct information.

Each analyzer also needs to check whether the current inputs are consistent with the outputs made before. For example, if the output action is "reducing speed", then we expect a reduced speed after a while. If the inputs are not consistent, we should suspect either the actuator or the sensor, or both. Therefore, an emergency action is required. Further details on the semantics of this conversation are given later in section 6.2.5.

6.2.3 Exit control

Intuitively, the synchronously exited conversation scheme is suitable for the conversation sketched in the previous section because the conversation is followed by critical output actions of the analyzers. The outputs (to the actuators) made by the analyzers are irrevocable and sometimes critical, thus making it dangerous to allow asynchronous exit in most cases.

However, we may need to allow a special kind of asynchronous exit (i.e., to send the output to the actuator before the total CAT result is determined) to handle an emergency situation. For example, the speed analyzer has decided to reduce the speed quickly after it received the information about an unexpected object from the vision analyzer and/or surface analyzer. In such a case, all the analyzers abandon their current execution and restart the conversation based on the new information.

6.2.4 CAT execution

Although all three CAT execution approaches (centralized, decentralized, and semi-centralized) are applicable, the decentralized and semi-centralized CAT approaches seem more suitable mainly because safety is the major concern in this type of applications and fast recovery is very important. In other words, the relatively high fault latency characteristics makes the centralized CAT approach less attractive in this application. Also, since "fail-stop", i.e., to stop the car if the system does not reach any final agreement, is allowed as a safe emergency action in SUVS, it is beneficial to utilize a conservative approach which may generate more "false alarms"

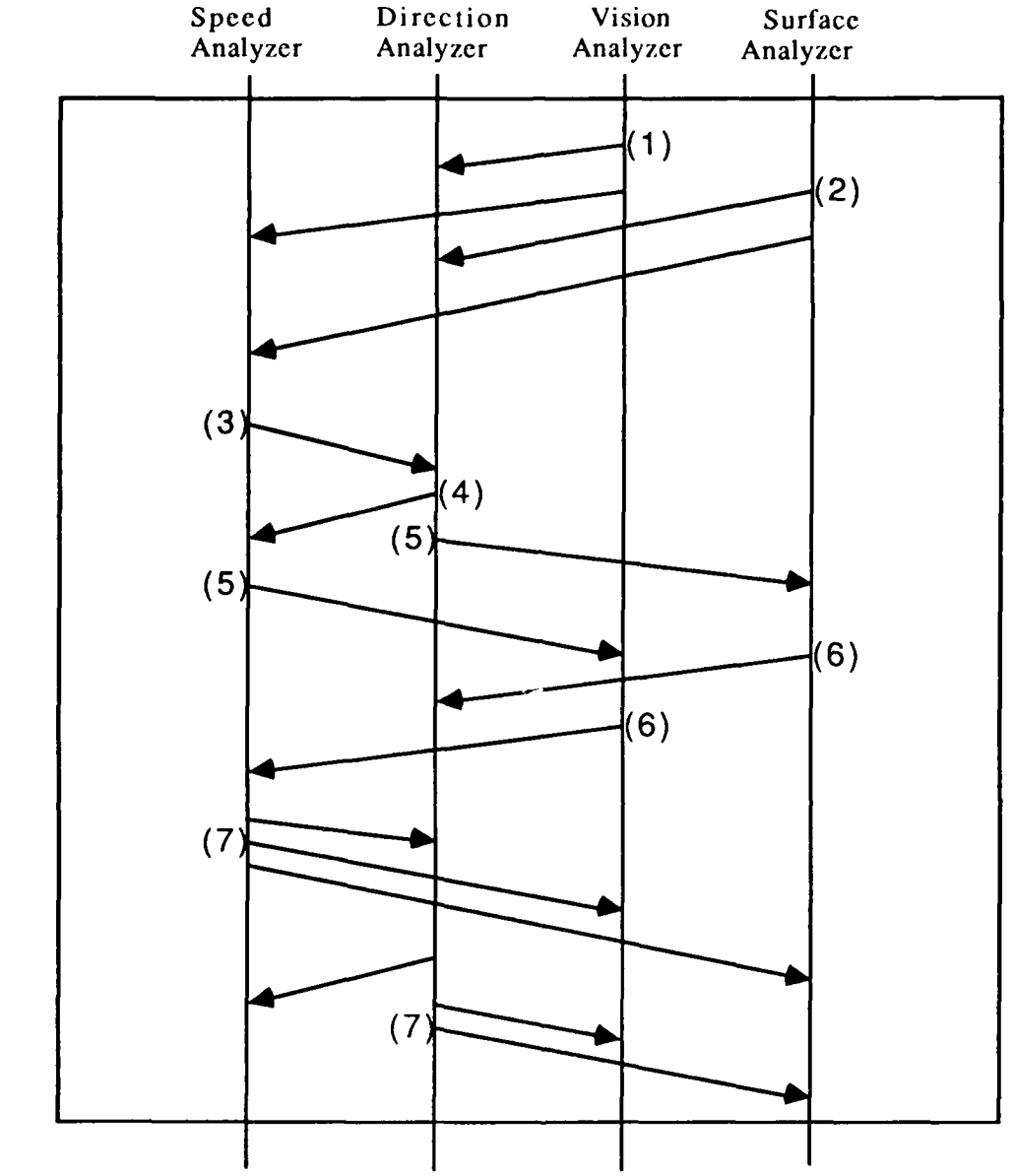


Figure 13. Conceptual conversation structure and information exchange among the tasks.

than other approaches. False alarms do little harm but late alarms or the absence of necessary alarms can be catastrophic. The decentralized and semi-centralized CAT approaches are more conservative approaches in the sense that under the approaches the system does not suffer from the kind of catastrophe that is possible under the centralized CAT approach due to abnormal behavior

of the sole generator of alarms, i.e., the head participant. If the total CAT result is a success, the analyzers output proper command messages to the actuators. Otherwise, all analyzers roll back and try again with alternate try blocks. In this kind of applications there is generally no need to use the previous input data (which were already used for the failed execution) for retry if the new input data can be easily obtained.

6.2.5 The fault-tolerant SUVS

In this subsection we describe the fault-tolerant SUVS (SUVS extended with conversations) in more detail. Figure 14 depicts the high-level logic of an analyzer task. (This logic is applicable to all four analyzer tasks.) The analyzer receives new input data periodically (every 10 msec in this illustration) from the corresponding sensor(s). The input data are checked to see whether they are consistent with the outputs made before. Then the data are exchanged among the analyzers and used to determine the next outputs.

```
task analyzer;

    every 10 msec do  ( on "timeout" => invoke emergency routine; )
        receive input data from the sensor(s) and analyze them;
                                        -- for validity and consistency check
        exchange the data among the analyzers;
        make a decision;
        exchange the decisions made along with the input data used among the analyzers;
        perform CAT;
        if "pass" => forward the decision to the actuator(s);
        if "fail" => roll back and retry;
    end do every;

end task analyzer;
```

Figure 14. High-level logic of an analyzer task in SUVS.

Once a decision is made by an analyzer it is broadcast to other analyzers. These preliminary decision results are used in parts of the CAT performed under the decentralized (or semi-centralized) execution approach. The exchanged information is used to ensure that the decisions made by the analyzers should not conflict with each other. Then the result of each CAT-segment (performed by each analyzer) is broadcast to facilitate the determination of the total CAT result. If the total CAT result is a success, the decisions made are forwarded to the actuators. Otherwise, the analyzers roll back and retry with an alternate interacting session. For every iteration of a task a watchdog timer is set. If a timeout occurs, an "emergency routine" is invoked.

Once it is invoked, normal operation is suspended and an attempt is made to stop the car in the safest and fastest way.

In principle, an alternate interacting session should be designed such that it may produce acceptable results for the cases where the primary interacting session fails to do so. Although designing efficient alternate interacting sessions is an open research topic and it is largely application-dependent, here we will briefly sketch a systematic approach which, we believe, is applicable to this type of applications. The major difference between the primary try blocks and the alternate try blocks in SUVS should be in the way the decisions are made. And those decisions are made based on several factors. For example, the speed analyzer makes a decision based on the current speed, RPM (revolutions per minute) of the engine, road condition (e.g., wet or dry), curve of the road, and existence of objects in front and if exists, characteristics (e.g., moving speeds) of the objects. Therefore, one way to design alternate try blocks is to apply the decision factors in different sequences.

Figure 15 shows two different approaches to designing the speed analyzer tasks. In Figure 15.a the road condition is first examined; then the curve status of the road is examined and so on. On the other hand, in Figure 15.b, the current speed is first examined; then the existence of an object is examined and so on. By doing so we can systematically produce multiple versions of software. Such produced versions are still considerably diverse in the detailed logics used and also have substantially different chances of encountering overflow/underflow conditions due to the diversity in the sequence of calculations used.

```
task primary speed analyzer;
    ....
    -- decision making routine of the primary try block
        { if road condition is dry
            => if the road is straight
                => if ...
                    ....
          else if road condition is wet of degree 1
            => if ...  }
```

Figure 15.a. The primary try block of the speed analyzer in SUVS.

```
task alternate speed analyzer;
    ....
    -- decision making routine of the alternate try block
        { if 0 < current speed <= 5 mph
            => if there is no object in front
                => if ...
                    ....
          else if 5 < current speed <= 10
            => if ... }
```

Figure 15.b. An alternate try block of the speed analyzer in SUVS.

A major portion of the CAT in the fault-tolerant SUVS is to check whether (1) the decisions are made based on correct information, (2) each decision made locally is reasonable with respect to both the recently observed condition of the car and the laws of physics, and (3) the decisions do not conflict with each other. The first part of the test (which is trivial in nature in comparison to the other two parts) can be facilitated by broadcasting the input data (which were used to make decisions) along with the preliminary decisions made. For example, the input data from the speedometer is initially received by the speed analyzer for every cycle of the task execution. This data is checked and then broadcast to other analyzers. The data may then be used by other analyzers in reaching certain preliminary decisions. Therefore, in the first part of the CAT the speed analyzer checks whether the speed data received from other analyzers together with their preliminary decisions are exactly the same as the original data that it received from the speedometer and has kept since. By doing so we can detect possible faults due to communication failure and/or memory failure.

The second and more important part of the CAT is largely to check if the preliminary local decision falls within a reasonable range. For example, if the preliminary decision on the acceleration to be made is beyond the capacity of the car, clearly a computation error can be suspected. The third part involves checking in each analyzer the possibility of conflict between the local decision and the preliminary decisions made by other analyzers. The decisions made by the analyzers are examined to see if there is any conflict among them. For example, as shown in Figure 16, decisions made by the speed analyzer and the direction analyzer should not conflict with each other.

```
task analyzer;
    ....
    -- conversation acceptance test (CAT) segment under the
    --  decentralized execution approach
       { if (attached data which were used to make decisions = original data)
            => if acceleration* = +2 and direction** = +3
                 => if (current speed < 20 mph) and (current RPM < 1000)
                      => if ....
                           => CAT is "pass"
                      => else if ...
                           => CAT is "fail"
         ....
         broadcast the result of the CAT-segment; }
       determine the total CAT result
```

* "acceleration", the output of the speed analyzer, is an integer value which indicates the acceleration ranged between -3 and +3. (Zero means no change.)
** "direction", the output of the direction analyzer, is an integer value which indicates the angle ranged between -9 and +9. (Zero means no change.)

Figure 16. A conversation acceptance test segment in SUVS.

6.2.6 Run-time support

One of the important run-time support functions in systems such as the fault-tolerant SUVS is to facilitate efficient and reliable communication among tasks. Since tasks run under tight synchronization, especially for CAT-related messages, message delay blocks the execution of other task(s). This may lead to the degradation of the system performance significantly. Furthermore, most messages are time-sensitive, i.e., the validity of a message depends on time. Therefore, protocols should be designed in such a way to support reliable and real-time communication. (For real-time communication the timing behavior of the protocol should be predictable.)

The system should also handle timeouts associated with conversations as well as those with protocols. The decision made by the analyzer becomes obsolete after a certain time period. Hence, absence of a report (CAT-segment result) from a participant within a timeout period during the CAT execution phase should be treated as the failure of the CAT. Finally, as mentioned in Section 4.7, an interrupt mechanism is required for fast initiation of the rollback of all participant tasks.

6.3 Discussion

In this section, an unmanned vehicle system was used to illustrate the factors to be considered in incorporating the conversation scheme into real-time control systems. The

centralized CAT approach is not suitable for this application because of its relatively high fault latency and high communication overhead characteristics. This means that if the ADT-Conversation structuring approach were to be used, decomposition of the CAT for decentralized or semi-centralized execution must be performed either by the program designer or by means of an automated tool. Also, since the entire control cycle is captured in one conversation, the synchronous exit approach is natural in this application. If the control cycle was implemented in the form of a series of conversations, then it might be possible to exploit the asynchronous exit approach in executing the conversations except the last one which is followed by actuator output actions. The approaches to implementation of the conversation scheme outlined in this section are believed to be applicable to many safety-critical applications. A preliminary version of a fault-tolerant SUVS testbed has been implemented on a workstation network at the investigators' institutes.

7. SUMMARY

This chapter presented several different approaches for implementing the conversation scheme in message-based DCS's. Important implementation factors such as the control of exits of processes upon completion of their conversation tasks and the approach to execution of the conversation acceptance test, were considered. A new efficient approach to run-time management of recovery information based on an extension of the recovery cache scheme was also proposed. Both exit control strategies, synchronous and asynchronous exits, and three different approaches to execution of CAT, centralized, decentralized, and semi-centralized, have been examined and compared in terms of system performance and implementation cost. Since each approach has merits and deficiencies, it is hard to say that one approach is simply better than another. Moreover, the implementation strategy should be carefully chosen based on the characteristics of the application system such as network topology, communication cost, etc.

However, it seems that the asynchronous exit approach is generally better than the synchronous exit approach. The former provides a higher performance during error-free execution than the latter. If the NLRB structuring approach is used, then the semi-centralized CAT approach or the decentralized CAT approach are more attractive than the centralized approach. On the other hand, if the ADT-Conversation structuring approach is used, then the selection of a good strategy for execution of a CAT depends on whether one can afford the effort required to decompose the CAT into distributed acceptance tests. If so, the semi-centralized or decentralized approach is more attractive and otherwise, the centralized CAT execution is the only choice.

Incorporation of the timeout capability into the conversation scheme is another area to be studied. The crash of a participant can result in the lockup of several other participants if the timeout mechanism is not used. Since participants enter the conversation asynchronously, the timeout period is an important design parameter, and an effective technique for determination of a proper timeout period needs to be developed. Integration of the conversation scheme and other established fault tolerance schemes [Bha87,Hec91,Toy87] is also an important area for future research.

Through analytic modeling studies, we have obtained some understanding of system performance behavior under various workload conditions. However, in order to obtain "real" data on implementation cost and system performance, experimental work and field experiences are necessary. Testbed-based evaluation [Chu87] of the proposed approaches in the contexts of additional real world applications (other than the unmanned vehicle system illustrated in this chapter) is regarded as a highly worthwhile research topic. Such efforts will also provide further insights into the types of applications for which the conversation scheme become a cost-effective approach to reliability enhancement.

8. REFERENCES

[And76] Anderson, T. and Kerr, R., "Recovery Blocks in Action: A System Supporting High Reliability", *Proc. 2nd Int'l Conf. on Software Engineering*, 1976, pp. 447-457.

[Bha87] Bhargava, B., editor, *'Concurrency and Reliability in Distributed Systems'*, Van Nostrand and Reinhold, 1987.

[Chu87] Chu, W.W., et al., "Testbed-Based Evaluation of Design Techniques for Fault-Tolerant Real-Time Distributed Computer Systems", *Proc. of the IEEE*, Vol. 75, No. 5, May 1987, pp.649-667.

[Cam83] Campbell, R.H., Anderson, T., and Randell, B., "Practical Fault Tolerant Software for Asynchronous Systems", *Proc. SAFECOM 83*, Cambridge, Oct. 1983, pp.59-65.

[Gre85] Gregory, S.T. and Knight, J.C., "A New Linguistic Approach to Backward Error Recovery", *Proc. FTCS-15*, 1985, pp.404-409.

[Hec91] Hecht, M., Agron, J., Hecht, H., and Kim, K.H.,, "A Distributed Fault Tolerant Architecture for Nuclear Reactor and Other Critical Process Control Applications", *Proc. IEEE Computer Society's 21st Symp. on Fault-Tolerant Computing*, Montreal, June 1991, pp.462-469.

[Hor74] Horning, J.J. et al, " A Program Structure for Error Detection and Recovery", *Lecture Notes in Comp. Sci.*, Vol. 16, Springer-Verlag, 1974, pp. 171-187.

[Kim76] Kim, K.H., Russell, D.L., and Jenson, M.J., "Language Tools for Fault-Tolerant Programming", *PETP-1*, Electronic Sciences Lab., USC, Nov. 1976.

[Kim82] Kim, K.H., "Approaches to Mechanization of the Conversation Scheme Based on Monitor", *IEEE Trans. on Software Eng.*, Vol. SE-8, No. 3, May 1982, pp.189-197.

[Kim85] Kim, K.H., Yang, S.M., and Kim, M.H., "Implementation of Concurrent Programming Language Facilities Supporting Conversation Structuring", Proc. *COMPSAC 85*, Oct. 1985, pp.445-453.

[Kim89] Kim, K.H. and Yang, S.M., "Performance Impacts of Lookahead Execution in the Conversation Scheme", *IEEE Trans. on Computers*, Vol. 38, No. 8, August 1989, pp.1188-1202.

[Kol80] Kolstad, R.B. and Campbell, R.H., *Path Pascal User Manual*, Univ.of Illinois at Champaign-Urbana, Jan. 1980.

[Man89] Mancini, L.V., and Shrivastava, S.K., "Replication within Atomic Actions and Conversations: A Case Study in Fault-Tolerance Duality", *Proc. FTCS-19*, 1989, pp.454-461.

[Oza88] Ozake, B.M., Fernandez, E.B., and Gudes, E., "Software Fault Tolerance in Architectures with Hierachical Protection Levels", *IEEE Micro*, Vol. 8, Aug. 1988, pp.30-43.

[Ran75] Randell, B., "System Structure for Software Fault Tolerance", *IEEE Trans. on Software Eng.*, June 1975, pp.220-232.

[Rus79] Russell, D.L. and Tiedeman, M.J., "Multiprocess Recovery using Conversations", *Proc. FTCS-9*, 1979, pp.106-109.

[Shi87] Shirivastava, S.K., Mancini, L., and Randell, B., "On the Duality of Fault Tolerant System Structures", *Tech. Memo. SRM/455*, Computing Lab., Univ. of Newcastle upon Tyne, 1987.

[Toy87] Toy, W.N., "Fault-Tolerant Computing", A chapter in *Advances in Computers*, Vol. 26, Academic Press, 1987, pp.201-279.

[Tyr86] Tyrrell, A.M., and Holding, D.J., "Design of Reliable Software in Distributed Systems Using The Conversation Scheme", *IEEE Trans. on Software Eng.*, Vol. SE-12, No. 9, Sept. 1986, pp.921-928.

[Wu89] Wu, J. and Fernandez, E.B., "A Simplification of a Conversation Design Using Petri Nets", *IEEE Trans. on Software Engineering*, Vol. 15, No. 5, May 1989, pp.658-660.

CHAPTER IV

PERFORMANCE IMPACTS OF LOOK-AHEAD EXECUTION IN THE CONVERSATION SCHEME

1. INTRODUCTION

As the use of distributed computer systems in safety-critical applications has steadily increased in recent years, the design of fault handling capabilities into the concurrent programs running on the distributed hardware has become a subject of serious interest to system designers [Bha87,Dav80,Fou84,Hec76,McD82,Ram81,Toy87]. The conversation scheme proposed by Randell in [Ran75] is one of the fundamental approaches to structured design of such fault-tolerant concurrent programs. As discussed in [Shr87] the scheme provides a means of facilitating failure atomicity and backward recovery in cooperating process systems in a manner analogous to that of the atomic action mechanism in object-based systems. On the basis of the abstract notion of the conversation in [Ran75], several concrete structuring approaches and supporting tools have been developed as described in [Cam83,Gre85,Kim82, Kim85,Oza88]. However, their utilities have not been fully tested and not much is known about the performance characteristics of the conversation scheme.

One of the costs of using the conversation scheme is the execution time increase due to the tight synchronization imposed among participant processes and due to the execution of the conversation acceptance test. This is an overhead. Another cost of interest to system designers is the recovery time, i.e., the time spent for recovering from detected faults. These time costs were analyzed in [Kim86] by use of a queueing network model. The results showed among other things that under practical circumstances the system performance is significantly affected by the synchronization required of the processes in exit from a conversation, not by the probability of acceptance test failure.

A fundamental approach to reducing the synchronization overhead is the lookahead. That is, by allowing the participant processes which complete their conversation activities including acceptance tests earlier than other participants to exit from the conversation and continue processing rather than to wait until all the participants have passed their acceptance tests, substantial reduction of the synchronization overhead can be achieved. When the participants exit from a conversation via a lookahead, they should maintain the recovery points established on their entries to the conversation. This is because other participants may later fail in their acceptance tests in which case all the participants must be brought back to the recovery line (i.e., the entry points of the conversation) for retry. Therefore, the recovery costs may increase when the lookahead is used. The possibility of incorporating the lookahead capability into the conversation scheme was discussed in [Kim76,Rus79,Yan86]. In this chapter, we analyze the impacts of the lookahead approach on the performance of the conversation scheme.

Our interest here is in studying the inherent overhead, i.e., the overhead due to acceptance test and synchronized exit, not the scheduling overhead due to limitations in the available processors. Therefore, we consider the cases of using multiprocessor systems or tightly coupled networks of computers in which each processor is dedicated to running a single process. This actually matches with the current trend in use of multi-microcomputer systems in real-time applications. The queueing network model developed in [Kim86] for the case of the basic conversation scheme without the lookahead capability is extended in this chapter to cover the cases with the lookahead capability. One attractive feature of this queueing network based analytic evaluation is the feasibility of covering a broad range of situations. The specific performance indicators analyzed include the system throughput, the average number of processors idling inside a conversation due to the synchronization, and the average time spent in a conversation. Comparison of the performance in the two cases, i.e., the case of synchronous exit without lookahead and the case of using lookahead, reveals that the lookahead approach offers potential for significant reduction of the execution overhead of the conversation scheme. In the next section a brief review of the basic conversation scheme is given together with an introduction of the lookahead approach. Section 3 then discusses the execution environment considered in this chapter and the queueing network models developed for both the basic conversation scheme and the scheme extended with the lookahead. The models are used in Section 4 to evaluate and compare the system performance under different execution approaches and workloads. Section 5 is the summary section.

2. BASIC CONVERSATION STRUCTURE AND LOOKAHEAD

The conversation is a two-dimensional enclosure of recoverable activities of multiple interacting processes, in short, recoverable interacting session [Kim82,Ran75]. As depicted in Figure 1 it creates a "boundary" which process interactions may not cross. The boundary of a conversation consists of a recovery line, a test line, and the walls defining the membership. Each participant process contains one or more try blocks designed to produce the same or similar computation results as well as an acceptance test which is a logical expression representing the criterion for determining the acceptability of the execution results of the try blocks. A recovery line is a coordinated set of the recovery points of interacting processes that are established (possibly at different times) before interactions begin. A test line is a correlated set of test points at which computation results of interacting processes are checked out via execution of acceptance tests. The correlated set of acceptance tests used at a test line may be viewed as a single global acceptance test called a conversation acceptance test. A conversation

is successful only if all the interacting processes pass their acceptance tests at the test line. If any of the acceptance tests fails due to a residual design error in the try block used, a hardware malfunction, a timeout enforced by a watchdog timer, etc., all the processes roll back to the recovery line and retry with their alternate try blocks. These alternate try blocks collectively define an alternate interacting session (AIS) (and may be viewed as an alternate interaction block (AIB)), whereas the set of primary try blocks executed first after the processes enter the conversation define the primary interacting session (PIS) (and may be viewed as the primary interaction block (PIB)).

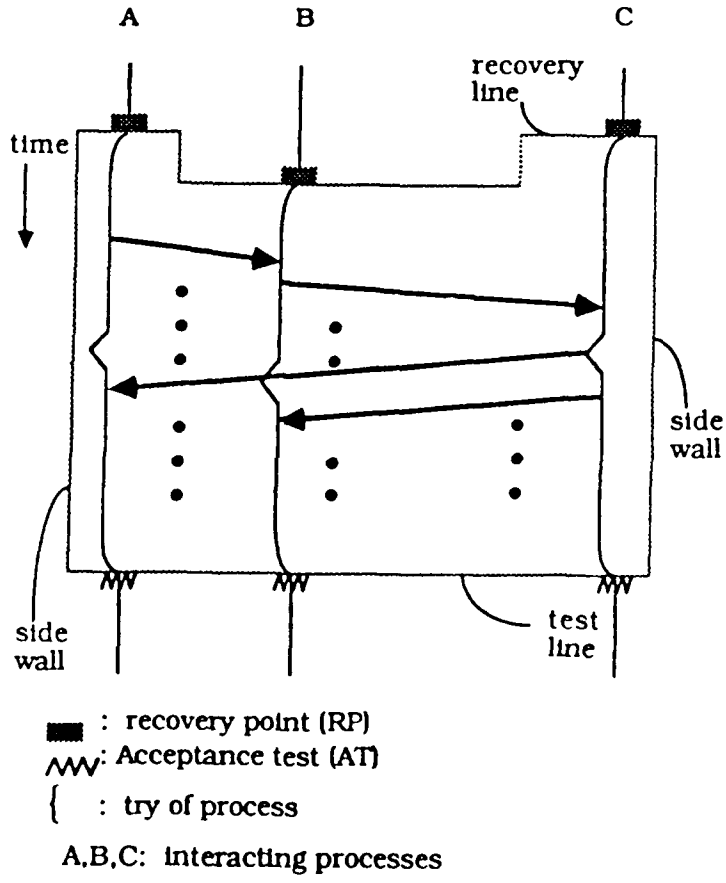


Figure 1. Abstract Conversation Structure.

A process which is inside a conversation cannot interact with a process which is not in the conversation. Conversations must be strictly nested in two dimensions. That is, when conversation C.nest is nested within conversation C, the set of processes that participate in nested conversation C.nest must be a subset of the processes that participate in C, the entire recovery line of C.nest must be established after the entire recovery line of C, and the entire test line of C.nest must be set before the entire test line of C.

In the basic conversation scheme sketched above, processes enter a conversation asynchronously but synchronize themselves before exiting from it. The synchronization considered here is of a special kind specifically required by the conversation scheme and thus different from the application- dependent synchronization required between cooperating processes. As mentioned in the preceding section, the synchronization can add significantly to the time cost of the conversation scheme. The lookahead is a fundamental way of reducing this synchronization overhead. Under the lookahead approach each participant process leaves the conversation as soon as it passes its own acceptance test. It does so with the awareness of the possibility that another participant may execute an acceptance test later and fail in the acceptance test, thus making it necessary for the former to roll back to the recovery line of the conversation. Therefore, the lookahead approach here is an optimistic approach and is aimed at trading increase in recovery costs for reduction of synchronization overhead.

A process that has exited from a conversation C1 via lookahead may enter another conversation C2. If some participants of C1 are not participants of C2, then it is possible that C2 activities including acceptance tests are completed while C1 remains unfinished. In such a case, C2 should be treated as an unfinished conversation until C1 becomes completed. This is because if C1 fails, then all C2 activities that have taken place must be nullified as a part of the rollback to the recovery line of C1.

Although it is logically feasible to make provisions for a participant process to look ahead of the unfinished conversation to an unlimited extent, it is useful to limit the extent of the lookahead with respect to controlling the implementation complexity. In most practical applications, there are natural limits to the extents of lookahead possible; when a process progressing past the test lines of one or more unfinished conversations reaches a point where it needs to interact with other slowly following processes, it has to wait for the other processes to become ready for interaction. In addition, the lookahead should not be allowed to go past the points where critical irreversible actions, e.g., certain output actions, are taken. In the next section, the cases where the lookahead is allowed to limited extents are considered.

3. THE EXECUTION ENVIRONMENT ASSUMED AND QUEUEING NETWORK MODELS

The characteristics of the hardware and the software of the systems considered are described first. Queueing network models are then developed in Section 3.3.

3.1 Hardware characteristics

The type of systems considered in this chapter is depicted in Figure 2. A system consists of multiple, say N, computing nodes. Each node is equipped with a processor, a local memory, and a communication interface. The inter- node connection medium is either a high-speed bus or a common memory shared among multiple nodes. We assume that inter-node signaling is accomplished via interrupts or single-byte messages. Consequently, the communication overhead is negligible.

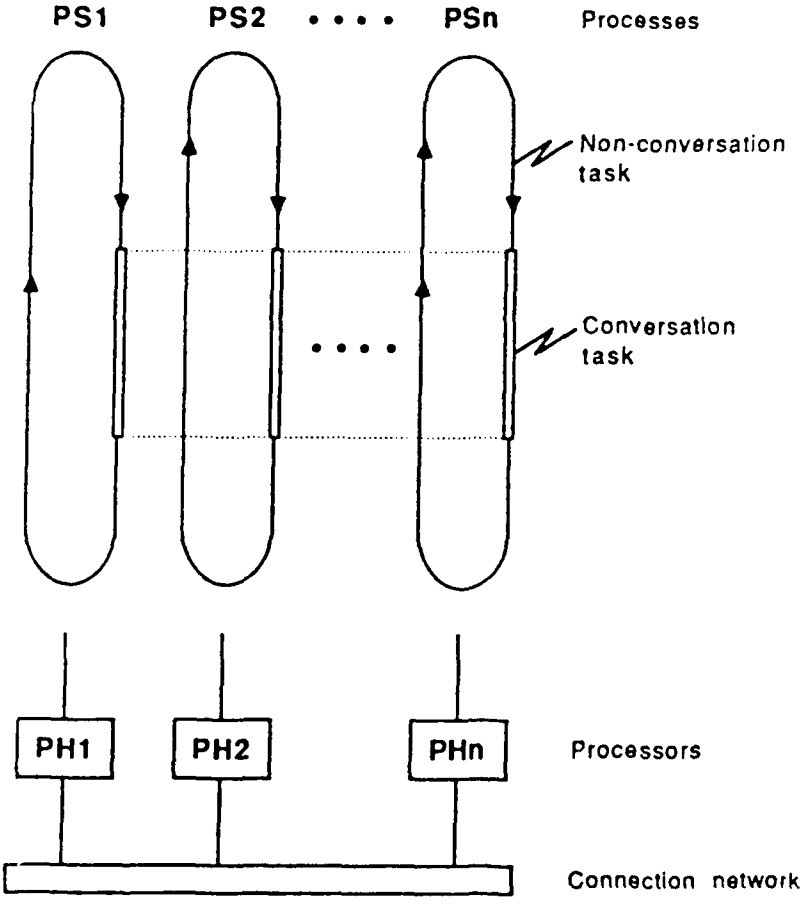


Figure 2. A Model of the Fault-Tolerant Systems in parallel Execution of Conversation

3.2 Software characteristics

As shown in Figure 2, each node in the systems runs a single process which is a non-terminating cyclic program in execution. Each process cycle consists of two steps: a process step inside a conversation called a conversation task, and a process step outside a conversation called a non- conversation task. Each process alternates between two tasks and in every cycle

all the processes participate in the same conversation. Therefore, we are dealing with systems in parallel execution of conversations.

A conversation is successful only if all the participant processes pass their acceptance tests. If any of the acceptance tests fails, all the processes roll back to the recovery line and retry with their alternate try blocks. It is assumed that the conversation construct contains one primary interaction block and an infinite number of alternate interaction blocks. This means that every conversation succeeds eventually.

If a process fails its acceptance test which is a part of conversation CONV, it broadcasts the failure message to other participant processes. Upon receiving a failure message, the processes which have been executing their conversation tasks belonging to CONV abandon the tasks (including acceptance tests) and join the group of failed processes. The processes which have already completed their conversation tasks belonging to CONV also join the group of failed processes. On the other hand, the processes which have not entered CONV and have been executing their non-conversation tasks will complete the tasks and then immediately after entering CONV, they will also join the group of failed processes. After all the participants join the failed group, they retry with their alternate try blocks.

In this modeling study, the conversations nested within other conversations were not explicitly dealt with for the sake of simplicity. The following is a summary of the software characteristics assumed in this chapter. (1) Each process alternates between its conversation task and non- conversation task.
(2) All processes participate in the same conversation in every cycle.
(3) Each process has an unlimited number of alternate try blocks.
(4) Once a process fails its acceptance test, all other processes that have not finished the conversation tasks abandon the conversation tasks.
(5) There are no conversations nested within other conversations.

The lookahead capability is incorporated into the system in Figure 2 with the scope of lookahead limited. The lookahead scope is defined here in terms of the active conversations that a process can participate in and leave from via lookahead. In the simplest case, a process is allowed to make a "single conversation lookahead", which means that a process can continue lookahead as long as it has not exited from more than one unfinished conversations. In Figure [illegible] for example, assume that process B completes its conversation task (B2) in CONV1 while process A is still executing CONV1. Process B can leave CONV1 because there are no other

active conversations which it exited from. Assume further that process B completes its task (B4) in conversation CONV2 while process A is still executing CONV1. Now process B cannot leave CONV2 because CONV1 has not been completely validated. Therefore, under the single conversation lookahead rule a process is not allowed to leave a conversation if there is an earlier initiated but unfinished conversation that the process has participated in.

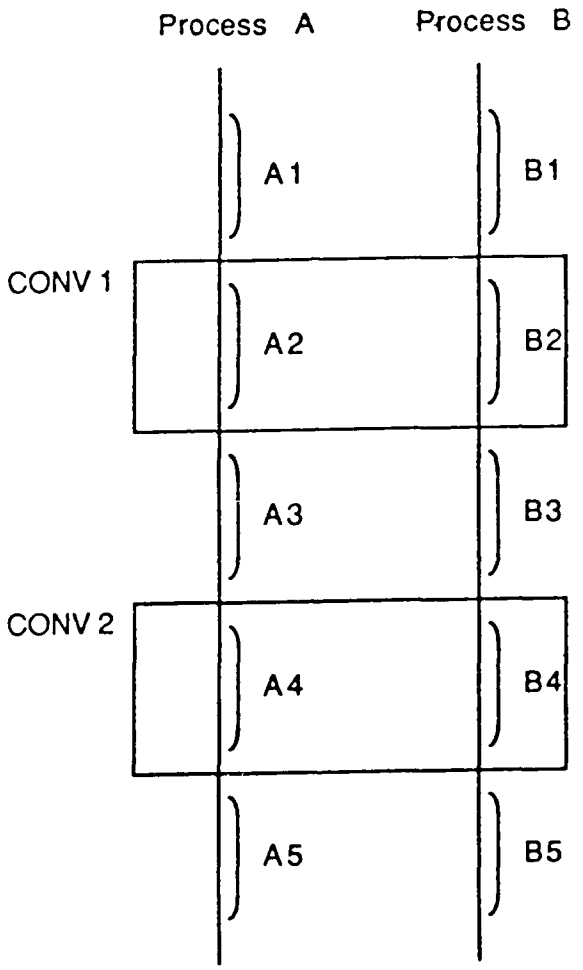


Figure 3. An Example of Conversation Structure.

Similarly, the two-conversations lookahead rule can be defined. In this case a process is not allowed to leave a conversation if the conversation is currently the third unfinished conversation that the process has participated in.

For convenience, the basic conversation scheme without the lookahead capability is denoted by CL-0 in the rest of this chapter whereas the conversation scheme operating under the single conversation lookahead rule and the scheme under the two-conversations lookahead

rule are denoted by CL-1 and CL-2, respectively.

3.3 Queueing network models

(1) Model for CL-0 (zero lookahead)

A queueing network model of the system shown in Figure 2 operating under CL-0 is depicted in Figure 4. Servers in the model represent processors and customers represent processes which alternate between two different tasks.

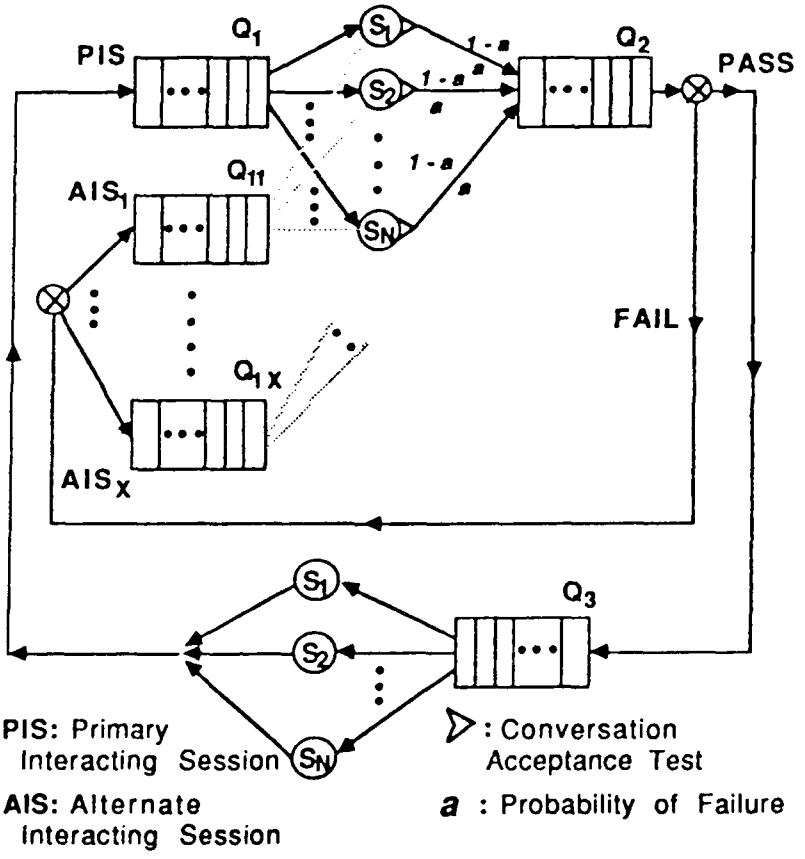


Figure 4. A Queueing Network Model of the System in Figure 2.

Each process moves among three different states: executing a conversation task, waiting for other processes to complete the conversation tasks, and executing a non-conversation task. In Figure 4, the customers in Q1 represent the processes in execution of conversation tasks whereas the customers in Q3 represent the processes in execution of non-

conversation tasks. The customers in Q2 represent the processes which have finished their conversation tasks and are waiting for others to finish. As soon as Q2 becomes full, all the processes in the queue move to either Q3 or Q1i, $1 \le i < \infty$, depending upon whether any of the processes has failed in its acceptance test or not. If they all have passed their acceptance tests, then they enter Q3. Otherwise, they all enter Q1i, $1 \le i < \infty$, the queue for their next alternate interacting session. Since Q3 is empty during the retry, there is no loss of information caused by merging all Q1i, $1 \le i < \infty$, into Q1. Figure 5 depicts a more abstract representation of the queueing network model depicted in Figure 4. Therefore, N processes, where N is the total number of processes in the system, are distributed over three different queues.

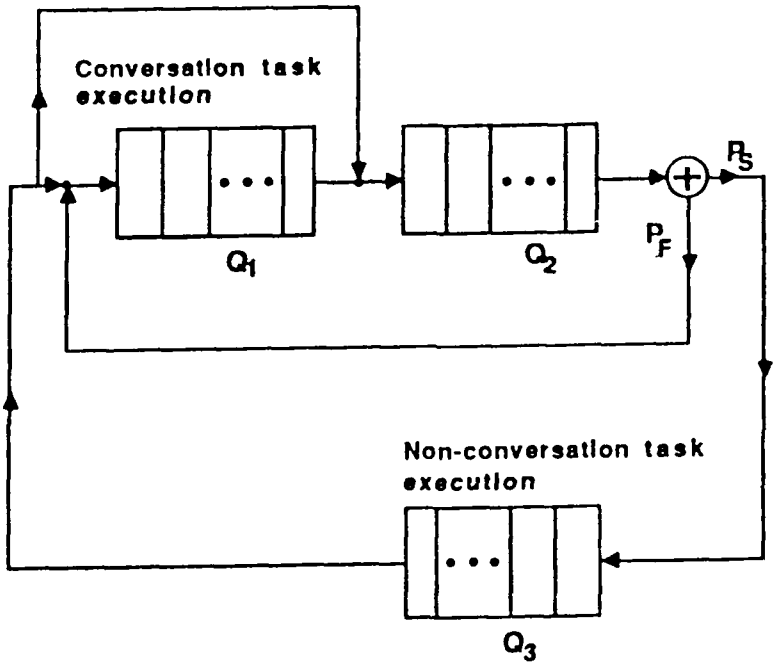


Figure 5. An Abstract Representation of the Queueing Network Model in Figure 4.

Since each process runs on a dedicated processor, no waiting time is needed for the process to execute its task. This characteristic is correctly represented in the queueing network model by the number of Q1 servers as well as the number of Q3 servers being always equal to N. Therefore, the execution time of a conversation task is represented by the length of stay in Q1 whereas the execution time of a non-conversation task is represented by the length of stay in Q3.

Let (n1,n2,n3) represent a state of the queueing network in which n1 processes are in Q1, n2 processes in Q2, and n3 processes in Q3, where N=n1+n2+n3. P(n1,n2,n3) denotes

the stationary probability of the state. However, this does not represent all possible states since (n1,n2,n3) does not indicate anything about whether any of the processes in Q2 has failed its acceptance test or not. If any of the processes in Q1 fails its acceptance test then all others in Q1 should immediately abandon their conversation tasks and enter Q2. From then on, a process which has been in Q3 proceeds directly to enter Q2 without executing its conversation task in Q1 as it leaves Q3 and enters the conversation. This is shown as a bypass around Q1 in Figure 5. When Q2 becomes full, all the processes return to Q1. Let (e,n2,n3) represent an error state in which n2 processes are in Q2, n3 processes in Q3, where N=n2+n3, and at least one of the processes in Q2 has failed its acceptance test. P(e,n2,n3) denotes the stationary probability of the state. As discussed above, when the network is in an error state (e,n2,n3), Q1 is empty. The following assumptions are also an integral part of the queueing network model.

A1: The execution times of the conversation tasks and those of the non- conversation tasks are exponentially distributed with means 1/u1 and 1/u3, respectively.

A2: The probability of failure in an acceptance test for each process is a.

The transitions of the six-process system among all possible states are depicted in Figure 6.

(2) Model for CL-1 (one-conversation lookahead)

The queueing network model developed above for the system operating under CL-0 can be extended as depicted in Figure 7 to represent the system operating under CL-1. This queueing model contains two sets of three queues, i.e., queue sets (Q1,Q2,Q3) and (Q1',Q2',Q3'). The three queues in each set correspond to the three queues in Figure 5. The customers in Q1 and Q1' represent the processes in execution of conversation tasks whereas the customers in Q3 and Q3' represent the processes in execution of non- conversation tasks. The customers in Q2 and Q2' represent the processes which have finished their conver sation tasks and are waiting for others to finish.

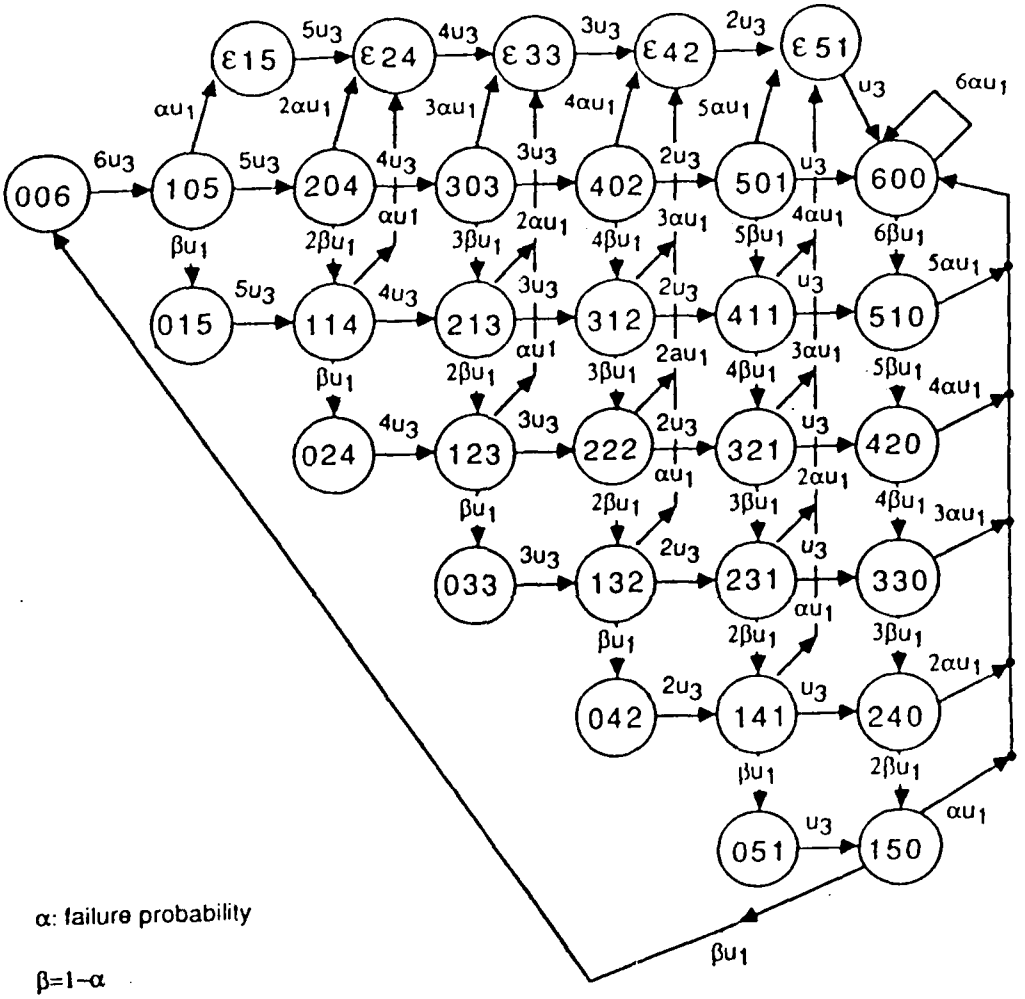


Figure 6. State Transition Diagram for the Six-process System Operating under CL-0.

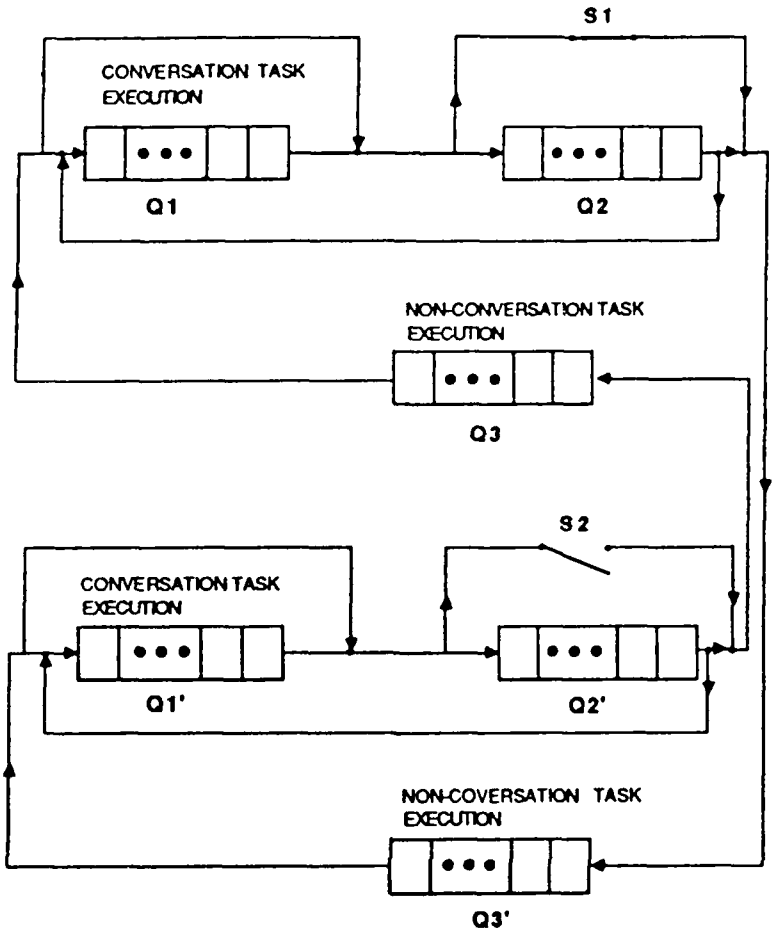


Figure 7. A Queueing Network Model for CL-1.

Therefore, this new queueing model looks similar to the model of the system in which processes alternate between two types of conversations. The only difference is that there are bypasses around Q2 and Q2' with on-off switches, S1 and S2, respectively. In a sense, the bypasses are lookahead paths and the switches enable/disable lookahead. The lookahead switches are opened and closed alternatively, i.e., if S1 is open, S2 is closed and vice versa. The role of these switches is to allow the process to make only one conversation lookahead.

For example, assume that processes A and B in Figure 3 start execution of A1 and B1, respectively, from Q3 in Figure 7. Initially, switch S1 is closed (i.e., a process which has passed its acceptance test can go ahead through S1 instead of waiting in Q2) and switch S2 is open. At two different times the processes enter conversation CONV1 and thus move from Q3 to Q1. Now process B completes its conversation task B2 and passes its acceptance test while process A is still executing its conversation task A2. Under CL-0 process B should wait in Q2

until process A completes its conversation task. Under CL-1 however, process B skips Q2 (since S1 is closed) and proceeds into Q3' to execute B3. If process B completes B4 in CONV2 and passes its acceptance test, then one of the following cases will arise.

Case 1: Process A is in Q1, i.e., still executes A2. In this case, process B should wait in Q2' until process A completes A2 because only one-conversation lookahead is allowed. The following two cases are possible later:

Case 1.1: Process A successfully completes CONV1. Process A enters Q3'. Now the status of the lookahead switches becomes reversed, i.e., S1 is open and S2 is closed. Therefore, process B moves from Q2' to Q3 and executes B5 immediately. (Now processes in the upper queue set (Q1,Q2,Q3) are ahead of processes in the lower queue set (Q1',Q2',Q3').)

Case 1.2: Process A fails the acceptance test of CONV1. Both processes roll back to Q1 and execute their alternate try blocks of CONV1. (All the lookahead executions done by process B become nullified.)

Case 2: Process A has successfully completed CONV1 and already left Q1. The status of two switches was changed when process A left Q1 and entered Q3'. In this case, process B moves to Q3 without waiting in Q2' because the lookahead path is available.

Let (n1,n2,n3,n1',n2',n3') represent a state of the queueing network in which n1, n2, n3, n1', n2', and n3' processes are in Q1, Q2, Q3, Q1', Q2', and Q3', respectively, where N=n1+n2+n3+n1'+n2'+n3' and N is the number of customers (i.e., processes) moving through the network. Figure 8 depicts the transition of the network among states when N=2. The steady state behavior of this network is discussed in Section 4.

(3) Model for CL-2 (two-conversations lookahead)

A further extension of the model in Figure 7 to represent the system operation under CL-2 results in the model depicted in Figure 9. This queueing model contains three sets of three queues, i.e., queue sets (Q1,Q2,Q3), correspond to the three queues in Figure 5. There are three lookahead switches, S1, S2, and S3, and only one switch is open at a time. Therefore, processes are allowed to make two-conversations lookahead.

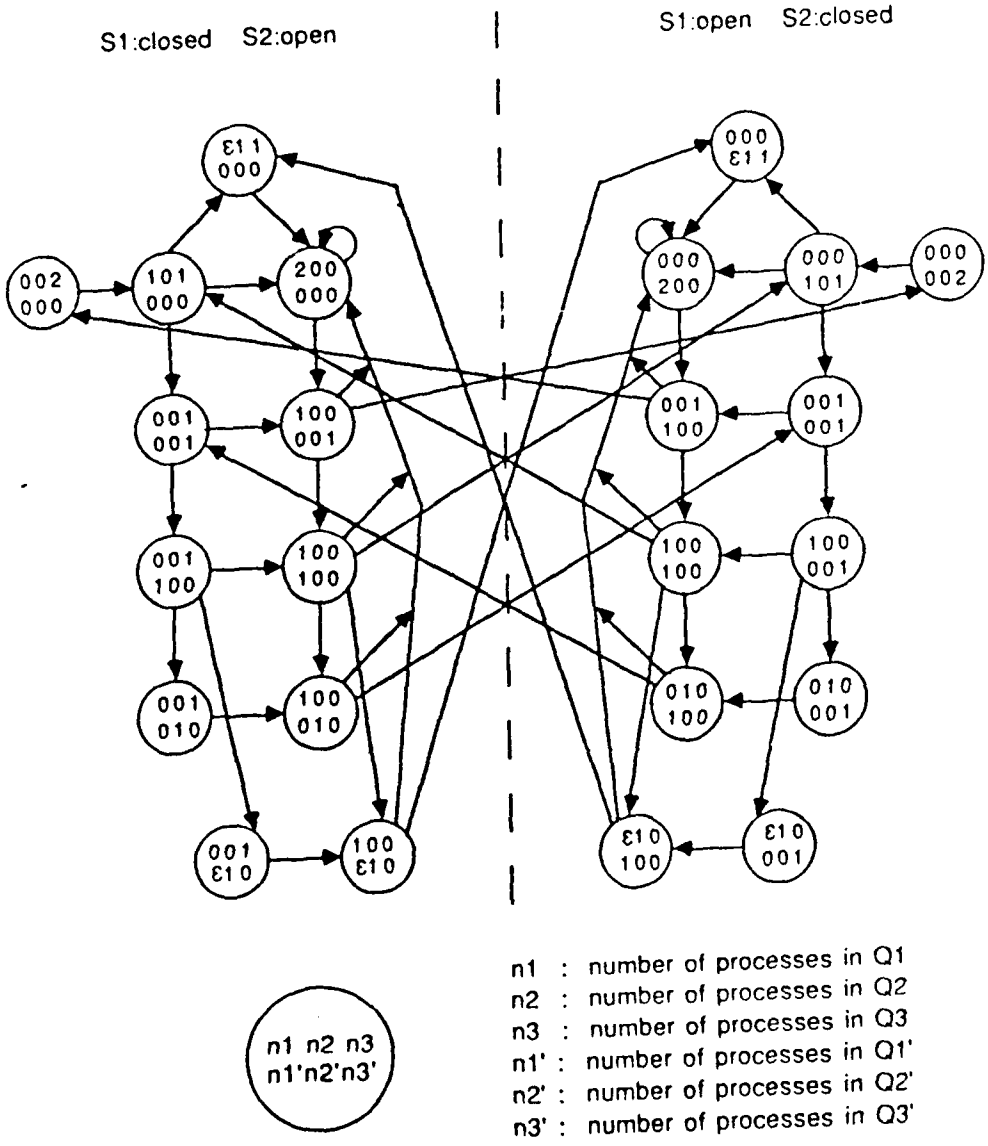


Figure 8. State Transition Diagram for the Two-process System Operating under CL-1.

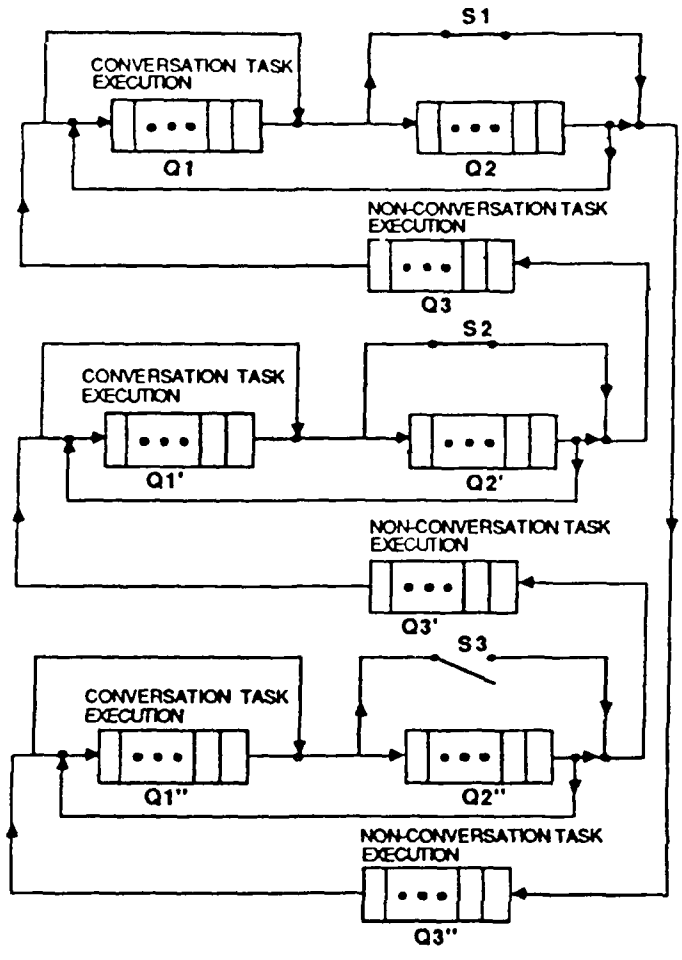


Figure 9. A Queueing Network Model for CL-2.

The state of this queueing network can be characterized by nine parameters, each representing the number of processes in a queue. The steady-state behavior of this network is discussed in Section 4.

3.4 Analysis of the queueing network models

The steady state balance equations for all the states of the queueing network models formulated in the preceding section (3.3) are included in Appendix A. From the equations, the stationary probability of each state can be numerically obtained. In the next section, the performance of the system in parallel execution of conversations is analyzed by making use of such values.

The complexity of the analysis process grows rapidly as the lookahead scope expands. A good indicator of this complexity is the number of states which a given system can be in. The exact number of all possible states of a system can be derived by use of the following formulae. Here n represents the total number of processes in the system.

(1) CL-0

a) Non-error states: $(n^2+3n)/2$

b) Error states: n-1

c) Total: $(n^2+5n-2)/2$

(2) CL-1

a) Non-error states: $(1/24)*(n^4+10n^3+23n^2+14n)$

b) Error states: $(1/6)*(n^3+3n^2+2n-6)$

c) Total: $(1/24)*(n^4+14n^3+35n^2+22n-24)$

	CL-0	CL-1	CL-2
2 processes	6	12	18
6 processes	32	237	965
12 processes	101	2092	21111

Figure 10. The Numbers of Queueing Network Models.

(3) CL-2

a) Non-error states: $(1/720)*(n^6+21n^5+145n^4+435n^3+574n^2+264n)$

b) Error states: $(1/120)*(n^5+10n^4+35n^3+50n^2+24n-120)$

c) Total:$(1/720)*(n^6+27n^5+205n^4+645n^3+874n^2+408n-720)$

Figure 10 summarizes the numbers of possible states for three different sizes of systems operating under three different execution schemes, CL-0, CL-1, and CL-2.

4. PERFORMANCE COMPARISON

In this section various system performance indicators obtained through the analysis of the steady-state behavior of the models developed in section 3.3, are discussed. Among the several performance indicators, the following are considered to be the most interesting ones:
(1) System throughput evaluated in terms of the number of successful conversations per unit time,
(2) Resource utilization evaluated in terms of the number of processors idling due to the synchronization required inside the conversation, and
(3) Conversation participation time, i.e., the average amount of time a process spends inside each conversation.

4.1 System throughput

The system throughput, TPc, indicated by the number of successful conversations per unit time, is obtained by use of the following formulae.

(1) Under CL-0

$TPc(0) = (1-\alpha)*u1*P(1,N-1,0)$,

where $(1-\alpha)$ represents the probability that each process passes its acceptance test, and u1 represents the completion rate for the conversation task.

(2) Under CL-1

$$TPc(0) = \sum_{n1'+n2+n3'=N-1} (1-\alpha)*u1*P(1,0,0,n1',n2',n3')$$

$$+ \sum_{n2'+n3'=N-1} (1-\alpha)*u1*P(1,0,0,\varepsilon,n2',n3')$$

(3) Under CL-2

$$TPc(2)= \sum_{n1+n3'+n1''+n2''+n3''=N-1} (1-\alpha)*u1*P(n1,0,n3,1,0,0,n1'',n2'',n3'')$$

$$+ \sum_{n1+n3+n2''+n3''=N-1} (1-\alpha)*u1*P(n1,0,n3,1,0,0,\varepsilon,n2'',n3'')$$

$$+ \sum_{n2+n3=N-1} (1-\alpha)*u1*P(\varepsilon,n2,n3,1,0,0,0,0,0)$$

Figures 11 and 12 depict the system throughputs under both CL-0 and CL-1 for the cases of the number of processes being two and twelve, respectively. The execution time ratio between the non-conversation task and the conversation task (u1/u3) varies from 0.1 to 10. The wide range of values for u1/u3 is examined because the ratio u1/u3 may actually vary widely among different applications. Since the execution time of the non-conversation task is fixed to 0.1 time unit, the execution time of the conversation task varies from 1 to 0.01 time unit. Each figure depicts for each conversation scheme (CL-0 or CL-1) four different curves corresponding to four different probabilities of acceptance test failure. Here we are interested only in those practical cases where the failure probability is less than 0.05.

Both figures show that under CL-0 the system throughput is not much affected by the probability of the acceptance test failure if there are a relatively small number of processes. However, the system throughput is more sensitive to the acceptance test failure probability under CL-1 than under CL-0. For example, when u1/u3=10 and the failure probability increases from zero to 0.05, the system throughput for the case of twelve processes decreases by 0.1 under CL-0 whereas it decreases by 1.0 under CL-1 (Figure 12). This is a reflection of the higher recovery cost under CL-1.

On the other hand, comparison of Figure 11 and Figure 12 reveals that as the number of conversation participants increases, the system throughput degrades more slowly under CL-1. For example, as the number of participants increases from two to twelve while u1/u3=10, the throughput degrades from 6.2 to 3.2 (i.e., 49% reduction) under CL-0 whereas the throughput degrades from 7.5 to 4.9 (i.e., 35% reduction) under CL-1. This also means that the benefits of lookahead are greater when the number of participants is larger. Figure 13

shows this phenomenon from another perspective. As long as the acceptance test failure probability is within a practical range, i.e., $0 < \alpha << 0.05$, the throughput increase resulting from incorporation of the single conversation lookahead rule is greater when the number of participants is larger.

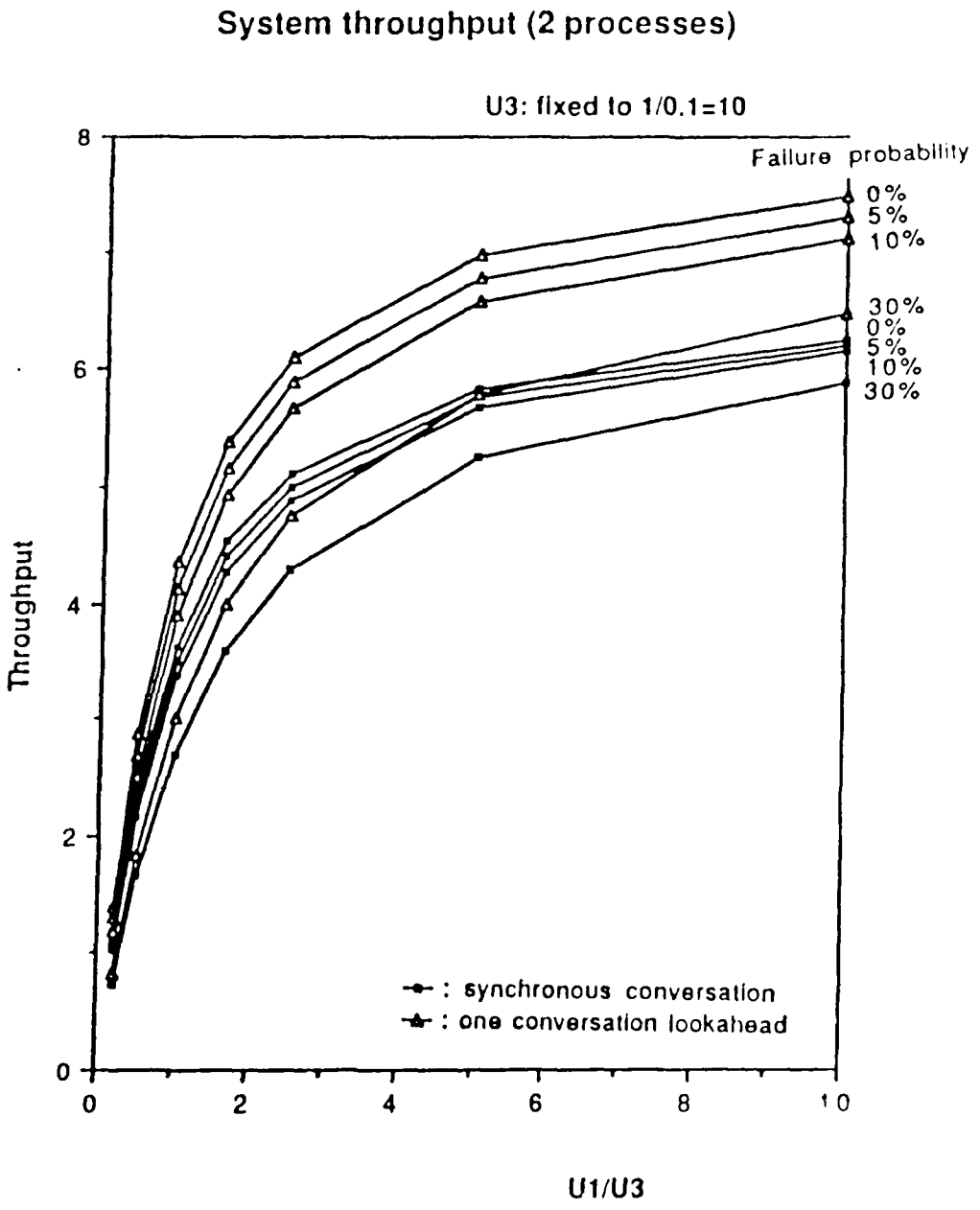


Figure 11. System Throughput under CL-0 and CL-1 (2 processes)

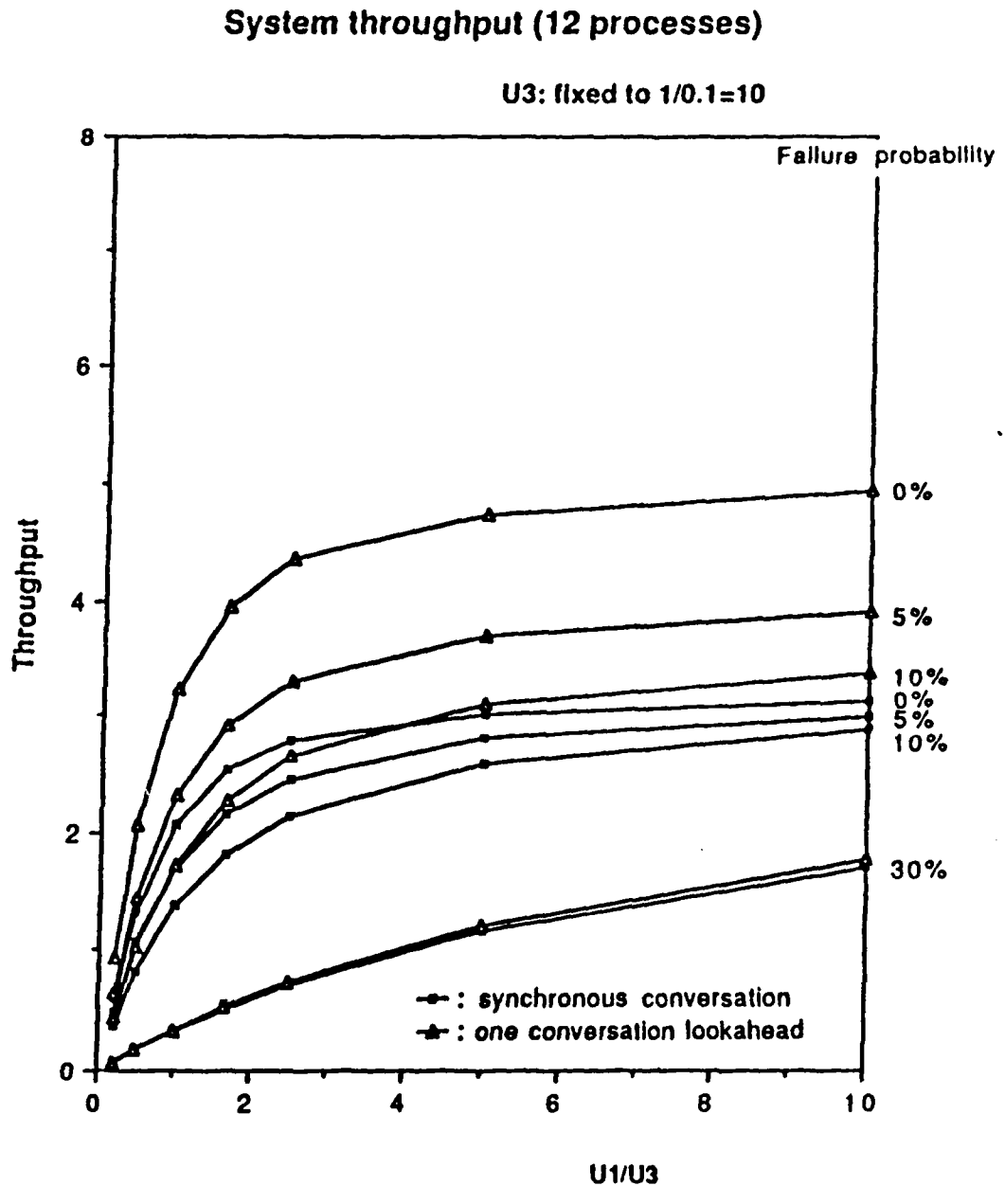


Figure 12. System Throughput under CL-0 and CL-1 (12 processes).

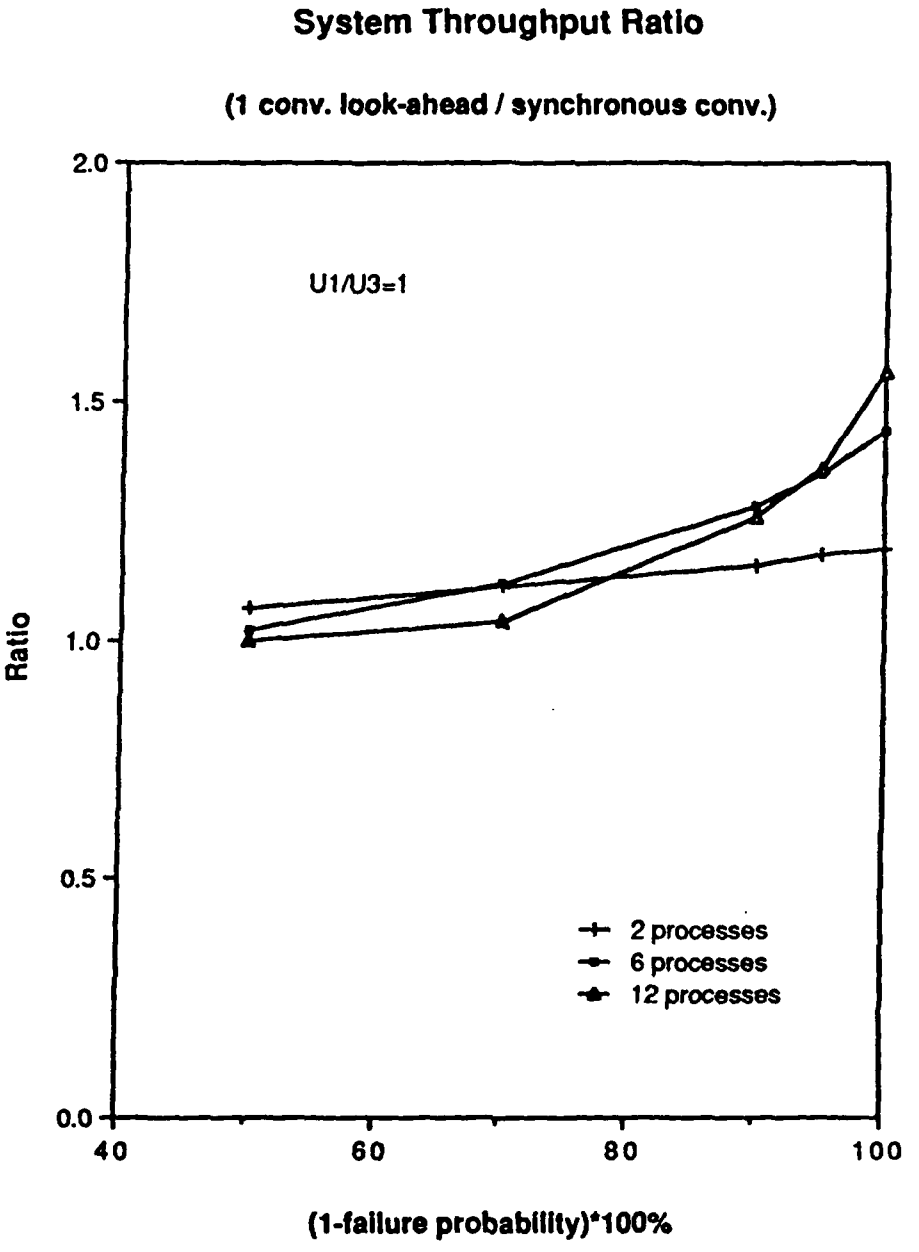


Figure 13. System Throughput Ratio.

Figure 14 shows the increase of system throughput resulting from the change of the scheme from CL-0 to CL-1 and then to CL-2. Two cases of the acceptance test failure probability, i.e., zero and 0.05, are displayed and u1/u3 is 10. The figure shows that when the failure probability is as large as 0.05, the benefits of changing from CL-1 to CL-2 are not substantial, especially if the number of participants is six or more. On the other hand, when

the failure probability is closer to zero, the benefits are substantial. For example, when the number of participants is six and the failure probability is zero, the benefit of changing from CL-0 to CL-1 is 46% increase in throughput whereas the benefit of changing from CL-0 to CL-2 is 69% increase in throughput. Therefore, CL-2 brings 23% additional increase in throughput in this case. The schemes permitting lookahead of three or more conversations can be analyzed in similar ways to support decisions regarding their adoption in given execution environments.

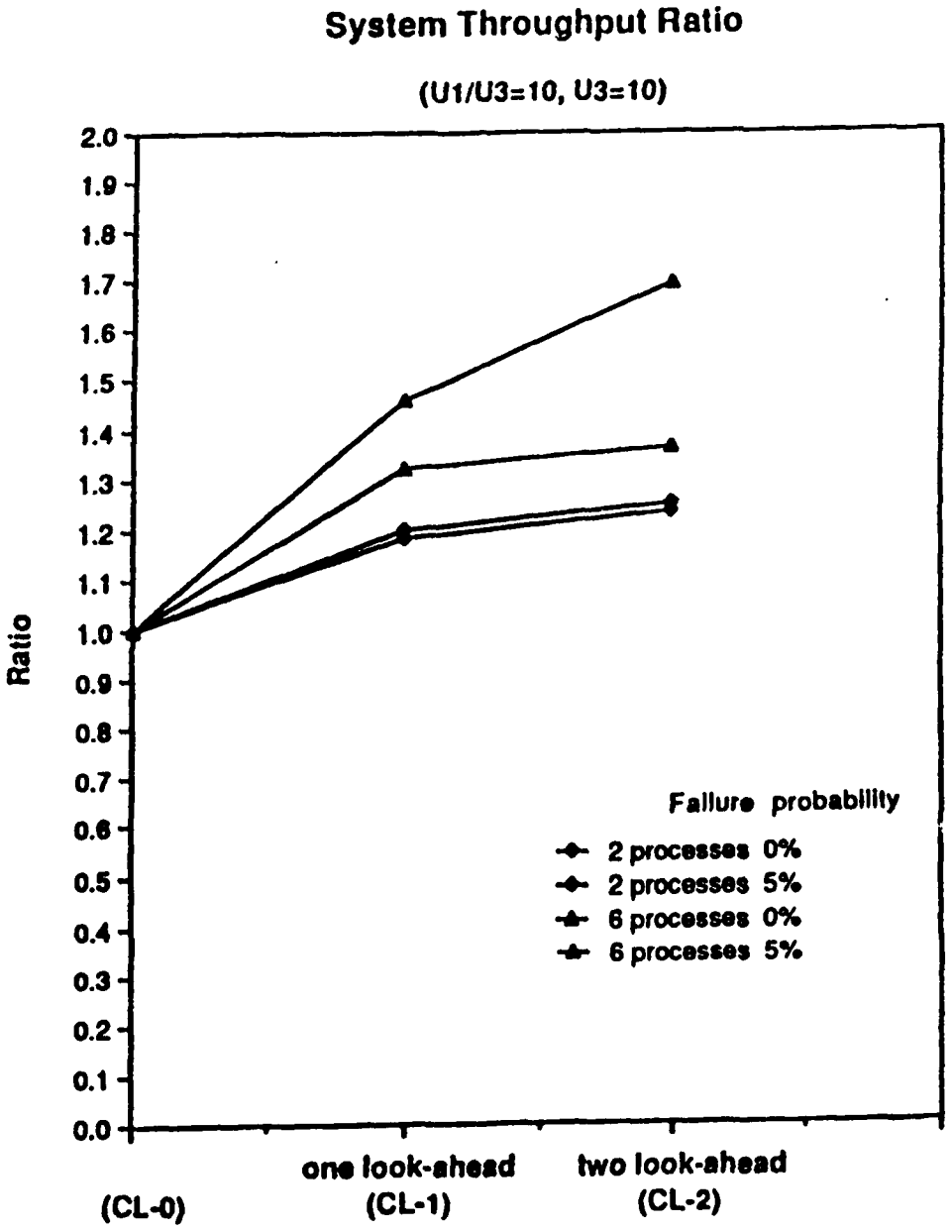


Figure 14. System Throughput Ratio.

4.2 Resource utilization

The number of processors idling due to the synchronization required on the processes inside the conversation is represented by the length of Q2 in the case of CL-0 and by the sum of the lengths of Q2 and Q2' in the case of CL-1. The following formulae can be used to derive the resource utilization.

(1) Under CL-0

$$MQL2(0)=\sum_{n2=1}^{N-1}\sum_{n1+n3=N-n2} n2*P(n1,n2,n3) + \sum_{n2=1}^{N-1} n2*P(\varepsilon,n2,n3)$$

(2) Under CL-1

$$MQL2(1)=\sum_{n2'=1}^{N-1}\sum_{n1+n3+n1'+n3'=N-n2'} n2'*P(n1,0,n3,n1',n2',n3')$$

$$+\sum_{n2'=1}^{N-1}\sum_{n1+n3+n3'=N-n2'} n2'*P(n1,0,n3,\varepsilon,n2',n3')$$

$$+\sum_{n2=1}^{N-1} n2*P(\varepsilon,n2,n3,0,0,0)$$

(3) Under CL-2

$$MQL2(2)=\sum_{n2'=1}^{N-1}\sum_{n1+n3+n1'+n3'+n1"+n3"=N-n2"} n2"*P(n1,0,n3,n1',0,n3',n1",n2",n3")$$

$$+\sum_{n2'=1}^{N-1}\sum_{n1+n3+n1'+n3'+n3"=N-n2"} n2"*P(n1,0,n3,n1',0,n3',\varepsilon,n2",n3")$$

$$+\sum_{n2'=1}^{N-1}\sum_{n3+n1'+n3'=N-n2} n2*P(\varepsilon,n2,n3,n1',0,n3',0,0,0)$$

$$+ \sum_{n2'=1}^{N-1} n2'*P(0,0,0,\varepsilon,n2',n3',0,0,0)$$

Figures 15 and 16 depict the queue lengths under both CL-0 and CL-1 for the cases where the number of processes are two and six, respectively. The probability of successful acceptance tests ($1-\alpha$) varies from 0.5 to 1. The completion rate for the non-conversation task (u3) was fixed at 10. Each figure depicts five different curves for each conversation scheme, each curve corresponding to a different execution time ratio (u1/u3).

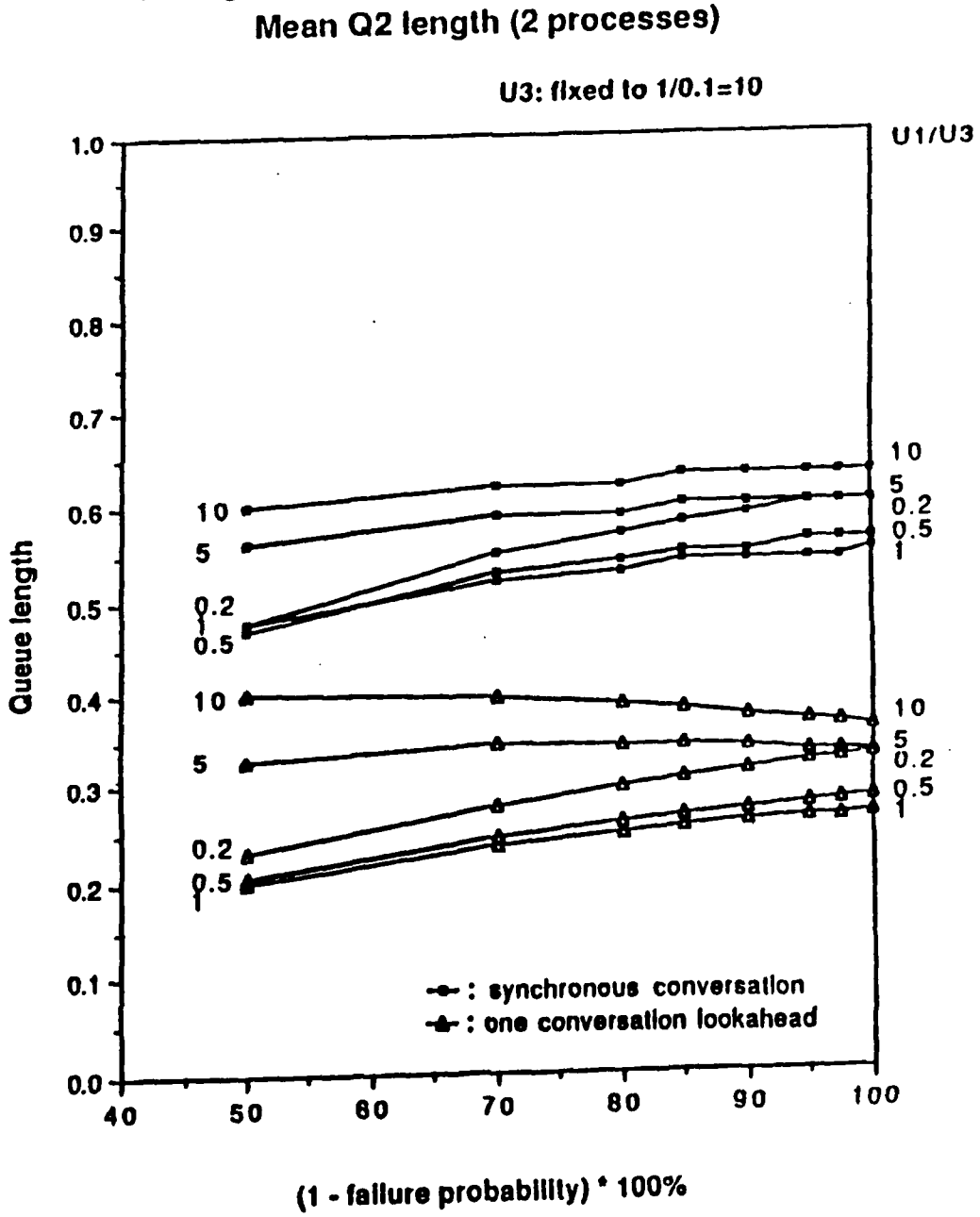


Figure 15. Mean Q2 Length under CL-0 and CL-1 (2 processes).

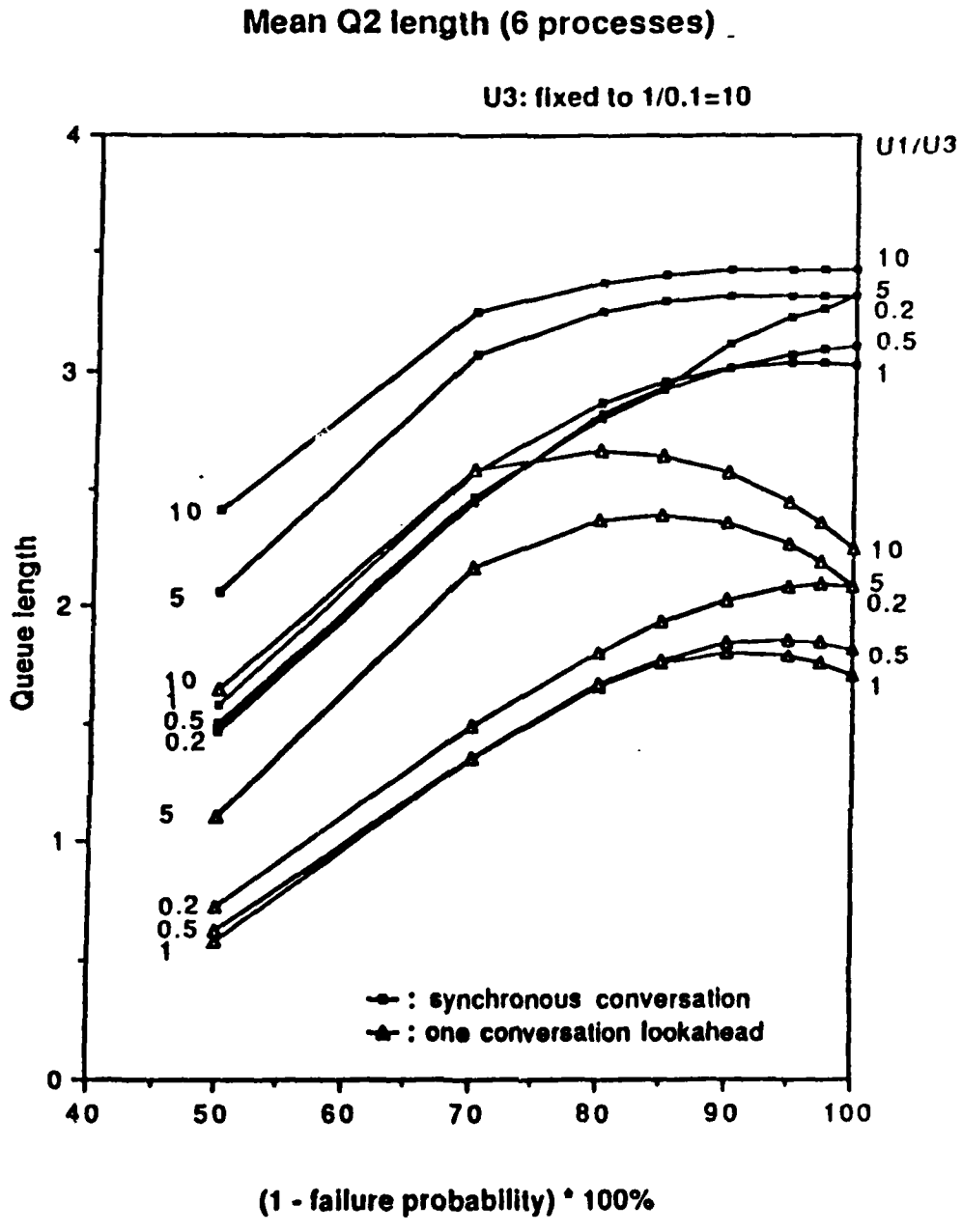


Figure 16. Mean Q2 Length under CL-0 and CL-1 (6 processes).

As expected, processor utilization is substantially higher under CL-1 than under CL-0 when the failure probability is within a practical range ($0 < \alpha << 0.05$). When the failure probability is close to zero in a system of six processors and u1/u3 is 10, the expected percentage of busy processors is about 63% under CL-1 whereas it is only about 43% under

CL-0. Also, under CL-0, processor utilization does not get much better as the failure probability approaches zero whereas it increases faster under CL-1.

Figure 17 shows resource utilization in a six-processor system under CL- 0, CL-1, and CL-2. Two different cases of u1/u3, 1 and 10, are shown. When u1/u3 is 10, the expected number of idling processors is 3.4 (57% of the processors available) under CL-0, 2.2 (37%) under CL-1, and 1.7 (28%) under CL-2.

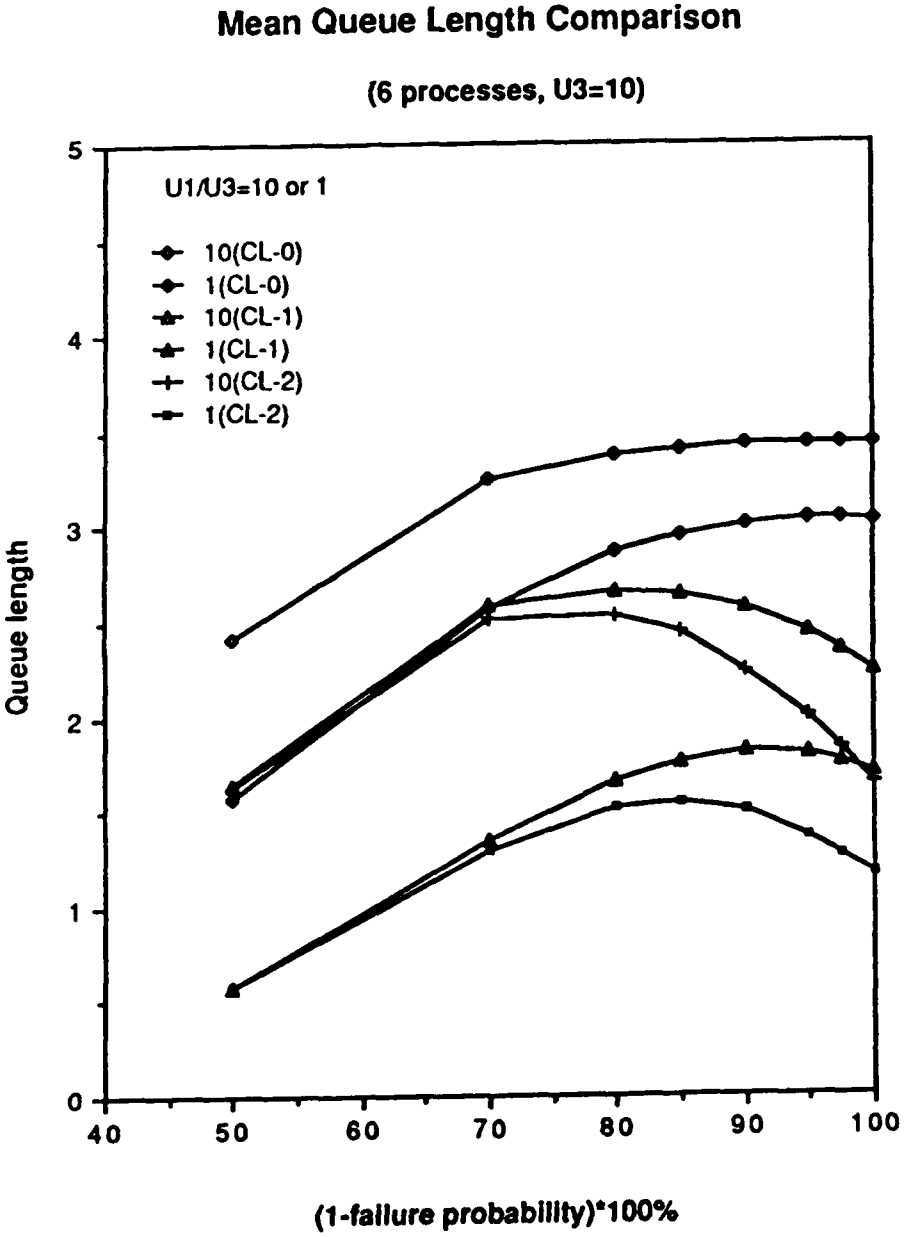


Figure 17. Comparison of Mean Q2 Length under CL-0, CL-1, and CL-2.

4.3 Conversation participation time

Mean participation time, Wp, can be obtained by use of the Little's Law [Kle75], i.e., Wp = L/T where L is the queue length and T is the throughput of the queue server. Since the customers in Q1 (also Q1', Q1") and Q2 (also Q2', Q2") represent the processes inside a conversation, L is the sum of the lengths of those queues and T is N*TPc where N is the number of processes and TPc is the number of successful conversations per unit time discussed in Section 4.1.

(1) Under CL-0

$$Wp(0) = (MQL1(0)+MQL2(0)) / (N*TPc(0))$$

(2) Under CL-1

$$Wp(1) = (MQL1(1)+MQL2(1)+MQL1'(1)+MQL2'(1)) / (N*TPc(1))$$

(3) Under CL-2

$$Wr(2) = (MQL1(2)+MQL2(2)+MQL1'(2)+MQL2'(2) +MQL1"(2)+MQL2"(2)) / (N*TPc(2)),$$

where MQLm(n) denotes the length of Queue Qm in model CL-n and TPc(n) denotes the system throughput under CL-n.

Figure 18 shows mean participation times under CL-0, CL-1, and CL-2 when the number of participating processes is six. Three different cases of failure probability are plotted. It shows that when the failure probability is within a practical range, the mean participation time improves significantly as the scheme changes from CL-0 to CL-1, but considerably less as the scheme changes from CL-1 to CL-2.

5. SUMMARY

Overall the lookahead approach reduces the synchronization overhead of the conversation scheme to a significant extent. The system performance improves substantially by all three measures used in this chapter. Interestingly, as the synchronization overhead plays

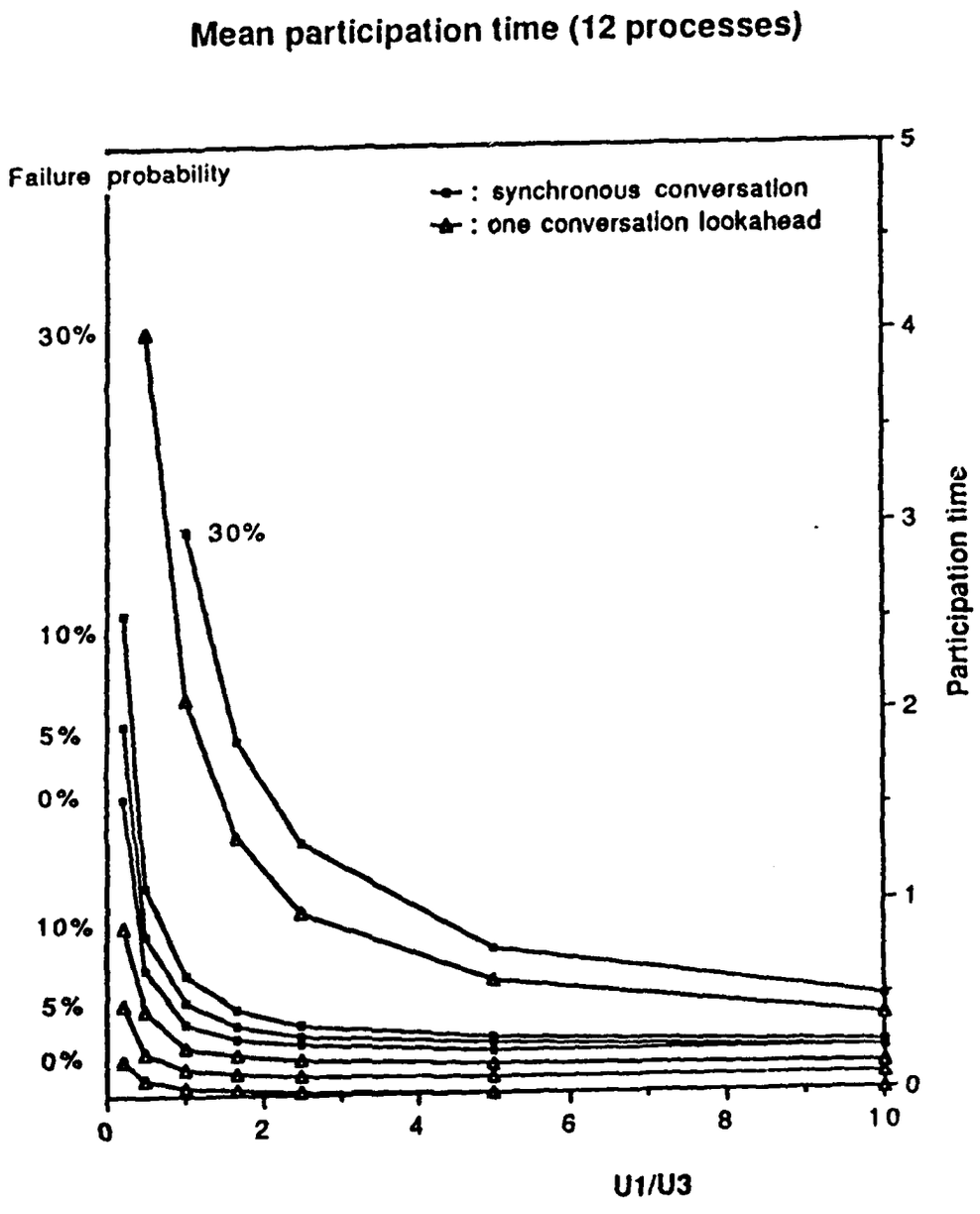


Figure 18. Conversation Participation Time under CL-0 and CL-1 (12 proecsses).

a less dominant role in determining the system performance under the lookahead approach (than under the basic conversation execution scheme), the impact of the acceptance test failure probability on the system performance is more noticeable. As the scope of lookahead

increases, the system performance improvement becomes gradually less substantial although implementation costs and recovery costs may grow steadily. Therefore, determination of a suitable limit on the scope of lookahead requires a tradeoff analysis reflecting various environmental characteristics.

The queueing network models developed in this chapter can be extended to represent the systems in which processes engage in multiple types of conversations including some nested within others. For example, Q3 in Figure 5 can be expanded into a series of Q1- and Q2-types representing different types of conversations followed by a queue of Q3-type in order to represent the systems in which processes engage in multiple types of non-nested conversations under CL-0. Such extended models will be useful in evaluating the potential performance of the systems being considered in specific application environments.

Formulation and analysis of the lookahead approach to execution of conversations represent only one of many research tasks needed to establish the conversation scheme as a design technique that can be widely practiced. There are still several other fundamental questions regarding the conversation scheme, e.g., how to design effective conversation acceptance tests and alternate interacting sessions, that remain unsatisfactorily answered. An experimental work aimed at finding answers to such questions and at validation of the predicted performances discussed here, is regarded as a highly worthwhile subject for future research.

6. REFERENCES

[Bha87] B. Bhargava, editor. "Concurrency and Reliability in Distributed Systems", Van Nostrand and Reinhold, 1987.

[Cam83] Campbell, R.H., Anderson, T., and Randell, B., "Practical Fault Tolerant Software for Asynchronous Systems", Proc. IFAC Safecomp 83, 1983, pp.59-65.

[Dav80] Davis, C.G. and Couch, R.L., "Ballistic Missile Defense: A Supercomputer Challenge", IEEE Computer, Nov. 1980, pp.37-46.

[Fou84] Foudriat, E.C., et al., "An Operating System for future Aerospace Vehicle Computer Systems", NASA Technical Memorandum 85784, April 1984.

[Gre85] Gregory, S.T. and Knight, J.C., "A new Linguistic Approach to Backward Error Recovery", Proc. FTCS-15, 1985, pp.404-409.

[Hec76] Hecht, H., "Fault-Tolerant Software for Real-Time Applications", Computing Surverys, Dec. 1976, pp.391-407.

[Kim76] Kim, K.H., Russell, D.L., and Jenson, M.J., "Language Tools for Fault-Tolerant

Programming", Tech. Memo. PETP-1, Electronic Sciences Lab., USC, Nov. 1976.

[Kim82] Kim, K.H., "Approaches to Mechanization of the Conversation Scheme Based on Monitor", IEEE Trans. on Software Eng., Vol. SE-8, No. 3, May 1982, pp.189-197.

[Kim85] Kim, K.H. Yang, S.M., and Kim, M.H., "Implementation of Concurrent Programming Language Facilities Supporting Conversation Structuring", Proc. 1985 COMPSAC, Int'l Computer Software & Application Conf., Oct. 1985, pp.445- 453.

[Kim86] Kim, K.H., Heu, S., and Yang, S.M., "An Analysis of the Execution Overhead Inherent in the Conversation Scheme", Proc. 5th Symp. on Reliability in Distributed Software and Database Systems, Jan. 1986, pp.159-168.

[Kle75] Kleinrock, L., 'Queueing Systems Volume 1: Theory', John Wiley & Sons, 1975.

[McD82] McDonald, W.C. and Wayne Smith, R., "A Flexible Distributed Testbed for Real-Time Applications", IEEE Computer, Oct. 1982, pp.25-39.

[Oza88] Ozaki, B.M., Fernandez, E.B., and Gudes, E., "Software Fault Tolerance in Architectures with Hierarchical Protection Levels", IEEE Micro, Vol.8, No.4, Aug. 1988, pp.30-43.

[Ram81] Ramamoorthy, C.V. et al., "Application of a Methodology for the Development and Validation of Reliable Process Control Software", IEEE Trans.
on Software Engr., Vol. SE-7, No.6, Nov. 1981, pp. 537-555.

[Ran75] Randell, B., "System Structure for Software Fault Tolerance", IEEE Trans. on Software Eng., Vol.SE-1, June 1975, pp.220-232.

[Rus79] Russell, D.L. and Tiederman, M.J., "Multiprocess Recovery using Conversations", Proc. FTCS-9, 1979, pp.106-109.

[Shr87] Shirivastava, S.K., Mancini, L., and Randell, B., "On the Duality of Fault Tolerant System Structures", Tech. Memo. SRM/455, Computing Lab., Univ. of Newcastle upon Tyne, 1987.

[Toy87] Toy, W.N., "Fault-Tolerant Computing", A chapter in Advances in Computers, Vol.26, Academic Press, 1987, pp.201-279.

[Yan86] Yang, S.M., 'Timing Specification and Verification for Fault-Tolerant Distributed Computer Systems', Ph.D. Dissertation, Dept. of Computer Sci. & Engr., Univ. of South Florida, Dec. 1986.

Appendix A. Steady State Balance Equations

The steady-state balance equations for the states of each queueing network model developed in Section 3.3 are as follows.

(1) Model for CL-0

$$P(n1,n2,n3) = [1/(n1*u1+n3*u3)]*[u1*(n1+1)*(1-\alpha)*P(n1+1,n2-1,n3) + u3*(n3+1)*P(n1-1,n2,n3+1)]$$

$$P(\varepsilon,n2,n3) = [1/(n3*u3)] * [\sum_{i=1}^{n2} u1*i*\alpha*P(i,n2-i,n3) + u3*(n3+1)*P(\varepsilon,n2-1,n3+1)]$$

(2) Model for CL-1

$$P(n1,0,n3,n1',n2',n3') = [1/((n1+n1')*u1+(n3+n3')*u3)] * [u1*(n1+1)*(1-\alpha)*P(n1+1,0,n3,n1',n2',n3'-1) + u3*(n3+1)*P(n1-1,0,n3+1,n1',n2',n3') + u1*(n1'+1)*(1-\alpha)*P(n1,0,n3,n1'+1,n2'-1,n3') + u3*(n3'+1)*P(n1,0,n3,n1'-1,n2',n3'+1)]$$

$$P(n1,0,n3,\varepsilon,n2',n3') = [1/(n1*u1+(n3+n3')*u3)] * [u1*(n1+1)*(1-\alpha)*P(n1+1,0,n3,\varepsilon,n2',n3'-1) + u3*(n3+1)*P(n1-1,0,n3+1,\varepsilon,n2',n3')$$

$$+ \sum_{i=1}^{n2'} u1*i*\alpha*P(n1,0,n3,i,n2'-i,n3')$$

$$+ u3*(n3'+1)*P(n1,0,n3,\varepsilon,n2'-1,n3'+1)]$$

$$P(\varepsilon,n2,n3,0,0,0)$$

$$= [1/(n3*u3)] * \sum_{i=1}^{n2} \sum_{X=0}^{n2-i} u1*i*\alpha$$

$$* \ P(i,n2-i-X,n3,n1',n2',n3')$$

$$+ u3*(n3+1)*P(\varepsilon,n2-1,n3+1,0,0,0)],$$

where $X=n1'+n2'+n3'$

(Note: The above steady state balance equations cover only the case where switch S1 is closed and S2 is open in the model for CL-1. Due to the symmetric characteristics of this queueing network, it is not necessary to consider the other case where switch S1 is open and S2 is closed. Similarly, in the following, we consider only the case where switches S1 and S2 are closed and S3 is open in the model for CL-2.)

(3) Model for CL-2

$$P(n1,0,n3,n1',0,n3',n1'',n2'',n3'') =$$

$$[1/((n1+n1'+n1'')*u1+(n3+n3'+n3'')*u3)]$$

$$* \ [u1*(n1+1)*(1-\alpha)*P(n1+1,0,n3,n1',0,n3',n1'',n2'',n3''-1)$$

$$+ \ u3*(n3+1)*P(n1-1,0,n3+1,n1',0,n3,n1'',n2'',n3'')$$

$$+ \ u1*(n1'+1)*(1-\alpha)*P(n1,0,n3-1,n1'+1,0,n3',n1'',n2'',n3'')$$

$$+ \ u3*(n3'+1)*P(n1,0,n3,n1'-1,0,n3'+1,n1'',n2'',n3'')$$

$$+ u1*(n1''+1)*(1-\alpha)*P(n1,0,n3,n1',0,n3',n1''+1,n2''-1,n3'')$$

$$+ u3*(n3''+1)*P(n1,0,n3,n1',0,n3',n1''-1,n2'',n3''+1)]$$

$$P(n1,0,n3,n1',0,n3',\varepsilon,n2'',n3'') =$$

$$[1/((n1+n1')*u1+(n3+n3'+n3'')*u3)]$$

$$* [u1*(n1+1)*(1-\alpha)*P(n1+1,0,n3,n1',0,n3',\varepsilon,n2'',n3''-1)$$

$$+ u3*(n3+1)*P(n1-1,0,n3+1,n1',0,n3',\varepsilon,n2'',n3'')$$

$$+ u1*(n1'+1)*(1-\alpha)*P(n1,0,n3-1,n1'+1,0,n3',\varepsilon,n2'',n3'')$$

$$+ u3*(n3'+1)*P(n1,0,n3,n1'-1,0,n3'+1,\varepsilon,n2'',n3'')$$

$$+ \sum_{i=1}^{n2''} u1*i*\alpha$$

$$*P(n1,0,n3,n1',0,n3',i,n2''-i,n3'')$$

$$+ u3*(n3''+1)*P(n1,0,n3,n1',0,n3',\varepsilon,n2''-1,n3''+1)]$$

$$P(e,n2,n3,n1',0,n3',0,0.0)$$

$$= [1/(n1'*u1+(n3+n3')*u3)]$$

$$* [\sum_{i=1}^{n2} \sum_{Y=0}^{n2-i} u1*i*\alpha$$

$$+ u3*(n3+1)*P(\varepsilon,n2-1,n3+1,n1',0,n3',0,0,0)$$

$$+ u1*(n1'+1)*(1-\alpha)*P(\varepsilon,n2,n3-1,n1'+1,0,n3',0,0,0)$$

$$+ u3*(n3'+1)*P(\varepsilon,n2,n3,n1'-1,0,n3'+1,0,0,0)],$$

where Y=n1"+n2"+n3"

P(0,0,0,e,n2',n3',0,0,0)

= [1/(n3'*u3)]

$$* \left[\sum_{i=1}^{n2'} \sum_{Z=0}^{n2'-i} u1*i*\alpha \right.$$

*P(n1,0,n3,i,n2'-i-Z,n3',n1",n2",n3")

+ u3*(n3'+1)*P(0,0,0,ε,n2'-1,n3'+1,0,0,0)],

where Z=n1+n3+n1"+n2"+n3"

CHAPTER V

AN APPROACH TO DYNAMIC EXECUTION TIME ESTIMATION

1. INTRODUCTION

Computer systems are being increasingly used for control of complex and sophisticated real-time applications such as flight control systems and ballistic missile defense systems. Since the real-time systems operate in the presence of concurrent events (e.g., inputs from multiple sensors and outputs to multiple actuators) the software is usually designed for a large number of concurrent tasks which may have different priorities and deadlines. Such tasks can be divided into two categories according to the strictness of their temporal constraints, namely, hard-real-time tasks and soft-real-time tasks. Output obtained after a deadline from a hard-real-time task is of no use, while the output usability for a soft-real-time task only decreases. The scheduling of the real-time tasks is not trivial due to these temporal requirements (or execution time deadlines). Most deadline-driven scheduling algorithms guarantee the schedulability of tasks at preruntime assuming that the worst-case execution time of each task is known. Although this preruntime (or design time) guarantee is necessary for critical real-time tasks, there is a limitation. As the real-time systems become more complicated and distributed it is almost impossible to guarantee the schedulability of all tasks at design time.

Adaptive scheduling can overcome this limitation and maximize processor utilization by incorporating runtime optimization. The adaptive scheduler should be designed such that it will (1) guarantee the most critical tasks are scheduled before deadlines, whose schedulability is verified at design time and (2) accommodate other tasks (usually soft-real-time tasks and aperiodic tasks) efficiently at runtime. For the former the deadline-driven scheduling algorithms, such as the shortest deadline first or least laxity preemptive scheduling, can be used. The "reward value" based on the time-value function [Jen85] is an example of an efficiency measurement for the latter type tasks.

Since all these real-time scheduling algorithms assume the execution time behavior of tasks is known, the accurate estimation of the execution time is crucial. System failure may result from a missed deadline due to under-estimated task execution times. On the other hand, over-estimates degrade system utilization significantly. (This over-estimate may create more serious problem than low CPU utilization under the adaptive scheduling such as time-value function based scheduling. We will show an example in Section 4.) Although several approaches to execution time estimation have been proposed [Gop90, Hab90, Par90, Pus89, Wei81] they seem insufficient to support efficient implementation of adaptive scheduling.

This chapter proposes a Dynamic Execution Time (DET) estimation technique for determining the worst-case execution time behavior of tasks. The DET estimation comprises two new concepts: (1) compile time estimation based on semantic as well as syntactic analysis, and (2) runtime estimation based on Execution Time (ET-) functions. DET estimation is

performed by two analyzers: Compile-time Analyzer (CA) and Runtime Analyzer (RA). The CA estimates the upper bound worst-case execution time of program segments and produces an ET-function which is used by the RA during runtime to calculate the remaining execution time of a task accurately for the scheduler. The usefulness of DET estimation is demonstrated with an example.

2. KNOWN ESTIMATION APPROACHES

Estimation of execution time is difficult because of input-dependent branches and loops, and unpredictable delays associated with resource contention. Moreover, in a distributed real-time system, the communication and synchronization between tasks make the analysis of timing behavior much more difficult. The experimental approach can, if the domain of the possible inputs for a given program segment is finite, obtain accurate worst-case execution times by making some finite number of test runs. However, the process is very costly and often impossible in most cases since the input domain is infinite. Hence, analytical approaches, which estimate execution time by analyzing program segments, have been studied by researchers.

Wei [Wei81] proposed a "limit" keyword in Path Pascal, which bounds the maximum number of loops, e.g., "for i=1 to N do limit 1000". In [Pus89] a loop bound can either be a limit for the maximum number of iterations or a time limit for the termination of the loop. Puschner and Koza also proposed new language constructs, "scopes" and "markers". A scope is a segment of a program's instruction code, limited by a special scope construct that is embedded into the syntax of a programming language. A marker is a special mark located within a scope. It specifies the maximal number of times the marked position in the program may be passed by the program flow between entering and leaving the scope. Markers are mainly used to state that the number of executions of one or more paths through a loop can be bounded. Loop sequence is another language construct which can be used when the sum of the iterations of the loops is bounded. That is, a loop sequence is a series of loops which has the property that the sum of the iterations of the single loops does not exceed a given constant value at runtime.

Puschner and Koza try to reduce the gap between the "structural" worst-case time and the "realistic" worst-case time by having programmers (or designers) provide information about the execution of their algorithms as much as possible. However, in many real-time programs, the control flows are determined by input values at runtime and the domain of the input values are often infinite. In such cases, very little information can be provided by programmers. Moreover, programmers may provide wrong information which causes the

program to fail at runtime.

Park and Shaw [Par91] consider the worst-case and best-case bounds using "timing schema" which are essential formulae for computing upper and lower bounds for program constructs. Their approach (which uses a special tool) decomposes a program statement into atomic blocks by examining the object code generated by a compiler. They compute the execution time of the statement using the execution time of the atomic blocks and timing schema. Loops are handled in a similar fashion except that they must know the range of iterations. Park and Shaw include control costs, and handle interferences such as clock interrupts by removing interference times from measured times.

The approaches discussed so far are considered to be static analytical execution time estimation techniques. The estimations are based mainly on syntactic structure of program segments and the hints provided by programmers. In contrast, Haban and Shin compute the remaining pure execution time of a task at runtime [Hab90] by employing a real-time monitor. The monitor is composed of dedicated hardware, called "test and measurement processors", that measure, with minimal interference, the true execution time including any resource sharing delay. The monitor is very useful in determining whether a task will terminate within a deadline, which makes adaptive scheduling possible.

In [Gop90], Gopinath and Gupta propose "Compiler Assisted Adaptive Scheduling" (CAADS) where the compiler examines the application code and inserts measurement code at appropriate boundaries. The measurement code enables execution times of the various parts of the program to be determined at runtime. Code reordering is discussed to allow greater adaptability and early failure detection. CAADS, unlike Haban and Shin's approach, does not require any hardware support. However, both approaches [Hab90, Gop90] ignore that the execution time of periodic tasks may vary significantly depending on the input values of each iteration.

3. DYNAMIC EXECUTION TIME ESTIMATION

Dynamic Execution Time (DET) estimation is performed by two analyzers: Compile-time Analyzer (CA) and Runtime Analyzer (RA). The CA attempts to estimate the execution time of a program not only by syntactic (or structural) analysis, but also by semantic analysis. The latter is done by tracing the influence of variable input data as it progresses through a simulation of program execution. As a result, each statement is classified either as a fixed statement (whose exact execution time is known at compile time), or as a variable statement (whose exact execution time is not known at compile time). The statements are grouped into segments based on the rule discussed in Section 3.1. The Execution Time (ET-) function and

the worst-case execution time of each segment is then obtained.

The RA calculates the remaining execution time of a task at runtime. The quick and accurate estimation of the remaining execution time is possible by using the ET-functions generated
by the CA. This time information makes efficient implementation of adaptive scheduling possible. Figure 1 shows the relationship among the CA, RA, and scheduler. We will discuss the details in the following subsections.

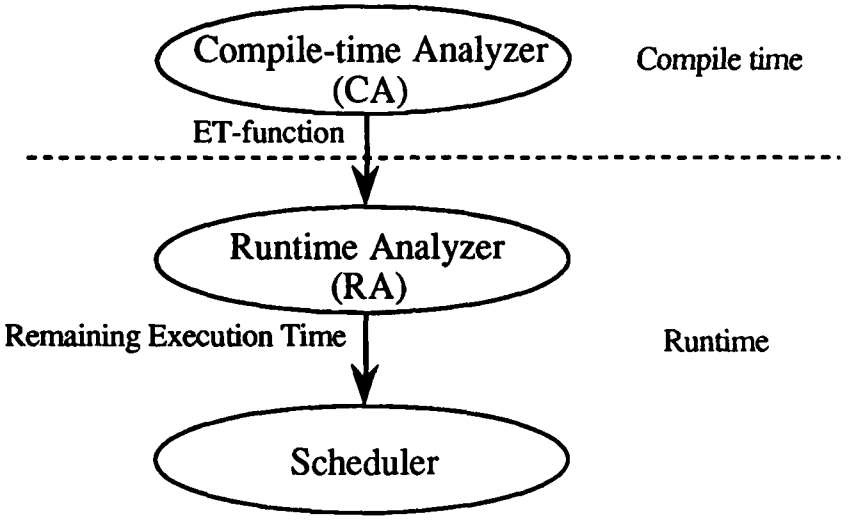


Figure 1. Relationship among CA, RA and Scheduler

3.1 Compile-time Analyzer

The Compile-time Analyzer (CA) has four phases: F/V-statement classification, partitioning, worst-case execution time estimation, and ET-function generation.

F/V-statement classification

During this phase, each statement of a program is classified either as a fixed statement (F-statement) or as a variable statement (V-statement). The execution time of an F-statement is known at compile time as either a constant or almost constant. The execution time of a V-statement, on the other hand, is unknown at compile time. There are largely two kinds of V-statements.

(1) The execution time of the statement is determined by the values of some variables or

operands. Therefore, if the values are unknown at compile time the exact execution time is also unknown. Examples include branch statements, loop statements, and statements involving string manipulation. Once the values of the unknown variables are known, very accurate execution time can be calculated.

(2) There are statements which may be blocked at runtime. Examples include any statement requiring synchronization such as I/O statements, "send" and "receive" statements, and rendezvous calls. These statements need a special treatment at runtime by the scheduler.

Our goal in this phase is not only to classify statements into F- or V-statements but also, more importantly, to minimize the V-statements as much as possible. This is possible by tracing the values of each variables (i.e., semantic analysis) along with the syntactic analysis. Since our approach to the syntactic analysis of statements is not different from others we focus our discussion on the semantic analysis.

It should be noted that the actual execution time also depends on the compiler optimization. The examples which are used to illustrate the DET estimation approach are provided at the source code level (i.e., C). However, this approach is claimed to be directly translatable to the compiled object image or assembly code. In fact, the semantic analysis is more naturally explained using low level assembly language terminology. Note also that the terms "instruction" and "statement" (including "operand" and "variable") are used interchangeably even though one one high level statement may consist of multiple instructions.

Semantic analysis is done by simulating the execution of each instruction and tracing the values of operands. Tracing is done by the CA through a process of interpretation which keeps track of information about the data (or operand) by taging it with a boolean flag called the "known" flag or k-flag for short [Reh87].

The k-flag allows the data to be identified in its associated word as either constant (known) or variable (unknown) at any point in time during the interpretation of the program. When a program to be analyzed is first loaded into memory, all cells or words that are loaded are marked as "known" (k-flag true), since they will always have this value at the start of execution. The memory cells that are not touched by the loading process are marked as "unknown" (k-flag false), since they are uninitialized. (If the target system always initializes memory prior to loading, then the k-flags are adjusted accordingly.) As interpretation takes place, the k-flags of an operation and its operands are combined by a logical AND to determine the k-flag of the result. Thus, if at least one of the k-flags of the instruction and its operands is false, the k-flag of the result is false.

For example, for the statement "x:=y+z" the addition is performed and the result is stored in x. At the same time, the k-flag of the operations (in this case "+" and ":=") and the k-flag of y and z are ANDed, and the result is stored in x's k-flag. The k-flags of most

operations are true except for those operations whose execution times depend on their operands, such as string manipulation instructions. More detail on k-flag is reported in [Reh87].

During semantic analysis a statement is marked as "Fixed" and called an F-statement if the execution time of the statement is known at compile time. Otherwise, the statement is marked as "Variable" and called a V-statement.

For a simple statement, marking of the statement is determined by the following rule:

if (k-flag of operation) OR
(k-flag of $operand_1$ AND k-flag of $operand_2$ AND • • •)
then F-statement
else V-statement

If the k-flag of an operation is true then the execution time of the statement is fixed regardless of its operands. On the other hand, if the k-flag of an operation is false the execution time of the statement depends on the operands. Therefore, if the operands are known then the statement is an F-statement. (If a statement consists of more than one instruction, the result of each instruction is ANDed.) For example, statements (1) through (5) in Figure 2 are marked as F-statement. Table 1 shows the relationship among k-flags of operation and operands and F/V marking.

Table 1.
Relationship among k-flags and F/V marking

k-flag of operation	k-flag of operand	F/V-statement
True	True	F
True	False	F
False	True	F
False	False	V

The execution time of a branch statement is determined by the branch choice. If the k-flag of the branch factor is true, i.e., the variables which determine the selection of the branch are known, we know that which branch will be executed at compile time. However, this does not necessarily mean that the execution of the whole branch statement is known because the selected branch may have some V-statement(s). In such cases branches of "if" statement are treated as separate blocks.

If the k-flag of the branch factor is false, the branch that is selected is not known at compile time. However, there may be cases in which the difference between the execution time of the longest branch and that of the shortest one is negligible or less than a certain value

determined according to the timing criticalness of the program. In such cases even if the branch factor is not known the execution time of the whole branch statement is almost constant, which should be treated as an F-statement. For this purpose, a flag called a "negligible flag" or "n-flag" for short is introduced. The n-flag of a branch statement is true if the execution time of the statement is almost constant regardless of the branch selected. For example, "if(a<b) c=5 else c=6" with unknown variables a, b and c. In this case although it is not known which branch is selected hence the k-flag of c is false, the execution time of this statement is constant hence an F-statement. Table 2 summarizes the relationship.

Table 2.
Relationship among n-flag, k-flag, and F/V marking

n-flag	k-flag of branch factor	F/V-statement
True	True	F
True	False	F
False	True	F/V
False	False	V

The execution time of a loop statement depends on both the loop factor which decides the number of loop iteration and body of the loop. Therefore, a loop statement is an F-statement only if the k-flag of the loop factor is true and the loop body has no V-statement. If a branch or loop statement has V-statement(s) in it F/V-statement classification is nested. As shown in Figure 2 statement (7) has three nested statements, (7.1), (7.2) and (7.3) and in turn (7.2) also has nested statements. These nested statements are, unless all of them are F-statements, partitioned as nested segments explained later.

We now discuss how the statements whose execution is possibly blocked at runtime can be treated. Among others, "send" and "receive" statements are typical in a distributed computing environment. We assume an asynchronous communication with the following syntax:

send message to destination

receive message from source within time_limit

Since the sending message is assumed to be never blocked under asynchronous communication "send" statement can be treated as a normal statement with no delay. (Although the exact execution time is sometimes unknown at compile time due to unknown size of messages.) The "receive" statement, on the other hand, may be blocked if the message is not ready. Since the delay may depend on many factors it is very difficult, if not impossible, to estimate the exact waiting time, even worst-case waiting time sometimes. Therefore, it is inevitable to attach the

```
# include <stdio.h>

# define EOS '\0'       /* End of String */
# define TIME_LIMIT 10

char string[80];

main()
{
        char c, source, destination;
        int  i, low, in, high, digit, sum;

(1)     i=0;                                                    F
(2)     low=0;                                                  F
(3)     in=0;                                                   F
(4)     high=0;                                                 F
(5)     sum=0;                                                  F

(6)     receive(string, source, TIME_LIMIT);                    V

(7)     while( c=string[i] != EOS )                             V
        {
(7.1)           send(c, destination);                           F
(7.2)           if( c < '0' )                                   V
(7.2.1)                 low++                                   F
                else if( c > '9' )
(7.2.2)                 high++                                  F
                else
                {
(7.2.3)                 in++;                                   F
(7.2.4)                 digit=atoi(c);                          F
(7.2.5)                 sum=sum+digit;                          F
(7.2.6)                 if(sum>100)                             V
                        {
(7.2.6.1)                       send(sum, destination);         F
(7.2.6.2)                       sum=0;                          F
                        }
                }
(7.3)           i++;                                            F
        }

(8)     send(low, destination);                                 F
(9)     send(in, destination);                                  F
(10)    send(high, destination);                                F
}
```

F, V : segments

F, V : statements

Figure 2. Sample Program

time limit which is used for two purposes. At compile time this is used as the worst-case execution time of the receive statement and at runtime time_limit is a timeout. (In practice, the programmer may want to specify the actions after timeout occurs.) Under the synchronous communication "send" statement is treated same as "receive" statement since the "send" statement can also be blocked. The other statements which are not discussed here can be treated in the same manner.

Partitioning

Once the analysis is done the program is partitioned into segments. (In real implementation this phase and the later phases can be done simultaneously with the F/V-statement classification.) The partitioning is done based on the following rule:

(1) A program block contains at most one F-segment and zero or more V-segments.

(2) An F-segment contains F-statements only.

(3) A V-segment contains one leading V-statement and the following F-statement(s) which appear before the next V-statement (same level statement case).

(4) There are three different V-statements, i.e., v_branch_statement, v_loop_statement, and v_function_call_statement.

Therefore, a program can be represented using BNF notation as

block ::= [F-segment] {V-segment}
F-segment ::= F-statement {F-statement}
V-segment ::= V-statement {F-statement}
V-statement ::= v_branch_statement | v_loop_statement
| v_function_call_statement
v_branch_statement ::= block {block}
v_loop_statement ::= block

The sample program shown in Figure 2 has one F-segment followed by two V-segments. The second V-segment has a nested F- and V-segment in it.

Worst-case Execution Time Estimation

The worst-case execution time of each segment is estimated based on F/V marks. For an F-statement the exact execution time is calculated. While for a V-statement, the worst-case is chosen. For example,

if $expr_1$ $stmt_1$ max($T(expr_1) + T(stmt_1)$,
else if $expr_2$ $stmt_2$ $T(expr_1) + T(expr_2) + T(stmt_2)$,
• • • ⇒ • • •
else if $expr_{n-1}$ $stmt_{n-1}$ $T(expr_1) + T(expr_2) + \bullet\bullet\bullet + T(expr_{n-1}) + T(stmt_{n-1})$,
else $stmt_n$; $T(expr_1) + T(expr_2) + \bullet\bullet\bullet + T(expr_{n-1}) + T(stmt_n)$)

while *expr* **do** *stmt;*
⇒ MAXIMUM_NUMBER_OF_LOOPS * (T(*expr*) + T(*stmt*)) + T(*expr*)

The worst-case execution time of the statement whose execution is possibly blocked at runtime, e.g., statement (6) in Figure 2, is given as the sum of the maximum delay time and the execution time. The worst-case execution time obtained by the DET estimation should be of better quality, i.e., closer to the actual execution time than those obtained by other approaches due to the semantic analysis of a program. This information is used for the preruntime schedulability test as well as for the remaining execution time estimation.

ET-function Generation

The last phase of the CA (Compile-time Analyzer) is the generation of the ET-function for each segment. Instead of choosing the worst-case execution time of a V-statement, the execution time is expressed as a function of unknown variables. For example, the ET-function of "for i=1 to N do stmt" is a function of unknown variable N and can be expressed as "ET=N*T(stmt)" where T(stmt) is the execution time of each loop. In the same way the ET-function of the second V-segment in Figure 2 can be expressed as a function of the length of the string received. During runtime, the length should be known by the time that statement (7) is about to execute. Hence the RA (Runtime Analyzer) can calculate the accurate, not worst-case, execution time and provide this information to the scheduler. The ET-function is generated using a bottom-up parsing. Figure 3 shows the CA except the F/V-statement classification.

In this section we have explained how the CA classifies F/V-statements, partitions a program into F/V-segments, estimates the worst-case execution time, and generates the ET-function of each segment. All of the timing information is stored into the Timing Information Block (shown in Figure 4) and eventually copied into the process control block at runtime. Figure 5 shows the Timing Information Block generated for each segment of the program in

Figure 2. As shown in this figure for the F-segment AET is calculated whereas for the V-segment both WET and ET-function are calculated. The CA is being implemented in C language for the PL/0 language [Wir76], a subset of Pascal. (Part of the CA was implemented in Pascal for the PL/0 language by Rehm [Reh87].) The CA will be extended for the C language and will include provisions for estimating "send" and "receive" statements in distributed systems.

3.2 Runtime Analyzer

The Runtime Analyzer (RA) is a small program in the runtime kernel. The RA, whenever invoked by a task, calculates the remaining execution time of the task using the timing information in the process control block. Figure 6 shows the control and data flow among tasks, the RA, and the scheduler. (Although we draw the RA separately from the scheduler to show the control and data flow, it can be a part of the scheduler in real implementation.)

```
compile_time_analyzer()
{       int   i = 1,    /* segment number */
              j = 1;    /* statement number */

        do {  read statement j;
              if V_statement
              {       i = i + 1;
                      if branch_statement
                              for(k=1; not end_of_branch ; k++)
                              {       go to the kth branch;
                                      compile_time_analyzer();
                              }
                      else if loop_statement
                      {       go to loop body;
                              compile_time_analyzer();
                      }
                      WET[i] = worst-case execution time of statement j;
                      ET[i] = ET-function of statement j;
              }
              else    /* F-statement */
              {       AET = actual execution time of the statement j;
                      ET[i] = ET[i] + AET;
              }
              j = j + 1;
        } while not end_of_statement;
}
```

Figure 3. Compile-time Analyzer (except F/V-statement classification)

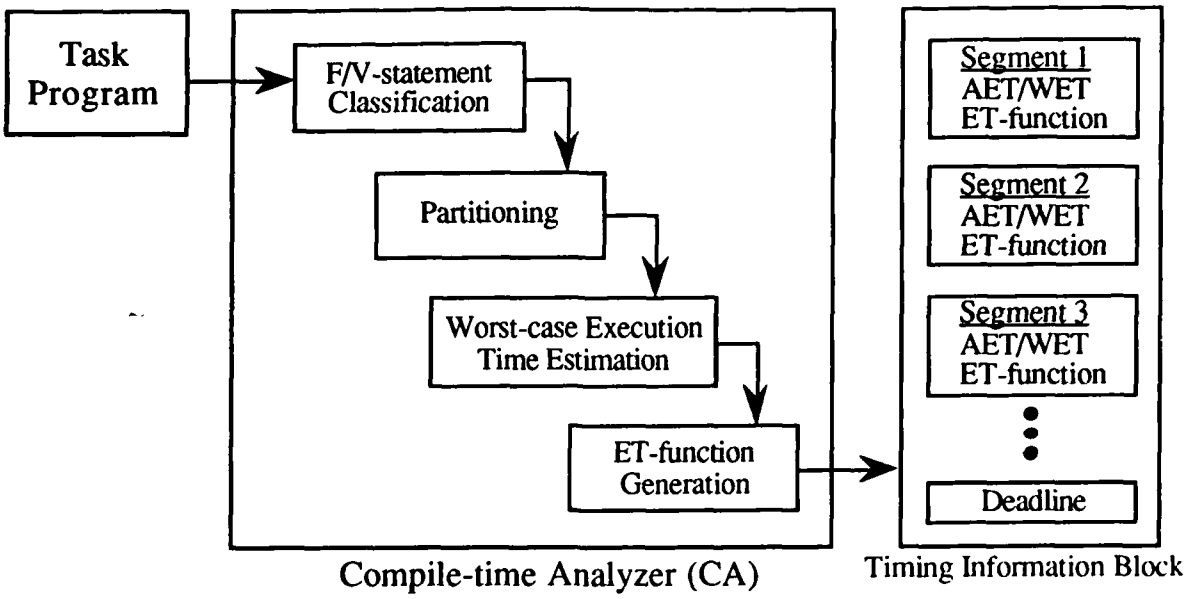


Figure 4. Block Diagram of Compile-time Analyzer (CA)

<table>
<tr><td>F-segment</td><td colspan="3">AET = 5 * $T_{assignment}$</td></tr>
<tr><td>V-segment</td><td colspan="3">WET = TIME_LIMIT + $T_{receive}$
ET-function = $T_{receive}$</td></tr>
<tr><td rowspan="8">V-segment</td><td colspan="3">WET = 80 * T_{loop_body} + 3 * T_{send}
ET-function = NUMBER_OF_LOOP * T_{loop_body} + 3 * T_{send}
NUMBER_OF_LOOP = SIZE_OF_INPUT_STRING</td></tr>
<tr><td>F-segment</td><td colspan="2">AET = T_{send}</td></tr>
<tr><td rowspan="6">V-segment</td><td colspan="2">WET
ET-function = T_{if} + $T_{increment}$
T_{if} = • • •</td></tr>
<tr><td>F-segment</td><td>AET</td></tr>
<tr><td>F-segment</td><td>AET</td></tr>
<tr><td>F-segment</td><td>AET</td></tr>
<tr><td rowspan="2">V-segment</td><td>WET/ET-function</td></tr>
<tr><td>F-segment</td><td>AET</td></tr>
</table>

* AET = Actual Execution Time
WET = Worst-case Execution Time
ET-function = Execution Time-function

Figure 5. Timing Information Block

When a program starts to run, the estimated remaining execution time is the same as the worst-case execution time. For example, the task shown in Table 3 has three segments with the worst-case execution time 100. Segment 1 is an F-segment with the estimated execution time 20. Segment 2 is a V-segment with the worst-case execution time 30 and the ET-function 5X+5. Segment 3 is also a V-segment. Now suppose, during the course of execution, that the value of X is 2 at the time that Segment 2 begins to execute. Since $ET_2 = 5(2) + 5 = 15$ the estimated remaining execution time is 65 instead of 80. Since the scheduler knows this before Segment 2 executes, the scheduler can allocate times more effectively. Similarly, once the value of Y is known the remaining execution time can be estimated more accurately. Figure 7 shows the pseudo codes of the RA.

Table 3. Example

Segment 1	F	AET_1=20	
Segment 2	V	WET_2=30	ET-function$_2$=5X + 5
Segment 3	V	WET_3=50	ET-function$_3$=10Y + 10

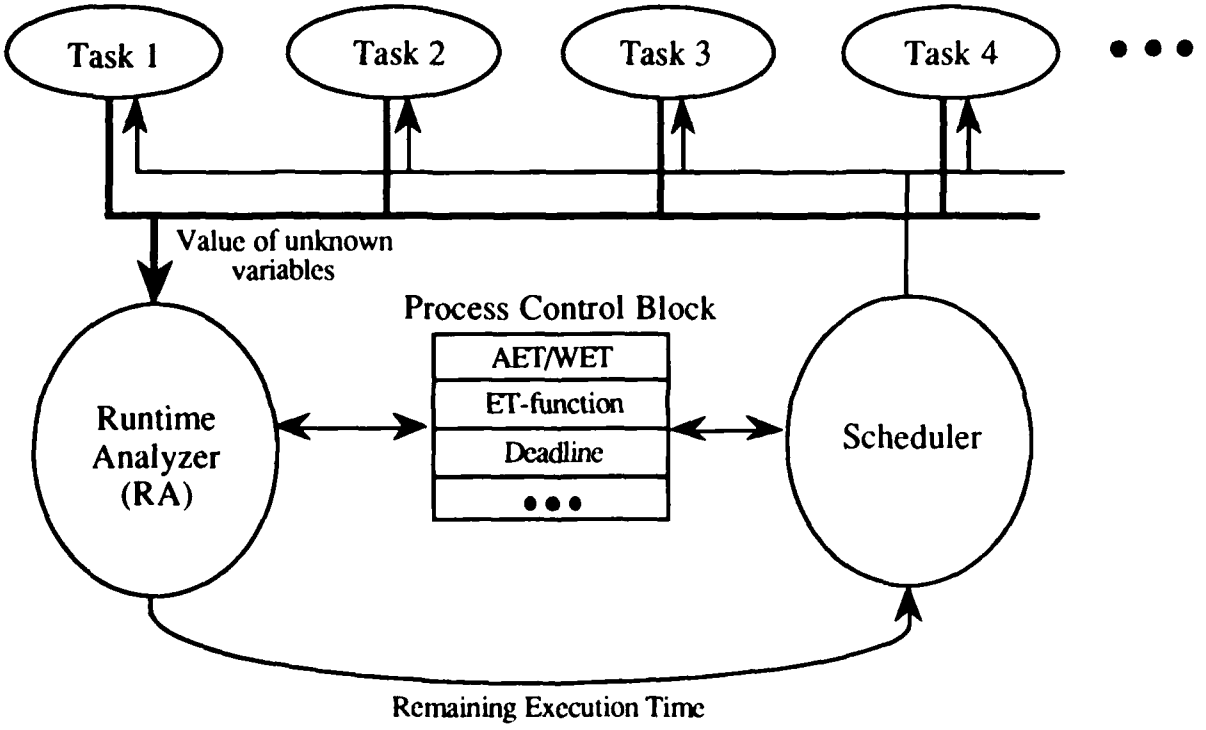


Figure 6. Control & Data Flow among Tasks, RA and Scheduler

```
runtime_analyzer()
{       int n;          /* number of segments */

        when invoked  /* unknown variable of the kth segment is known */
        {       /* RET is the remaining execution time */
                RET = ET_function(value_of_unknown_variable);
                for(i=k+1; i<=n; i++)
                {       /* WET[i] is the worst-case execution time of the ith segment */
                        RET = RET + WET[i];
                }
                send RET to scheduler;
        }
}
```

Figure 7. Runtime Analyzer

4. REAL-TIME TASK SCHEDULING

The real-time task scheduling approaches can be divided into three categories: (1) design-time guarantee, (2) runtime "best-effort", and (3) hybrid of the two approaches. The traditional deadline-driven scheduling algorithms, such as shortest-deadline-first, least-laxity-preemptive and rate-monotonic-preemptive, try to find the conditions for schedulability at design-time or at preruntime. Under such environments, although the accurate execution time is always necessary scheduling based on their runtime estimation is almost meaningless since all tasks are already guaranteed to execute within deadlines. However, this design-time guarantee approach suffers from the following two problems. First, due to the fact that the worst-case execution time is used (and in many cases those are too worst) the processor utilization is usually very poor. Second, and more seriously, as real-time systems become more complicated and distributed, the design-time guarantee is extremely hard if not impossible. Therefore, the second and the third approaches offer more appropriate solution.

One promising scheduling concept under the second approach (and also applicable for the third approach) is based on the "time-value" function which defines a reward value at the completion of a task. Tasks of real-time systems may have different time-dependent value functions based on the criticalness of the tasks. For example, in Figure 8 Task A has reward value V_A if the task is completed by time d_A; otherwise, the reward value becomes zero. On the other hand, Task B has reward value of V_B until d_{B_1}; then the reward value decreases and becomes zero eventually at time d_{B_2}. Task C has a different reward value function as shown in the figure. (In general, we can say that a task with the Task A-type reward function is called a hard-real-time task and a task with the reward function type of Task B or Task C is called a soft-real-time task.) The scheduler should be designed to maximize the reward values and/or

minimize the penalty. Since the DET estimation approach produces very accurate remaining execution time information it makes more effective scheduling possible.

Moreover, typical real-time systems provide a control mechanism based on a three-step process: (1) obtain inputs from the external environments, (2) process inputs based on a control algorithm, and (3) produce control outputs back to the external environments before the deadline. Therefore, once the inputs are given, a more accurate execution time, instead of the worst-case execution time, can be determined by DET estimation. And, as a result, more effective scheduling becomes possible. The following subsections demonstrate two cases where the DET estimation is very useful for real-time task scheduling.

4.1 The Restricted Preemptive Scheduling based on Time-Value Function

The performance criterion of deadline-driven scheduling is to minimize the number of tasks which miss the deadline. In the design time guaranteed case, the number should be zero. However, under time-value-based scheduling, tasks should be scheduled such that the reward value is maximized.

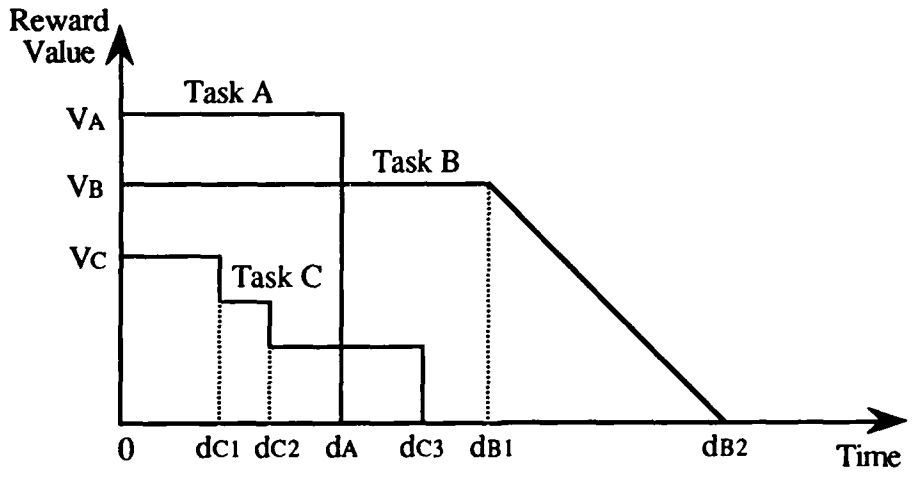


Figure 8. Time-Value Function

Suppose that two tasks, Task A and Task B have time-value functions as shown in Figure 9(a). Both tasks consist of three segments as shown in Figure 9(b). Deadlines are 15 and 10, and the estimated worst-case execution times are 10 (3,5,2) and 7 (2,3,2), respectively, although their actual execution times are 8 (3,3,2) and 7 (2,3,2). Assume that the preemption can be done only at the end of each segment. Task A is scheduled initially since it is not possible to schedule both tasks within deadlines, and Task A has a higher reward value.

After completion of Segment 1 of Task A, the RA reevaluates the remaining execution times, which turn out to be 5 and 7 instead of 7 and 7. Now we can schedule Task B which results in a successful completion of both tasks (Figure 9(c)). This would not be possible without using DET estimation since the scheduler would schedule Task A until it is completed, which would not permit Task B to complete on time (Figure 9(d)).

4.2 Estimation of the Proper Timeout Period

DET estimation can be used to select proper timeout periods at runtime. Suppose two tasks, Task A and Task B, run concurrently on different nodes. They are periodic tasks and receive inputs from the sensors for each execution cycle. At the end of each cycle, Task A sends the result to Task B. Now the question is how to select the proper timeout period of the receive statement in Task B. This is not easy if both tasks receive inputs asynchronously for each cycle. Using DET estimation this problem may be solved by dynamically selecting the timeout period at runtime. That is, upon receiving input from the sensor Task A estimates its execution time and informs Task B. Task B then sets up a timeout period based on that information. By doing this, (1) processor utilization will increase, (2) more effective real-time scheduling is possible, and (3) earliest detection of abnormal behavior of tasks is possible.

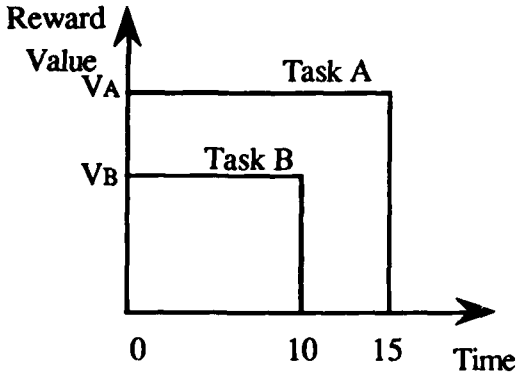
5. CONCLUSION

The execution time behavior of a program is affected by many factors, namely, input values, compiler optimization, resource contention, communication uncertainties, and synchronization overhead. Consequently, techniques that fail to address these factors may not be of much use to the scheduler due to the wide variations in task execution times. This dilemma forces system designers to compensate for uncertainties by tolerating worst-case timing scenarios. However, in real-time systems both under- and over-estimation must be avoided. System failure may result from missed deadline due to under-estimated task execution times and alternatively, over-estimates degrade system utilization and may cause poor performance under time-value-based scheduling.

This chapter has introduced the DET estimation, which is distinguished from other approaches because: **1)** semantic analysis as well as syntactic analysis is done by tracing values of variables of a program using k-flags, and **2)** ET-function is generated. The ET-function makes it possible to calculate more accurate runtime execution times when the values of variable factors are known. The remaining execution times of the tasks produced by DET estimation are very useful to real-time task scheduling because they improve predictability

which supports better resource utilization.

The research reported here is a part of our concurrent effort to develop the Application-Specific Distributed Real-Time Kernel which is designed to support real-time embedded control systems. We are investigating real-time scheduling algorithms which can be benefited by the DET estimation, especially scheduling with the time-value function. The Runtime analyzer (RA) and the scheduling strategies studied along with the Compile-time Analyzer (CA) will be implemented for the kernel.

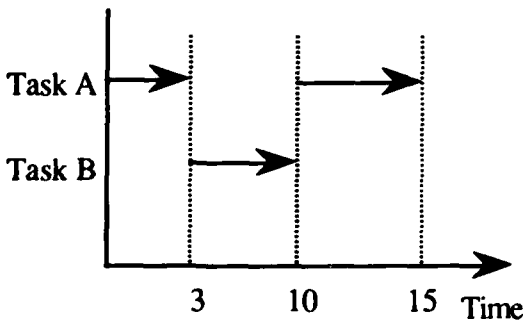


(a) Time-Value function

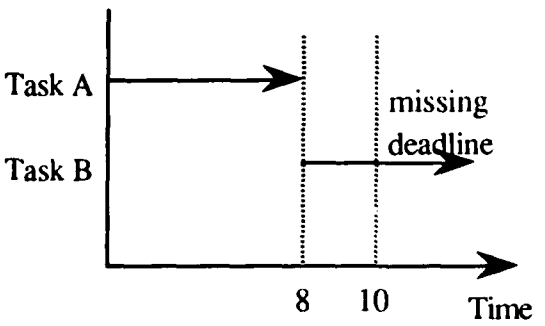
	Task A	Task B
AET_1	3	2
$WET_2(AET_2)$	5(3)	3(3)
$WET_3(AET_3)$	2(2)	2(2)

AET_i : Actual Execution time of segment i
WET_i : Worst-case Execution Time of segment i

(b) Timing Information



(c) Scheduling with DET estimation



(d) Scheduling with only worst-case execution time estimation

Figure 9. Scheduling based on Time-Value function

6. REFERENCES

[Gop90] Gopinath, P., and Gupta, R., "Applying Compiler Techniques to Scheduling in

Real-Time Systems", IEEE Proceedings of Real-Time Systems Symposium, 1990, pp.247-256.

[Hab90] Haban, D., and Shin, K.G., "Application of Real-Time Monitoring to Scheduling Tasks with Random Execution Times", IEEE Transactions on Software Engineering, December 1990, pp.1374-1389.

[Jen85] Jensen, E.D., and Locke, C.D., and Tokuda, H., "A Time Driven Scheduling Model for Real-Time Operating Systems", IEEE Proceedings of Real-Time Systems Symposium, 1985, pp.112-122.

[Liu73] Liu, C.L., and Layland, J.W., "Scheduling Algorithms for Multiprogramming in Hard-real-time Environment", Jour. ACM, January 1973, pp.46-61.

[Mok78] Mok, A.K., and Dertouzos, M.L., "Multiprocessor Scheduling in a Hard-real-time Environment", Proceedings of 7th Texas Conference on Computing Systems, 1978, pp.5.1-5.12.

[Par91] Park, C.Y., and Shaw, A.C., "Experiments with a Program Timing Tool Based on Source-Level Timing Schema", IEEE Computer, May 1991, pp.48-57.

[Pus89] Puschner, P., and Koza, CH., "Calculating the Maximum Execution Time of Real-time Programs", The Journal of Real-time Systems, 1989, pp.159-176.

[Reh87] Rehm, P.H., 'A C Language Cross Development Environment for Real-Time Programming', M.S. Thesis, University of South Florida, 1987.

[Wir76] Wirth, N., 'Algorithm + Data Structures = Programs', Prentice Hall, 1976.

[Wei81] Wei, A.Y-W, 'Real-Time Programming with Fault-Tolerance', Ph.D. Dissertation, University of Illinois at Urbana-Champaign, 1981.

CHAPTER VI

TP/C: A REAL-TIME COMMUNICATION PROTOCOL IN LAN/MAN

1. INTRODUCTION

Communication systems that carry real-time data should assure timely delivery of messages, particularly when the medium is shared by several nodes. Examples are LAN/MAN which carries packetized voices/images and distributed systems such as flight control systems and ballistic missile defense systems. Bus and ring (e.g., Ethernet, token-ring, FDDI ring) are common media of communication in such systems. When dealing with real-time data, the crucial performance measure for medium access protocols (MACs) is the minimization of message loss (viz., not delivering data within given deadlines) rather than the minimization of average delays or maximization of throughput and utilizations. While minimum-laxity-first and minimum-deadline-first policies for scheduling messages are suitable, they require a system-wide knowledge to achieve the optimal performance.

Several MAC protocols have been studied to deal with real-time messages, including CSMA/CD [Chl85], token-passing [Jan87,Kam86], Virtual-Time CSMA [Med86,Mol85] and Window protocols [11]. We propose an alternative protocol, named "token passing with concession (TP/C)", for real-time messages and communication channel scheduling. This can be implemented on top of the existing contention-free protocols such as token bus, token ring or FDDI ring protocol. The protocol seems useful for real-time control applications in distributed computing environments as well as for voice and data communication in LAN/MAN environments. The performance metrics such as percentage of messages lost and effective channel utilization were obtained by simulation under various network and protocol parameters and compared with other protocols. We also discuss how the protocol can be implemented in a hard-real-time kernel.

2. LAN PROTOCOLS FOR HARD-REAL-TIME COMMUNICATION

Need for real-time delivery of messages can be found in many applications. In voice transmission, a small number of lost frames has been shown to have little, if any, effect on human speech intelligibility [Kur84]. Therefore, some frame loss is allowed and is even desirable because it alleviates channel traffic. This is an example of a "loss" system. However, distributed simulation and control systems have more stringent time constraints where message loss is not tolerated [Ram87]. These are "no-loss" systems, i.e., all message are required to arrive within specified intervals. Whether the system is a "loss" system or a "no-loss" system the major performance measure for time-constrained application is, thus, rate of message loss rather than average delay in non-time-constrained applications. Some of the existing MAC protocols may seem suitable for time-constrained communications as well. However, these

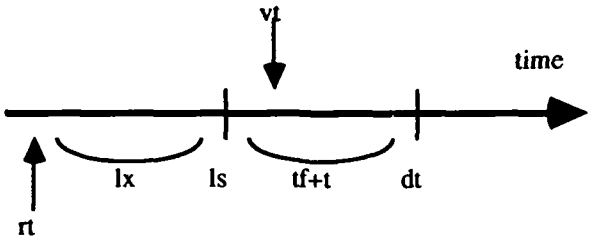
protocols do not consider scheduling of packets on a system-wide basis, and it becomes very difficult to tune the protocol for varying loads.

Virtual Time CSMA (VTCSMA) and Window Protocols have been proposed for broadcast multiple networks with a large number of stations and bursty traffic. They are designed to adapt to the input load. In both cases it is assumed that the time axis is slotted (synchronous mode). The length of a slot is equal to the end-to-end propagation delay, t, which is considered as the unit of time. The message transmission time, tf, is a multiple of the length of a slot, and tf >= 1. A node can transmit a message only at the beginning of a slot. Collision duration, C, is less than or equal to tf. If C < tf then the channel is said to have collision detect capability. The channel can be sensed idle, busy with the transmission of a successful message, or with more than one messages in which case a collision has occurred. Furthermore, a station can "remember" the previous channel state and make decisions based on the previous state.

VTCSMA: This protocol is an extension of CSMA/CD protocol with virtual clock. Each node is equipped with two clocks, a real-time clock, rt, and a virtual-time clock, vt. The real-time clock always runs at unit speed. The virtual-time clock stops running when the channel is busy and resumes running when the channel is subsequently sensed idle. When it runs the virtual-time clock runs at a rate *d* times faster than the real-time clock. A frame is sent only if its ls <= vt, where ls is the the latest time to send a frame (see Figure 1). Each node senses the channel in lock-step. When a node finds that the channel is idle, either after a successful frame transmission or after a collision, the node drops any waiting frame if its ls < rt because it will not meet its deadline. The virtual-time clock is then set to the real-time clock (vt = rt) and begins to run. When a collision occurs during the transmission of a message, the sender node re-transmits its frame immediately with probability p, or modifies the ls of the frame to be a random number from the interval (rt, ls), then the frame is put back in the queue of frames waiting to be transmitted.

Window Protocol: In this protocol each station maintains a window structure. Window position and size depend on the present and past channel state. The lower edge of the window is always the real-time clock maintained by each station. The upper edge is initialized at *d* time units above the lower edge and changes dynamically. At the beginning of a time step messages are transmitted if they are inside the current window; or a message is dropped if ls < rt. Each station keeps a stack and when a collision is detected the current upper edge of the window along with the id of the collided frame are pushed onto the stack. The upper edge of the window is then adjusted to half the previous value. The message in the stack is removed if the upper edge of the window is less than the current time. When the channel is detected idle the

stack is popped up and the previous upper edge of the window is restored, i.e., the window expands. If the stack is empty the window upper edge keeps expanding by *d* units.



rt : current time
vt : virtual time
dt : deadline
ls : latest time to send a frame
lx : laxity time
tf : transmission time
t : propagation delay

Figure 1. Various Times.

It has been shown that the performance of these protocols is better than other protocols such as CSMA/CD. However, they have several drawbacks. Since they are contention-based protocols (1) performance would decrease significantly as the input load becomes higher and (2) it is not easy to predict whether a message would be deliverable or not. Also, the performance would suffer a lot if the global synchronization is not achieved.

3. TP/C PROTOCOL

In this section we describe the "token-passing with concession (TP/C)" protocol which can be implemented on top of protocols such as token bus, token ring or FDDI ring protocol. The TP/C protocol works as follows: A token has a counter which is initially set to zero. The token is passed among nodes in the system and a node which possesses the token is permitted to transmit a fixed size message. The messages are discarded if ls < rt. The node transmits the most urgent (i.e., shortest valid time) message. If the queue contains any message whose validity time would expire before the token returns (we can always calculate the worst-case and the best-case turn around time of the token since each message size is known.), the node increases the counter attached to the token and sets the "urgent" flag before forwarding the token. Then the node forwards the token to the next node. A node skips transmitting a message and simply forwards the token if (1) the node has no message to transmit or (2) the node has only messages whose validity times will not expire before it gets the token in the next turn and the counter, attached to the token, is not zero. That is the node yields its turn to accommodate

more urgent messages in other nodes. Figure 2 shows the pseudo code for TP/C protocol.

```
Constant;                       /* Common to all stations */
        MAXturn = n1;           /* The worst-case token turn-around time,
                                i.e., the token turn-around time when all stations send messages
*/
        MINturn = n2;                   /* The best-case token turn-around time,
                                i.e., the token turn-around time when no station send messages
*/
Initialization;                 /* Within each station */
        urgent = FALSE;
Loop
        Wait for free token;
        IF urgent       {Send the most urgent message;      /* It was set in the previous turn
*/
                        urgent=FALSE;  decrease token_counter by 1}
           ELSE IF the laxity time of the most urgent message <MAXturn
                        {Send the most urgent message}
           ELSE IF token_counter = o
                        {Send the most urgent message}
        IF the laxity time of any remaining message <MINturn
                        {discard these messages}            /* not possible to deliver in time
*/
           ELSE IF  the laxity time of any remaining message <MAXturn
                        {urgent=TRUE;  increase TOKEN COUNTER by 1}
        forward free token to the next station;
Forever;
```

Figure 2. The TP/C protocol.

4. PERFORMANCE ANALYSIS

4.1 Simulation Model and Parameters.

Discrete-event system-level simulation [Mac87] is used. Events are identified by type and time of occurrence. They are kept in a list sorted by time. On every simulation step the next event is retrieved from the list, the simulation clock is advanced to reflect the event's time, and finally the event is deleted from the list. Relevant events are identified for each protocol. Simulation parameters include network parameters, input load parameters, protocol-dependent parameters and statistics collection parameters.

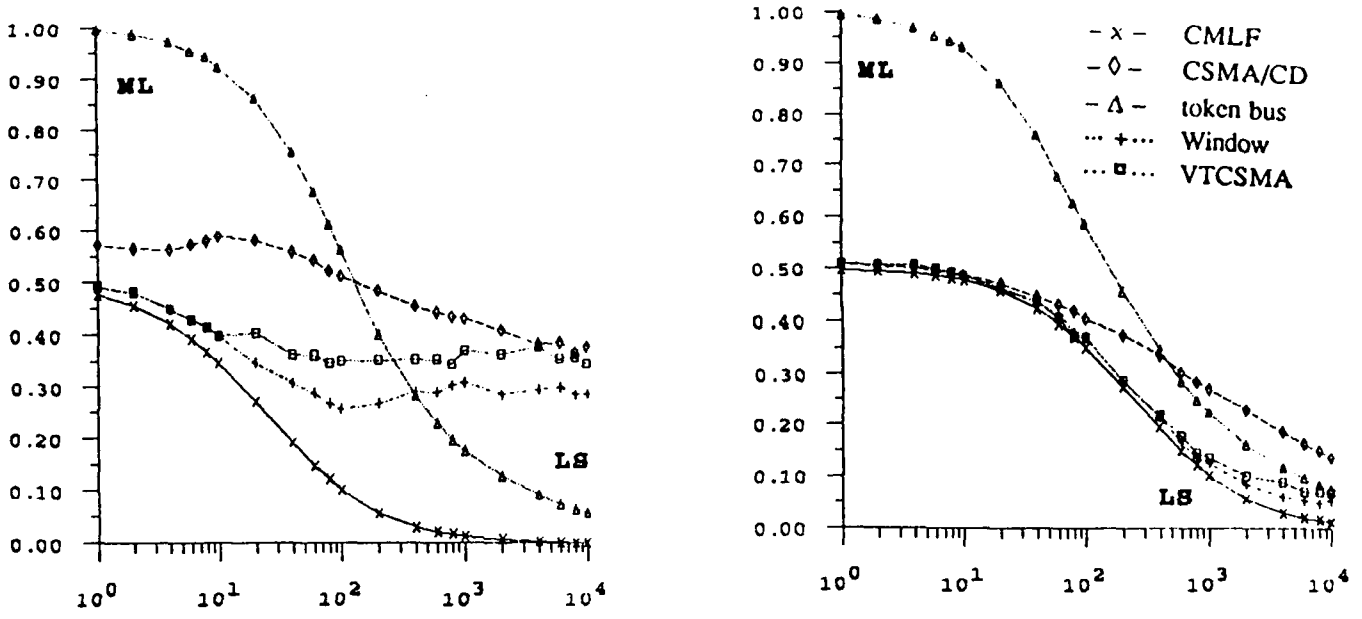
(1) Network parameters are protocol type, N (number of stations), CL (cable length in kilometers), R (data rate in Mbps), t (maximum end-to-end propagation delay) and a (normalized propagation delay which is defined as the ratio of the end-to-end channel propagation delay to the packet transmission time). Most of the analysis is performed for a = 0.01 and a = 0.1. For a 5 Km and 10Mbps typical network this corresponds to frame

transmission times between 250 and 2500 usec or frame size between 2500bits and 25Kbits.
(2) Incoming frames are assigned at random to the queues attached to the stations. In the infinite population case a new frame is assigned to an inactive station (whose queue is empty). Frames that do not meet the given deadlines are discarded. Input load parameters are message length distribution (exponential, normal or constant), tf (frame transmission time), ti (frame inter-arrival time), g (load intensity or g = tf/ti), ls (latest time to send a frame) and the distribution of a message deadline (exponential, normal or constant).
(3) Protocol-dependent parameters include d (virtual time rate for VTCSMA case or window size for Window protocol case), p (re-transmission probability), C (collision detection time) and THT (token holding time). It is assumed that all local clocks are well synchronized or there is a system-wide time reference that "ticks" every time unit. Collision detection takes one time unit (C = 1), i.e., idealized synchronization.
(4) The "Batch Means Method" is used to gather statistics on the measures of interest [Mac87]. A simulation run is divided into batches and the performance measures are calculated for each batch consisting of 2000 frames. The measured data are averaged and a confidence interval is calculated. The simulation will run either until the confidence level (95%) is reached or until the number of batches collected reaches the maximum number (50) whichever comes first.

4.2 Comparative Analysis

Performance Measures: The three most commonly used measures in evaluating the performance of LANs are information throughput, channel utilization, and delay [Abe91]. In this study we have measured the rate of message loss (ML) and other performance indicators such as (1) D (normalized average frame delay) which is defined as the time from a frame's arrival time at a station until the end of successful transmission, (2) ECU (Effective Channel Utilization), (3) CCU (Collision Channel Utilization) and (4) NTL (Normalized average Transmitted frame Length) which is the average length of transmitted frames divided by the average length of arrived frames.
Centralized Minimum Laxity First Scheduler: The Centralized Scheduler can be represented by a single queue of messages. There is only one server and messages are served on a first-come-first-served basis if there is no timing constraint associated. Since no time is wasted in collision resolution, queueing delay is thus minimized. In the time-constrained case messages are ordered by their ls's, and served accordingly. Therefore, the message loss is minimized. The scheduler is known as Centralized Minimum Laxity First (CMLF) scheduler. The CMLF scheduler is not realizable in practice and is used to provide an upper bound on performance [Zha90].

Comparison: Figures 3a and 3b show ML versus lx for the protocol studied. The curves were obtained for N=200, a=0.01 and 0.1, input load 1.0. Protocol parameters for VTCSMA and Window are selected to minimize the ML whereas no attempt made to optimize the performance of CSMA/CD and token bus. The curve for the CMLF protocol is the ideal lower bound for ML. The window protocol performed very close to CMLF especially at low values of ls. Virtual time performs close to the window protocol most of the time. At low values of ls the CSMA protocol performs very well, but its performance deteriorates for higher values of ls and a. The token protocol performs better than the others as the input load increases and for large values of ls. (Curves for TP/C protocol are not included. However, the preliminary results show that it performs better than ordinary token bus, particularly when lx is small.)



(a) Case 1: N=200, g=1.0, and a=0.1 (b) Case 2: N=200, g=1.0, and a=0.01

Figure 3. Performance comparison.

From the study we can draw the following conclusions:

(1) ML in the CSMA/CD protocol is very insensitive to N at low values of a, however, this degrades as a increases, specially at high g values. ML deteriorates in the token bus protocol as N increases. Window and VTCSMA were less sensitive to N over a wider range. (Here, the comparison is made with the fixed input load regardless of the value of N.)

(2) The Window protocol performed very close to CMLF especially at low values of ls. VTCSMA performs close to the Window protocol most of the time. At low values of ls CSMA/CD performs very well, but its performance deteriorates for higher values of ls and a.

(3) The token protocol performed better than the others for large values of g and ls. However, for smaller values of ls ML increased rapidly. At large input loads the token overhead became

negligible compared with the time spent transmitting data frames. This protocol also outperformed the others for very large values of a. This is due to its collision-free nature. This protocol may be more suitable than the others for situations involving high data rates, such as in fiber optics networks, or very long distances.

(4) All protocols were equally fair in transmitting variable length frames under all conditions, i.e., NTL is close to 1.

(5) The Window protocol was much less sensitive to protocol parameters (d and p) than VTCSMA. This makes the protocol implementation simpler and easier to maintain. In a real implementation, an issue to consider is the synchronization of the clocks required by these two protocols. Imperfect synchronization will cause a degradation in performance.

(6) Overall, the Window and VTCSMA protocols performed better than CSMA/CD and token bus over a wider range of input parameters.

5. IMPLEMENTATION ENVIRONMENT

The study of real-time communication protocols is part of our on-going effort to investigate techniques and methods for development of ultra-reliable real-time systems. The motivation is the increasing reliance on computer systems for control of time and safety-critical applications. The major research tasks include system-level fault tolerance, reliability modeling and scheduling of real-time tasks and messages. Also, a distributed real-time system testbed, named Simplified Unmanned Vehicle System (SUVS), [Yan91] has been developed for experimental research. Our target distributed real-time system consists of autonomous computing nodes which communicate through a network. Each node is composed of three large layers which are the Network layer, the System layer and the Application layer: (1) The Network layer provides communication between tasks in different nodes. The TP/C protocol and other real-time protocols will be implemented in this layer.

(2) The System layer is the layer which provides the functions of the OSI (Open Systems Interconnection) transport and session layers as well as the functions of traditional operating systems. Since it is not our intention to develop a general purpose DOS, the design issues should be different from the ordinary DOS design. We focus on three major functions which are real-time message handling, task scheduling and fault tolerance. (3) The Application layer consists of a set of tasks which communicate to each other through message passing mechanism. Software fault tolerance techniques such as the recovery block scheme, N-version programming and the conversation scheme can be incorporated. Figure 4 depicts how messages are handled within each node.

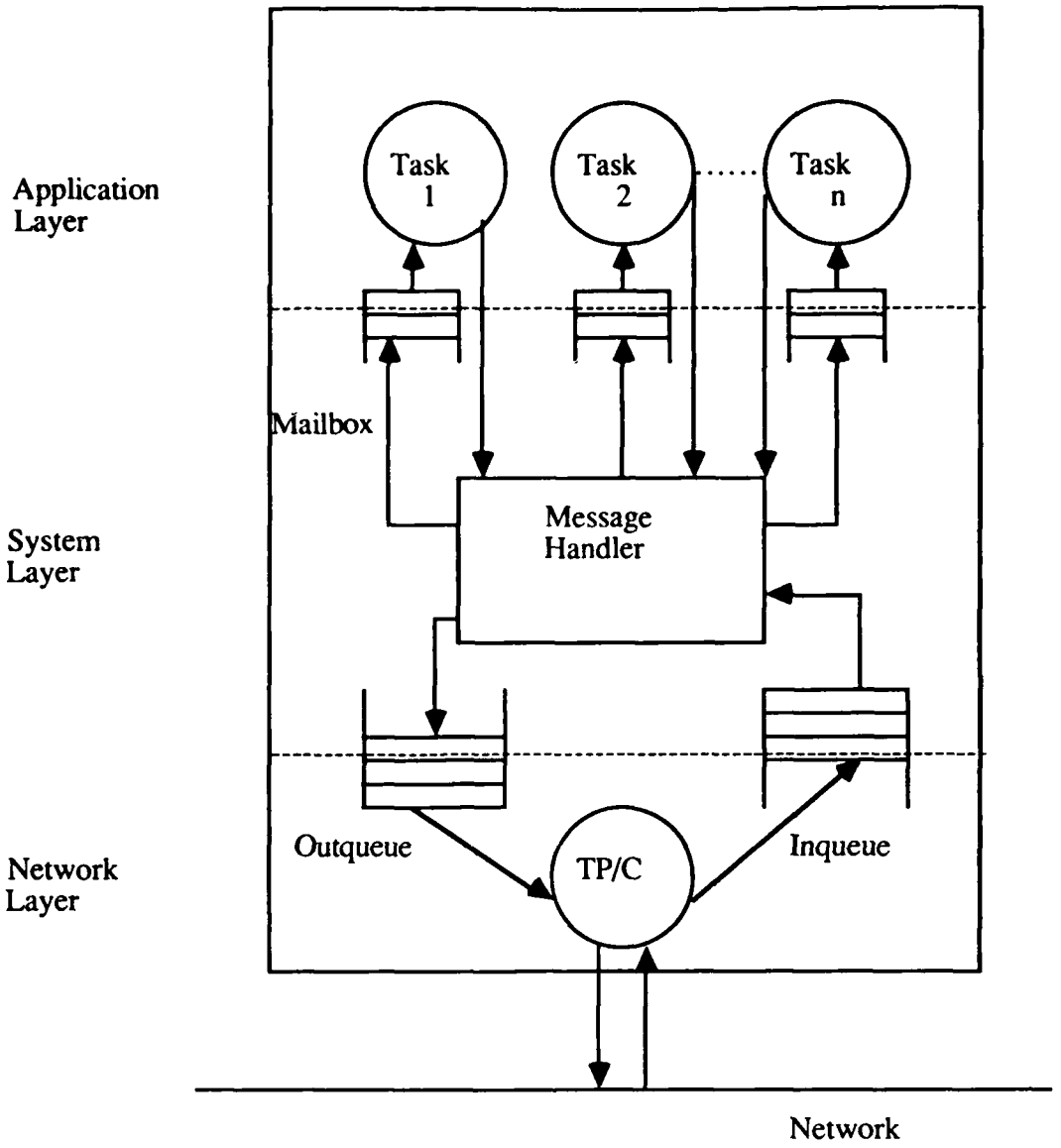


Figure 4. Messages flow in three layers.

6. CONCLUSION

In this chapter we proposed a hard-real-time communication protocol, named Token-Passing with Concession (TP/C) protocol, and analyze the performance. We compare the performance with other protocols based on simulation. A discrete-event simulation technique was used to model the network. Large amounts of data were collected and plotted to verify the performance of the protocols under different input load and network and protocol parameters. The result shows that the TP/C protocol performs well, especially when input load is high and is suitable for very fast networks such as fiber-optic ring network. The TP/C protocol is more predictable than VTCSMA and Window protocol in the sense that the message scheduling is

more deterministic. This is because the TP/C protocol does not require back-off due to collision. Another advantage of the TP/C protocol is that it is less affected by loose synchronization among clocks in the system. We will analyze the effect of non-synchronous clocks. We also plan to analyze the performance of the TP/C on ring networks (i.e., as an extension of token ring and FDDI ring protocols.)

7. REFERENCES

[Abe91] B.W. Abeysundara and A.E. Kamal, "High-speed local area networks and their performance: a survey", ACM Computing Survey, June 1991, pp. 221-264.

[Chl85] I. Chlamtac, "An ETHERNET compatible protocol for real-time voice/data integration", Computer Networks and ISDN Systems, 10, 1985, pp. 81-96.

[Jan87] D. Janetzky and K.S. Watson, "Performance evaluation of the MAP token bus in real time applications", Advances in Local Area Networks, IEEE Press, New York, 1987, 383 - 410.

[Kam86] M. Kaminski,"Protocols for communicating in the factory",IEEE Spectrum, pp. 56-62, Apr. 1986.

[Kur84] J.F. Kurose, M. Schwartz, Y. Yemini, "Multiple-access protocols and time-constrained communication", ACM Comput. Surv., vol. 16, no. 1, pp. 43-70, Mar. 1984.

[Mac87] M.H. MacDougall, "Simulating computer systems. Techniques and tools", The MIT press. Cambridge, 1987.

[Med86] J. S. Meditch, D. H. Yin, "Performance analysis of Virtual Time CSMA", Proc. of IEEE Infocom '86, April 1986.

[Mol85] M. L. Molle and L. Kleinrock, "Virtual Time CSMA : Why two clocks are better than one", IEEE Transactions on Communications, Vol. COM-33, No. 9, Sept. 1985.

[Ram87] K. Ramamritham, "Channel Characteristics in Local Area Hard Real-Time systems", Comput. Networks and ISDN Syst., 13, 1987, pp. 3-13.

[Yan91] S.M. Yang, et al., "SUVS: A Distributed Real-Time System Testbed for Fault-Tolerant Computing", 1991.

[Zha90] W. Zhao, et al., "A window protocol for transmission of time constrained messages", IEEE TC, Sep. 1990.

CHAPTER VII

SUVS: A DISTRIBUTED REAL-TIME TESTBED FOR FAULT-TOLERANT COMPUTING

1. INTRODUCTION

Computer systems are increasingly being used for complex and sophisticated real-time applications such as flight control systems and ballistic missile defense systems. In such time critical applications correctness (i.e., both logical and temporal characteristics) of the system behavior should be rigorously validated. Although various formal (or analytic) verification/validation methods have been proposed [Hoa85,Jah86,Mil80,Ram89] there is a limit to proving the correctness of the system behavior formally. The situation becomes worse in a distributed computing environment where concurrent tasks/processes communicate through a communication network. Therefore, experimental validation techniques are highly desirable in such systems [Avi85,Bha89, Chu87,IEE86,Kim88a, Koh86,Kop89, McD82].

The importance of experimental research (or experimental validation and evaluation of techniques/methodologies) has been established by the distributed computer systems research community, especially in a real-time environment [DCS89]. As pointed out in [Chu87], testbed-based evaluation provides more accurate results than software simulation because testbeds can be configured to represent the operating environments and input scenario more accurately than simulation.

Researchers at the University of Texas at Arlington (UTA) have formulated a distributed system testbed to support experimental research at UTA. The testbed, named SUVS (Simplified Unmanned Vehicle System), is being used to conduct experimental evaluation of techniques and methods for design of reliable distributed real-time systems. More specifically, we plan to experiment with various fault tolerance techniques such as the conversation scheme [Ran75] and the voting scheme. As part of the research effort to convert a concept into a practical technology, we demonstrate how multiple versions of software can be systematically produced. We also demonstrate how the acceptability routine can be realized in a "real" application. The testbed will also be utilized for clinical study of specifications, design and implementation methods studied at UTA for development of real-time distributed and parallel systems.

The software part of SUVS consists of a set of sensor tasks, analyzer tasks, and actuator tasks. The first version of SUVS was implemented using Verdix Ada (version 5.5) on a Micro Vax 3900/Ultrix. The second version, reported here is written in C, runs on a network of eight SUN Workstations, and X-windows is used to provide a graphic interface. The fault-tolerant SUVS (FT-SUVS) is then implemented. The target architecture we have simulated for FT-SUVS is a hybrid parallel architecture (i.e, a modified hypercube), named the Hybrid Higher Radix Hypercube (HHRH). Processor nodes are partitioned into clusters. Nodes in a common cluster communicate through shared memory, while communication with nodes in

other clusters is through point-to-point connections. This approach provides several advantages: (1) short diameter and average node distance which makes communication between nodes fast and predictable (speed and predictability are the most important factors in real-time systems), (2) redundancy which is essential for fault-tolerant computing, and (3) reconfigurability as required by many applications.

This chapter reports the implementation details of SUVS and FT-SUVS and the results of the first phase of experimentation with FT-SUVS. The conversation scheme was chosen as a major fault tolerance scheme in FT-SUVS mainly because cooperation among tasks for error detection and recovery is needed. We have also implemented and experimented with the voting scheme. Various faults such as node crash, link failure, and software design errors are simulated and injected for the experiment. How these faults are detected, the system's recovery mechanisms, and the response time under various situations are described. We also demonstrate how the voting scheme along with other fault tolerance schemes (i.e., comparison scheme) can be incorporated with the conversation scheme in the second phase planned for FT-SUVS.

Section 2 describes the functional and temporal requirements, as well as the design and implementation of SUVS on the SUN network. In Section 3, FT-SUVS is introduced: the HHRH architecture for FT-SUVS and the system level error detection and recovery schemes incorporated into FT-SUVS. Section 4 discusses FT-SUVS implementation details. The experimental results on the first phase of FT-SUVS, as well as the plan for the second phase experiment are discussed in Section 5. Section 6 is a summary discussion.

2. SUVS (SIMPLIFIED UNMANNED VEHICLE SYSTEM)

Real-time systems cover various applications: from factory automation to nuclear power plant, from automobile engine control to space shuttle and aircraft avionics, and from robotics to command-and-control systems. These real-time applications share much commonality. Typically, a set of sensors acquire real-time data from other systems (i.e., the outside world). The sensor data is processed and the output is sent to a set of actuators which then respond to the outside world under strict timing constraints. Nevertheless, each application (or group of applications) has its own distinctive features. Some applications require more rigorous verification of logic and timing correctness depending on the criticality of the mission scenario. For example, some scenarios require tight synchronization among the sensors and actuators, some require tightly coupled communication whereas other may operate in loosely coupled environments. Therefore, the scenarios of real-time system testbeds must be selected such that they cover (and closely match) the characteristics of the target applications of interest.

Otherwise, the methods and results obtained for one application may not be directly applicable to other applications. In this section we describe the functional and temporal requirement of the SUVS. The design and implementation details of the SUVS are also described here.

2.1 SUVS: A Real-Time System Scenario

SUVS (Simplified Unmanned Vehicle System) is a real-time control system scenario implemented for our testbed. SUVS controls a vehicle with no assistance from the human driver. It periodically receives data from sensor readings such as speedometer, temperature sensor, direction sensor, etc., and reacts to the system (i.e., the vehicle) by generating appropriate signals to the actuator devices such as accelerator, brake, steering wheels, etc. Decisions are made based upon the current inputs from the sensors and the current status of the road and the vehicle. The decisions (or the response) must be made within a specified time.

SUVS was chosen as our testbed mainly because it has many interesting features. These features include (1) a hard-real-time system scenario, (2) a ultra reliable computing scenario, (3) interaction and cooperation among analyzer tasks (i.e., the decision makers), and (4) both tightly coupled and loosely coupled communication alternative scenarios.

2.2 Functional Decomposition and Timing Constraints in SUVS

The SUVS consists of three different sets of tasks, i.e., sensor tasks (or sensors), analyzer tasks (or analyzers) and actuator tasks (or actuators) as follows:

(1) Sensors are input devices which periodically (every 100 msec in this experiment) provide data to the analyzer tasks. Speedometer, engine temperature, direction indicator, vision and surface sensors are implemented.

(2) Analyzers make decisions based on the sensor data and the current status of the road and the vehicle. They exchange information among themselves. The decisions are forwarded to the actuators. Speed, direction, vision and surface analyzers are implemented.

(3) Actuators are output devices which control movement of the unmanned system. They receive decisions from the analyzers. Actuators for the brake, accelerator, steering handle and vision camera are implemented for the SUVS.

The information flow among these tasks are shown in Figure 1.

The SUVS is a hard-real-time system where the response time should be guaranteed. The proper action or decision must be made within a given deadline (200 msec in this experiment). Therefore, one of our major objectives in this experiment was to demonstrate that

the system meets this time requirement even under various fault conditions in both the hardware and the software.

2.3 Design and Implementation

The first version of the SUVS, as described above, runs on a Micro VAX 3900/Ultrix and consists of two procedures, one for main and one for "make decisions", and twenty additional concurrent tasks, i.e., five sensor tasks, four analyzer tasks, four actuator tasks and seven buffer

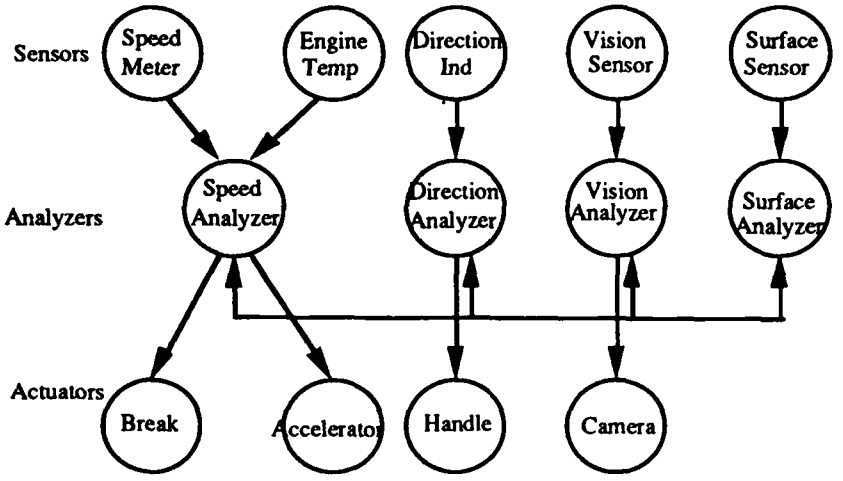


Figure 1. Information flow of the Simplified Unmanned Vehicle System (SUVS).

tasks. Future research efforts will investigate the suitability of Ada for the implementation of hard-real-time systems running on a distributed/parallel environment.

The second version (i.e., C version) runs on a network of eight SUN workstations. We chose a SUN network mainly because we believe that both the loosely coupled and the tightly coupled environments can be easily simulated using sockets, pipes, and shared memory. Also, due to the popularity of the Unix/C environment, the SUVS software is portable. And, in fact, we plan to run analyzer tasks of SUVS on a Sequent parallel computer system.

Figure 2 shows the system configuration: One node (i.e.,workstation) runs all sensor tasks and one node runs all actuator tasks; Four nodes are occupied by four analyzer tasks; Two extra nodes are used for the environment simulator and the graphics interface. The

environment node simulates the road status (i.e., curves and surface condition). The road status is determined by a given scenario. The environment simulator receives inputs from the actuators (e.g.., brake for deceleration, acceleration from the accelerator, etc.) which determines the status of the vehicle (i.e., speed, position and angle of camera view, etc.). Thus, the sensors get data from the environment simulator whereas the actuators output data to the environment simulator.

Initially the system starts by giving a scenario number to the environment simulator node (there are several scenarios with different road configurations). The graphics node draws the road and the vehicle using the information provided by the environment simulator node. This node also displays the gauges such as speed and engine temperature. A separate graphics node is used because there is a time overhead that might affect the real-time execution of the environment simulator node. The X-windows graphic interface is used for this implementation.

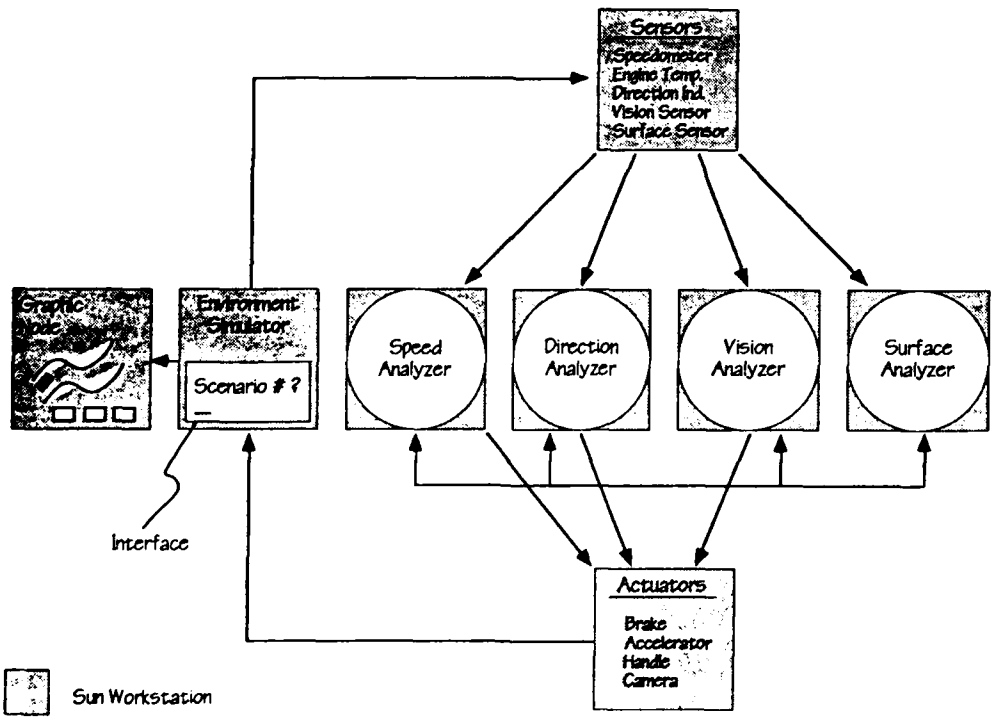


Figure 2. SUVS testbed configuration.

3. FAULT-TOLERANT SUVS

This section describes the target architecture we propose for fault-tolerant SUVS (FT-

SUVS). We also discuss the fault tolerance schemes incorporated in FT-SUVS along with a proposed conversation structure for FT-SUVS.

3.1. The Hybrid Higher Radix Hypercube (HHRH) Architecture

The architecture we have proposed for this testbed is a hybrid parallel architecture, named Hybrid Higher Radix Hypercube (HHRH). The architecture and a possible implementation of the architecture with the description of the system components are given. The performance of the HHRH architecture is briefly analyzed.

Binary vs. Higher Radix Hypercube

The Hypercube structure can be visualized as a cube of any dimension with a node at each corner. Each node typically has its own processing unit, local memory, a communication processor, an optional floating point processor, a kernel of the operating system and the application program. At the lowest level of the hypercube family is the binary hypercube, for which the relation:

$N = 2^n$ where N = number of nodes and n = dimension of the hypercube.

The proposed hybrid architecture is partially obtained by increasing the radix beyond 2. In the resulting topology we examined the benefits (if any) that are obtained as a result of radix enhancement. For the higher radix hypercube (HRH) [Bhu84] the following relation is true:

$N = r^n$ where N = number of nodes, n = dimension of the hypercube, and r = radix

The radix of the hypercube affects the parallel system characteristic parameters, such as diameter, degree of node, average node distance, and message traffic density.

Diameter $= \log_r N = n$

Degree of node $= n*(r - 1)$

Average Node Distance (AND) $= \Sigma \{d * (r-1)^d * C(^n_d)\} / (N - 1)$

Message Traffic Density (MTD) $= 2 * (AND) / \{n*(r - 1)\}$

The Hybrid Higher Radix Hypercube (HHRH) Architecture

The major advantage of using the hypercube structure for designing fault-tolerant real-time systems is the inherent redundancy provided in processors and communication links. However, as the number of nodes increases, the distance between nodes (i.e., the number of intermediate nodes between two communicating nodes) becomes longer. This long distance not only increases the message traffic among the nodes but, more importantly, makes accurate prediction of the system behavior and performance difficult. The HRH architecture remedies these weaknesses to some degree with certain limitations.

On the other hand, the shared memory bus architecture seems to be more predictable in its behavior and performance. However, it suffers from two problems: (1) the bus is a single

failure point, and (2) the bus is a potential communication bottleneck. Therefore, a shared memory architecture is not suitable for some real-time applications which require ultra reliable computing and/or high communication traffic among nodes.

The HHRH architecture remedies these problems by combining the hypercube structure with shared memory units. The shared memory units not only provide high bandwidth to nodes within clusters but reduce the distance between nodes in different clusters. Figure 3 illustrates a simplified diagram for a sixteen-node HHRH obtained by grouping four nodes into a cluster from an HRH with N =16, r = 4. The components of Figure 3 include the nodes (00, 01, ... 32, 33), the cluster memory units (CM0, CM1, CM2, CM3), and the links between the nodes and memory units. Nodes that are directly connected to the same cluster memory unit are grouped as PC0, PC1, PC2 or PC3. The overall topology between the processor clusters is that of an HRH, while the grouping of the processor nodes of the HRH architecture into clusters makes it hybrid. The grouping is accomplished by collapsing relevant bit positions. The nodes that have identical non-collapsed bits, irrespective of the collapsed bit values, belong to the same processor cluster and share the same cluster memory. In Figure 3, for example, nodes 00, 01, 02, and 03 belong to the processor cluster PC0 and share the cluster memory CM0. Therefore, the HHRH architecture can be represented with four parameters: HHRH[N, n, r, x] where N = number of nodes, n = dimension of the hypercube, r = radix, and x = number of collapsed bit. Each cluster consists of 2^x nodes.

The major components of the HHRH architecture, in addition to the underlying network, include the processor node, cluster memory and the system manager.

(1) Processor Node: As mentioned earlier, the node at each corner of the cube consists of the main CPU, a communication processor, a floating point processor and local shared memory. An internal bus interconnects these devices. The communication processor handles all message requests that require routing to other nodes as well as the cluster memory. The I/O channels are connected via links to neighboring nodes in other processor cluster groups, and one channel connects to the memory unit shared by all processor nodes in its cluster. Figure 4 is a typical node processor layout.

(2) Cluster Memory: The cluster memory is a high speed memory unit that consists of 2 sub-devices: the message handler and the memory mailbox. Figure 5 is a block diagram of the cluster memory shared by p node processors. The memory mailbox is a passive device that stores messages at appropriate locations. These predesignated location addresses are stored in a lookup table in the message handler. Each node has (p-1) slots to post messages to all the p possible nodes that share the memory unit. Thus, a total of p*(p-1) slots exist. The message handler is a dedicated unit for handling message requests for the cluster memory. The

handler's hardware includes a processing unit, a message request buffer and I/O channels that connect to the member nodes. Figure 6 shows the main components in the message handler unit. An additional lookup table provides information on the read/write status for the message slots. The message request buffer is accessed by the processing unit on a FIFO basis. The detailed execution of the message handling is discussed in [Bha88a].

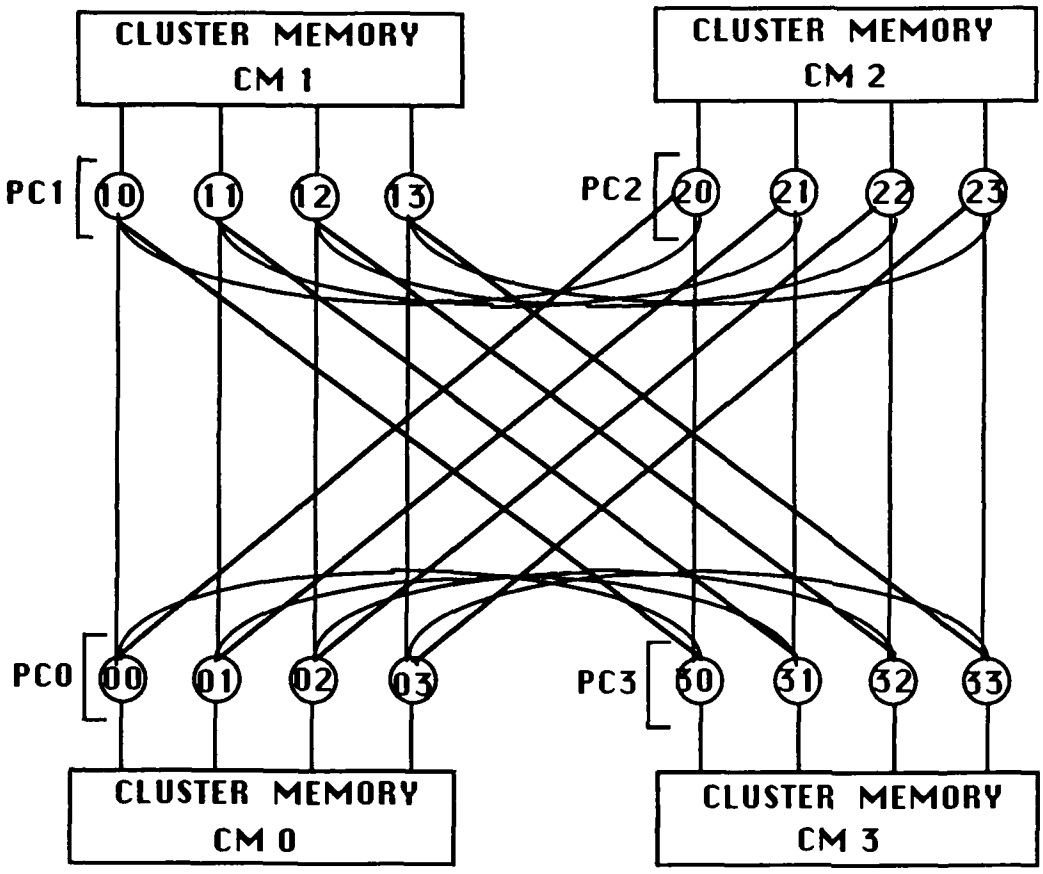


Figure 3. The HHRH with N=16 and r=4.

Performance Analysis of the HHRH

The analysis of the HHRH necessitates redefining certain terms because the introduction of memory units must be accounted for. In the network analysis the following is obtained [Bha88a].

Hybrid Diameter (HD) = $(\log_r N) - x + 1$ memory traversal

Degree of node = $n*(r-1) - 2^x + 2$

Hybrid Average Node Distance (HAND)

$$= [1*\{(r^x-1) + C(^{n-x}{}_1)*(r-1)^1\} +$$
$$2*\{(r^x-1) + C(^{n-x}{}_2)*(r-1)^1 + C(^{n-x}{}_2)*(r-1)^2\} +$$
$$....$$
$$(n-x+1)*\{(r^x-1) + C(^{n-x}{}_{n-x})*(r-1)^{n-x}\}]/(N-1)$$

Hybrid Message Traffic Density (HMTD) = $\{2*HAND\}/\{(r-1)*(n-x)+(r^x-1)\}$

The fault tolerance capability, in terms of the number of disjoint paths, is found to be $n*(r-1)$. This is the same as that obtained for a regular HRH, but is an improvement over the binary hypercube. It should be noted that the reliability of the paths is now greatly dependent on the reliability of the memory units.

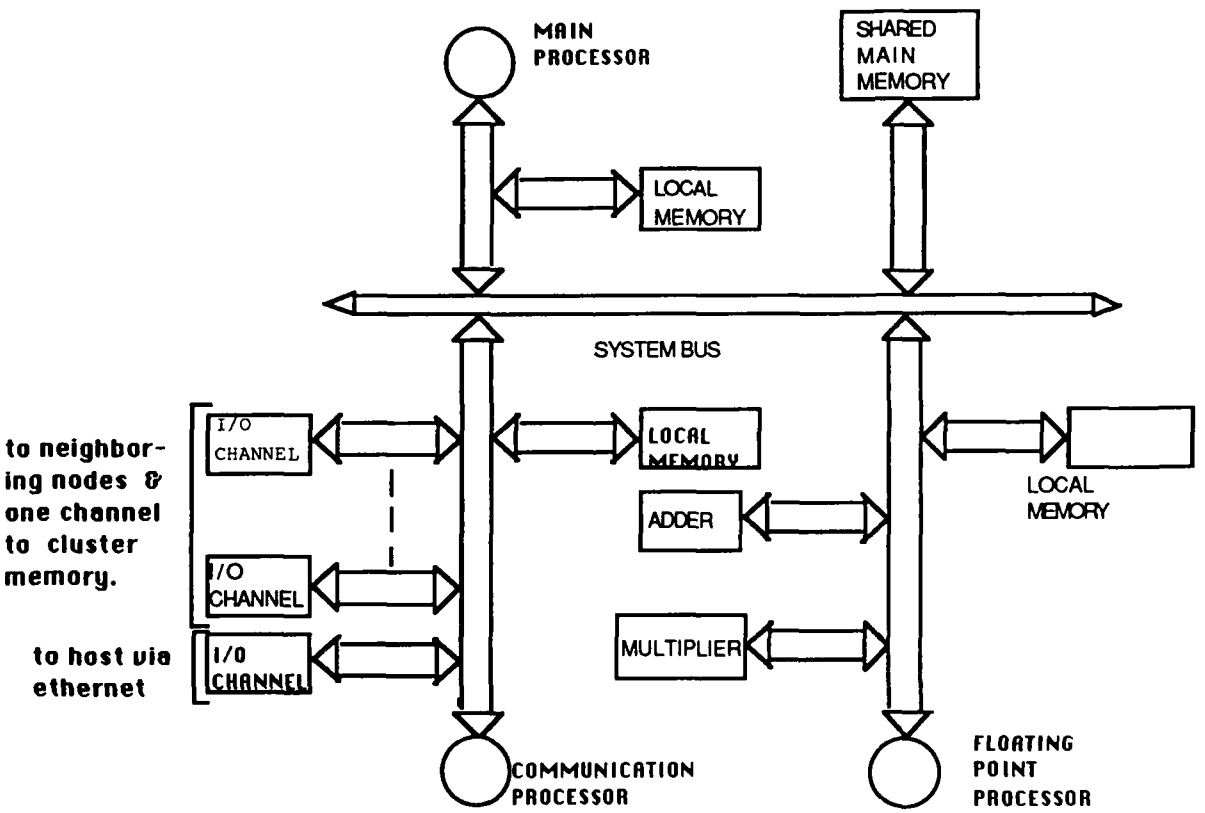


Figure 4. Node processor layout.

(3) System Manager: This device is connected to the nodes in the network via a global communication channel such as the Ethernet. The system manager has varied functions. It serves as the administrative console and also acts as the gateway to the hypercube. It supports a program development system that includes compilers, simulators and vector tools that users can access. In addition, the system manager is able to download data/instructions to the processors in the network in a short span of time.

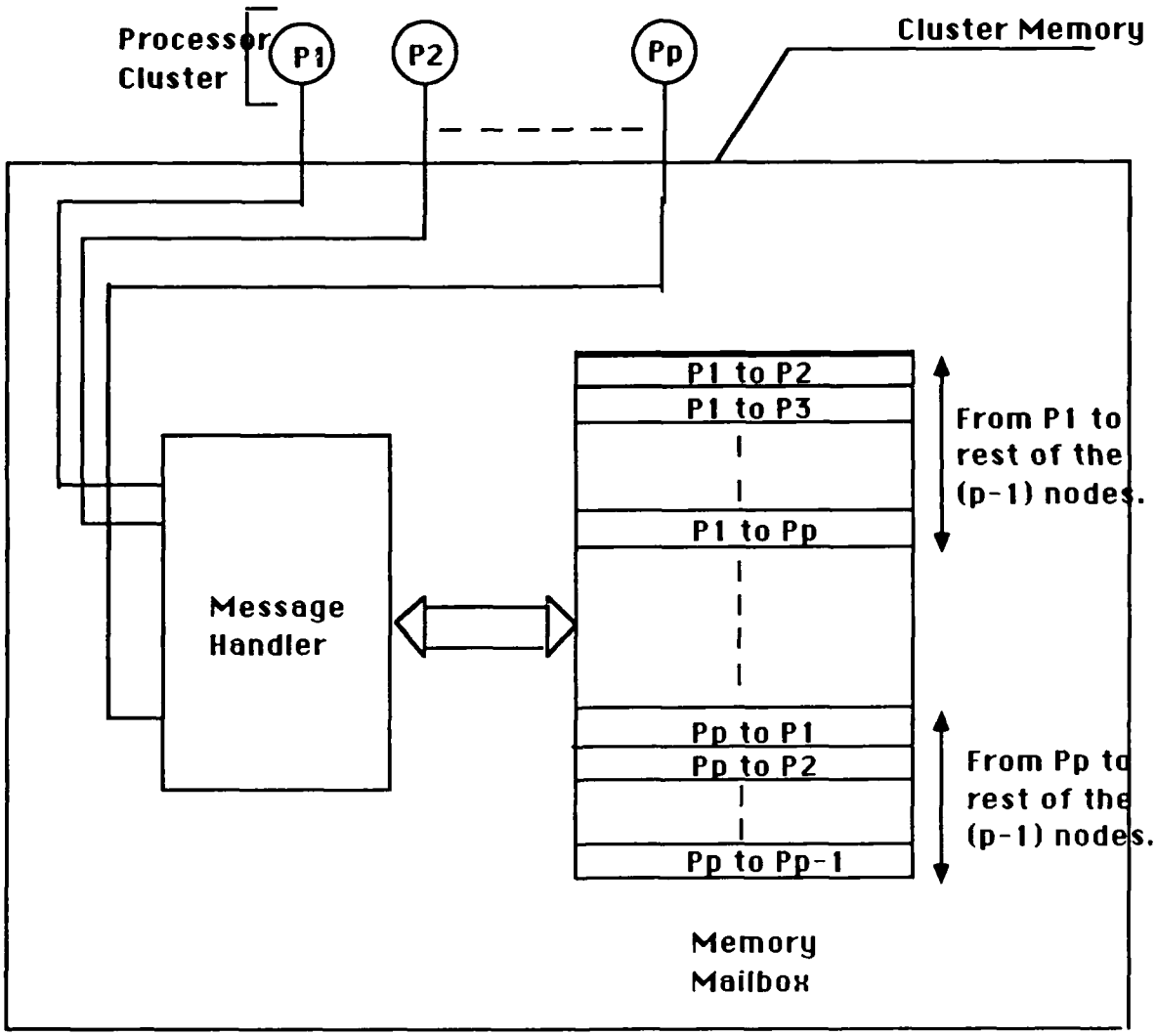


Figure 5. Cluster memory layout.

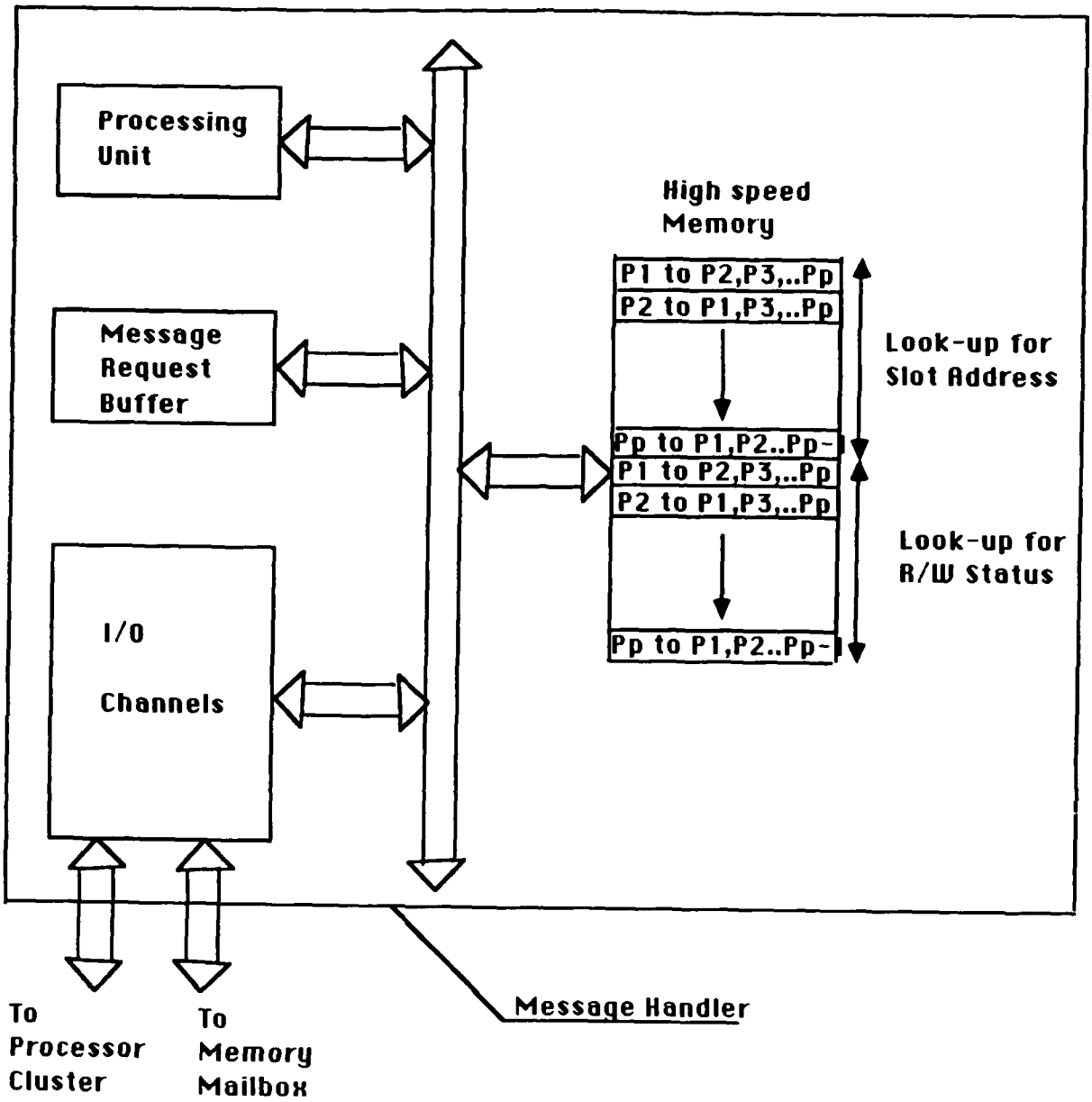


Figure 6. Memory handler layout.

3.2 Fault Tolerance Schemes for SUVS

System (as well as software) reliability is achieved by fault avoidance, fault removal, and fault tolerance [Mus90]. Fault tolerance is based on redundancy. Software redundancy (as well as hardware redundancy) can be incorporated in two different forms: (1) multiple

identical copies of processes (or tasks) and (2) multiple versions of processes. Many fault tolerance schemes have been proposed: they include check pointing, recovery block, triple modular redundancy, conversation scheme, PTC (Programmer Transparent Coordination) scheme, MVP (multiple version programming), DRB (Distributed Recovery Block), and more [Kim85]. These schemes can be classified into two groups based on how the faults are detected and recovered.

Fault detection/recovery schemes

In real-time systems fault refers to, not only errors in logic, but also to failures to respond in time (i.e., missed deadlines). Therefore, timeout should be incorporated as a basic form of error detection mechanism. In addition, there are largely two different ways to detect errors: by approval and by consensus.

(1) Approval can be implemented as an acceptability test (i.e., acceptability test on produced results, environment or both). Some acceptability test criteria are application dependent, whereas some are application independent. Examples of the latter criteria include common exceptions (such as overflow, underflow, divide by zero, etc.), commonly used functions (such as sorting, searching, square root, etc.), and auditing of the environment. Recovery block, conversation scheme, and DRB schemes are based on acceptability tests.

(2) Concensus can be implemented either as a bit-by-bit comparison (called a simple voting) or as a comparison with a certain bound (called a bounded comparison). A simple voting is easier to implement and is application independent. However, it may not be used for systems with multiple versions because there may be a slight discrepancy in the results that are actually correct within acceptable limits (e.g., due to round-offs floating point calculations, etc.) Therefore, in most cases, a simple voting is not applicable to multi-version software systems. MVP is based on concensus.

Independent Recovery vs. Cooperative Recovery

In the previous subsection we classified fault tolerance schemes based on how the errors are detected (and also recovered). We should also consider how to coordinate the rollback and recovery actions to avoid the domino effect [Ran75] for systems where multiple processes are involved in error detection and recovery. This cooperative recovery is inevitable in many applications.

The coordination among processes can be done either at design time (i.e., static coordination) or at runtime (i.e., dynamic coordination). A hybrid form of coordination is also possible. The conversation scheme [Ran75] is an example of static coordination, whereas the PTC scheme [Kim88b] and some check pointing schemes [Bha88b, Ram88] are based on dynamic coordination.

The Conversation Scheme

The conversation scheme is based on approval and static cooperation among interacting processes. The conversation is a two-dimensional enclosure of recoverable activities for multiple interacting processes, in short, a recoverable interacting session [Kim82,Ran75]. It creates a "boundary" which process interactions may not cross. The boundary of a conversation consists of a recovery line, a test line and the walls defining membership as shown in Figure 7. Each participant process contains one or more Try-blocks designed to produce the same or similar computational results as well as an acceptance test (conversation acceptance test or CAT). The Try-block is a logical expression representing the criterion for determining the acceptability of the execution results for the Try-blocks. A recovery line is a coordinated set of recovery points for interacting processes that are established (possibly at different times) before the interactions begin. A test line is a correlated set of acceptance tests for the interacting processes.

A conversation is successful only if all the interacting processes pass their acceptance tests which form the test line. The participants are allowed to leave the conversation after all the participants have passed. If any of the acceptance tests fail, all of the processes must roll back to the recovery line and retry with their alternate Try-blocks. Thus, the alternate Try-blocks collectively define an alternate interacting session (AIS). And, the primary Try-blocks which are executed first inside the conversation, define the primary interacting session (PIS). Note however, that parallel execution of the AIS and PIS are also possible as discussed in Section 4. A process that has executed its try block, and passed its acceptance test is said to have finished its conversation task.

The conversation scheme has been adopted as the primary fault tolerance mechanism to be used for FT-SUVS. This choice was made because: (1) the scheme provides recovery actions for interacting processes, (2) software redundancy is furnished, (3) real-time recovery is achievable, and (4) the scheme is relatively easy to implement.

3.3 FT-SUVS

The FT-SUVS has been implemented on a sixteen-nodes HHRH (i.e., HHRH [16,2,4,2]) as shown in Figure 3. These sixteen nodes run multiple copies and/or versions of four analyzer tasks simultaneously.

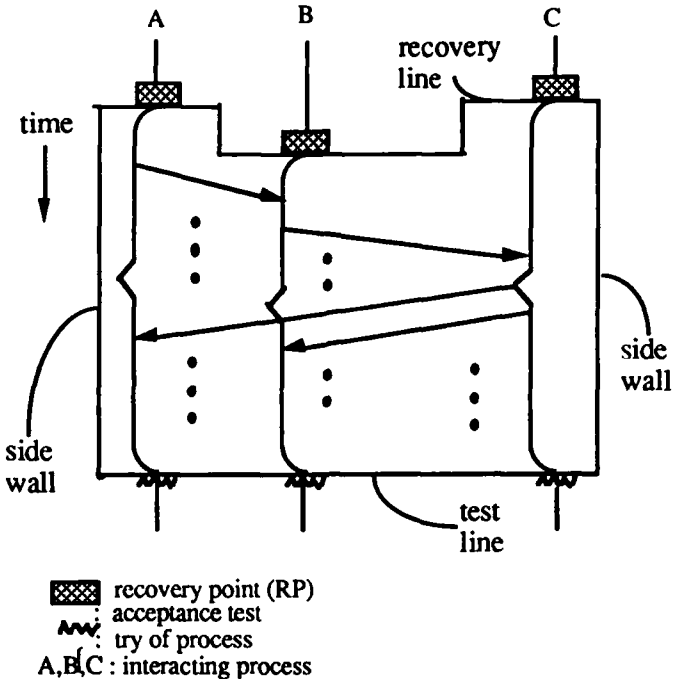


Figure 7. Conversation.

The following conversation structure is proposed for the FT-SUVS.

(1) Four analyzer tasks, i.e., speed, direction, vision, and surface analyzers, cooperate (i.e., exchange information and preliminary decisions) to make decisions on any action.

(2) Output (to actuators) is made only when all of the analyzer tasks agree.

(3) Upon disagreement, the alternate algorithms (or alternate interacting session) are executed.

(4) If they don't reach any final agreement within a specified time or they have failed in all alternate algorithms, the system goes into emergency mode and tries to stop the vehicle in the safest and fastest manner.

Although the conversation scheme is the primary fault tolerance scheme in FT-SUVS, incorporation of other schemes is also possible (e.g., a simple voting and a comparison scheme within the system at various stages). Among others, two configurations, one with a simple voting scheme and one with the conversation scheme, have been implemented and tested.

4. IMPLEMENTAION OF FT-SUVS

This section discusses the implementation details of FT-SUVS. The sixteen node HHRH architecture is simulated using four SUN Workstations. FT-SUVS is implemented with two different fault tolerant schemes: one with the simple voting scheme and one with the conversation scheme.

4.1 HHRH Architecture Simulator

As described in Section 3 a sixteen node HHRH (HHRH[16,2,4,2]) is used for this experiment. The architecture, consists of four clusters with each cluster consisting of four nodes. This architecture is simulated on four SUN Workstations (i.e, each workstation simulates a cluster of four nodes as shown in Figure 8). These sixteen nodes run multiple copies of the four SUVS analyzer tasks. The copies are either all identical, all different, or an even mixture of both.

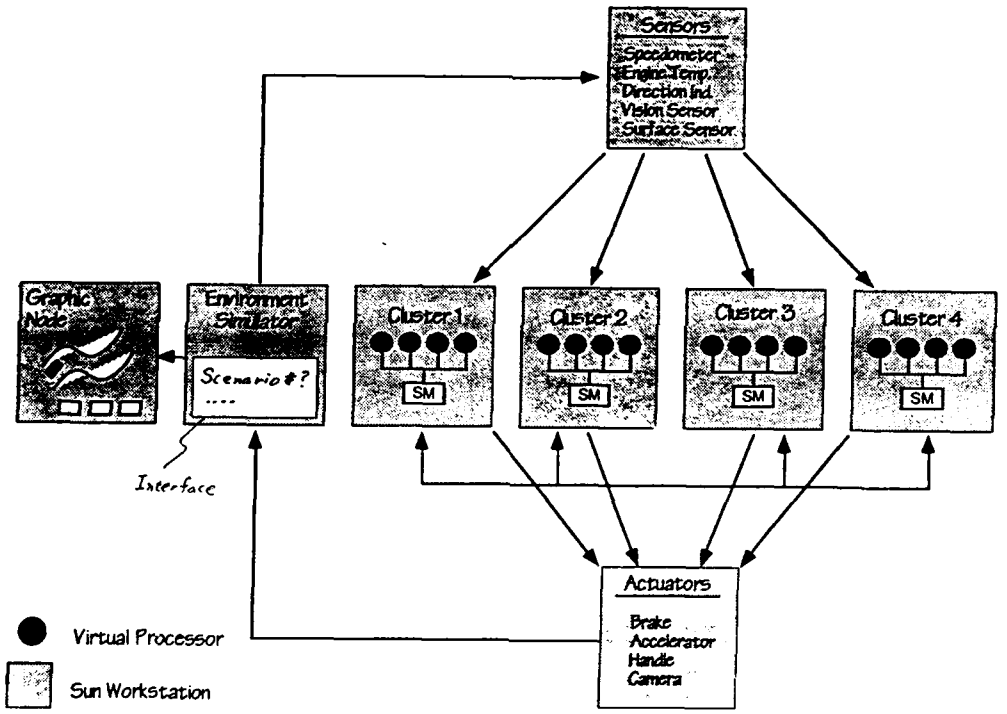
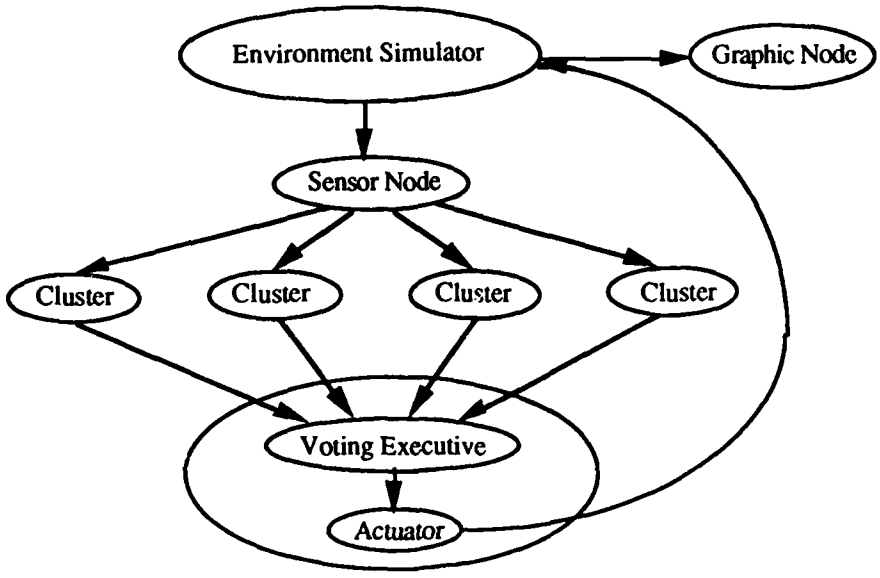


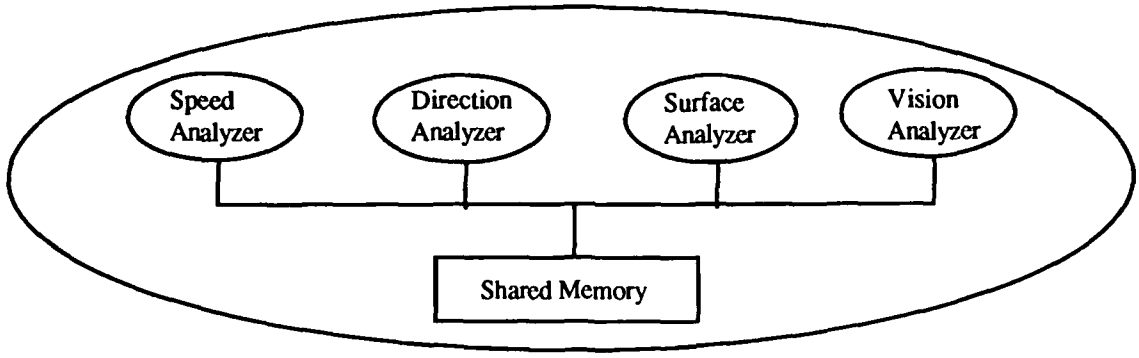
Figure 8. SUVS Testbed with Software Redundancy in HHRH Architecture (FT-SUVS).

4.2 FT_SUVS(I)

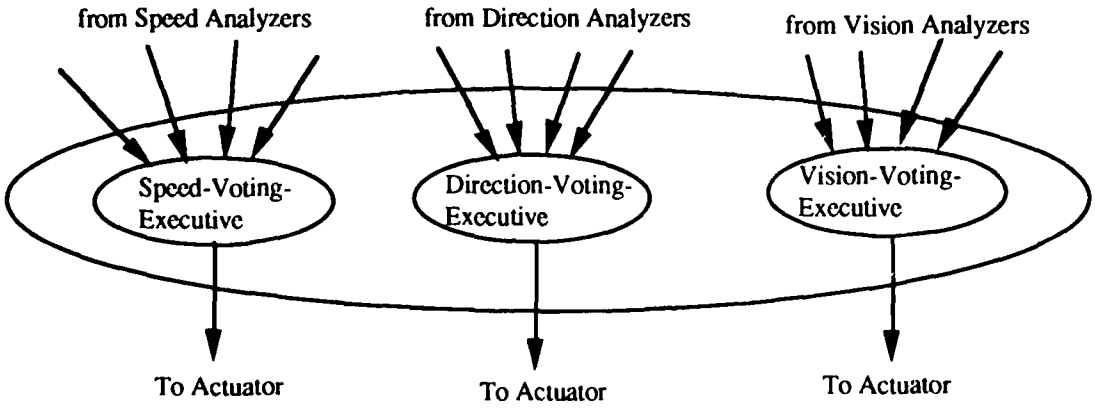
The voting scheme is implemented in FT-SUVS(I). As shown in Figure 9, each cluster runs a set of analyzer tasks which provides quadruple redundancy. All sixteen tasks run simultaneously for every input from the sensor tasks and forward their decision to one of the three



a. A network configuration with four sets of analyzer tasks.



b. A cluster



c. Voting_Executives

Figure 9. Configuration of FT_SUVS(I).

Voting_Executives at the actuator. Speed_Voting_Executive receives decisions on speed, either acceleration, deceleration or no change, from different speed analyzer tasks, each residing on a different cluster. Similarly, Direction_Voting_Executive and Vision_Voting_Executive receive decisions from multiple copies of direction analyzers and vision analyzers, respectively. The Speed_Voting_Executive process looks as follows:

```
Speed_Voting_Executive()
{ for (;;)
        { start_timer();
          for (i=1;i<=no_of_clusters; i++)
                { recv_speed_analyzer (speed_in);
                  speed_array[i] = speed_in
                }
          speed_voting_routine (speed_array, i, &speed_out)
          send_speed_to_actuator (&speed_out)
        }
}
```

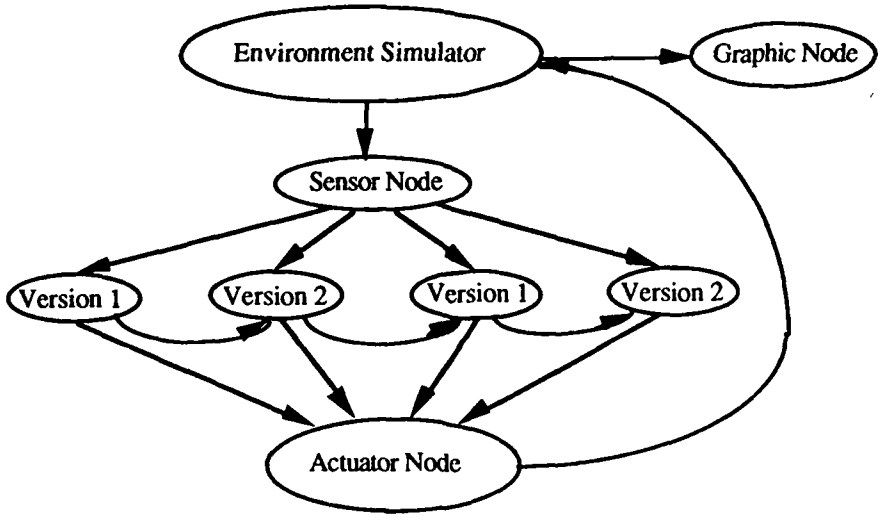
As shown above the timer is set for every voting. If a timeout occurs the control jumps out of the loop and voting is done based on the decisions received by that time. The voting policy we have chosen is a conservative approach. If the voting is unanimous or 3:1, the decisions are forwarded to the actuator. Otherwise, (i.e., the voting is split with 2:2, 2:1:1, or 1:1:1:1) the decision is reserved. A successful vote (i.e., unanimous or 3:1 voting) on a consecutive subsequent decision from the analyzers is forwarded to the actuator and, in this case, the previous outputs are ignored. If the voting is not successful on consecutive attempts the emergency routine is invoked and the vehicle is stopped in the fastest and safest fashion.

We believe that the conservative approach is suitable to this kind of application because "fail stop" is allowed. That is, it is better to stop the vehicle if we are not confident about the decision. The conservative approach may generate more "false alarms", but we can avoid catastrophic situations. However, there are some applications such as Ballistic Missile Defense, where "fail stop" is not allowed. In such environments some form of false alarm filtering must be employed.

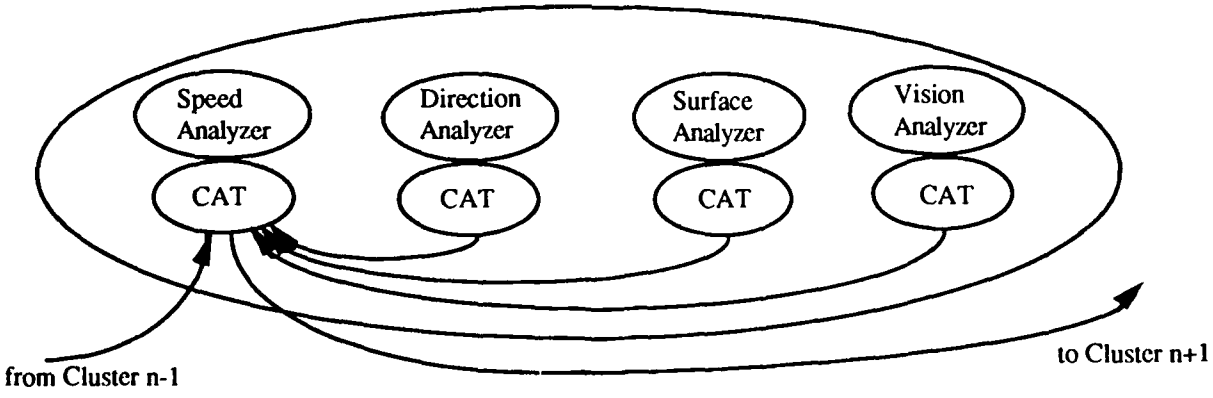
4.3 FT_SUVS (II)

The conversation scheme is implemented in FT-SUVS (II). Each cluster runs a set of analyzer tasks which participate in the conversation of every execution cycle. As shown in Figure 10, Clusters 1 and 3 run the original version of SUVS, whereas Clusters 2 and 4 run the second version of SUVS. The analyzer tasks in Cluster 2 forward their decision to the

actuators only when the analyzer tasks in Cluster 1 has failed either in logic or in time as determined by the CAT. Similarly, the tasks in Cluster 3 (or Cluster 4) forward decisions to the actuators only when clusters 1 and 2 (or Clusters 1,2, and 3) have failed the CAT.



a. A network configuration with two versions of analyzer tasks



b. CAT execution in cluster n.

Figure 10. Configuration of FT-SUVS(II).

Implementation of the Second Version of SUVS

In principle, the second version of the analyzer tasks should be designed such that it may produce acceptable results for the cases where the original version of the analyzer tasks

fails to do so. Although designing efficient multiple versions is somewhat application-dependent we believe a systematic approach is possible for this type of application.

The major difference between the two versions in SUVS is in the way the decisions are made. Decisions are made based on several factors. For example, the speed analyzer makes a decision based on current speed, RPM (revolutions per minute) of the engine, road condition (e.g., wet or dry), curve of the road, the existence of obstructive objects in front and if they exist, characteristics(e.g., moving speeds) of the objects. Therefore, the second version is designed by applying the decision factors in a different sequence. Such produced versions are still considerably diverse in their logics used and also have substantially different chances of encountering same software failure.

Conversation Acceptance Test (CAT)

There are three different approaches to execution of the CAT: Centralized, Decentralized, and Semi-centralized [Yan89]. In the centralized CAT approach only one designated participant, named "head" participant, contains the complete CAT routine. Therefore, the head participant executes the CAT when all the participants have finished their execution of Try-blocks and then broadcasts the CAT result to other participants. In the decentralized CAT approach, each participant performs its own acceptance test, and the participants exchange their results with each other. Therefore, every process receives the results of the other participants and figures out by itself the result of the "non-local" acceptance test.

The semi-centralized approach, which is adopted for the FT-SUVS(II) implementation, compromises the above two approaches in such a way that the "local" acceptance test is done by each participant and the "non-local" CAT result is determined by the head participant. Thus, each participant performs its own acceptance test and sends the result to the head participant. The head participant judges the success or the failure of the CAT depending upon whether all the reports received are success reports or not, and then broadcasts the CAT result. We have chosen this approach mainly because of the low communication overhead and the possibility of fast recovery as pointed out in [Yan89].

The CAT checks whether (1) the decisions are made based on correct information, (2) the decision made locally is reasonable with respect to both the recently observed condition of the vehicle and the laws of physics, and (3) the decisions do not conflict with each other. The first part of the test (which is trivial in nature by comparison to the other two parts) can be facilitated by sending the input data (which is used to make decisions) along with any preliminary decisions. For example, the input data from the speedometer is initially received by the speed analyzer for every cycle of the execution. This data is checked and then broadcast to other analyzers. The data may then be used by other analyzers in reaching certain preliminary

decisions. Therefore, in the first part of the CAT, the speed analyzer checks whether the speed data received from other analyzers together with their preliminary decisions, are the same as the original data that it received from the speedometer and has since kept. By doing so we can detect possible faults due to communication and/or memory failures. The second and more important part of the CAT is largely to check if the preliminary local decision falls within a reasonable range. For example, if the preliminary decision on a change in the acceleration is beyond the capacity of the vehicle, then clearly a computation error can be suspected. Actually, this reasonableness test of the local preliminary decision can be performed even before it is sent to other analyzers. The third part (i.e., checking the possibility of conflict) can be viewed as a "non-local" logic test. That is, the decisions made by the analyzers are examined to see if there is any conflict among them.

Handling of Redundant Messages

In some cases the actuators receive decisions from more than one cluster. This may happen if Cluster 2 (or Cluster 3 or 4) forwards the decisions to the actuator due to timeout before it receives the completion signal from Cluster 1 (or Cluster 2 or 3). In order to handle these duplicated decisions a sequence number is attached to every set of decisions. That is, the sequence number is increased for every cycle of execution. The actuators ignore the decisions whose sequence numbers are the same as or smaller than the one previously received.

5. EXPERIMENTAL RESULTS AND FUTURE WORKS

The preliminary experimental result and failure modes are summarized here. The plan for future SUVS testbed experimentation is also discussed.

5.1 Fault Mode and Fault Injection

We considered both hardware and software faults in this experiment . Hardware faults include node crash, link failure, and cluster memory failure. Software failures include design/implementation errors and calculation overflow/underflow. Software failures cannot be handled by the simple voting scheme. Table 1 summarizes how these faults are simulated and injected into the system. The table also shows that how these faults are detected and recovered in FT-SUVS(I) and FT-SUVS(II).

5.2 Time Measurement (Preliminary Results)

In this experiment we measured the response time of each cycle: Sensors get data from

the environment simulator every 100 msec; These input data are processed by the analyzers that make decisions and forward the decisions to the actuators; The actuators finally react to the system by sending output to the environment simulator. The measurement is done at two points in the environment simulator, (1) the time that the new input set is forwarded to the sensors and, (2) the time that the outputs from the actuators arrive. The response time deadline is 200 msec. (We plan to scale down the execution cycle and response time deadline in future experiments).

Fault Mode	How to Inject	How to Detect/Recover
FT-SUVS(I)		
Node crash	Using exit(1) system call at a given execution cycle	Voting with timeout
Link failure	Sensors stop sending data at a given execution cycle	Voting with timeout
FT-SUVS(II)		
Node Crash	Using exit(1) system call at a given execution cycle	Timeout
Link failure	Infinite loop at a given execution cycle	Timeout
Software failure	Sending corrupted data to the analysis at a given execution cycle	CAT

Table 1. Fault injection and detection in FT-SUVS's.

Figure 11 shows the response time of the first two hundred execution cycles for SUVS and FT-SUVS(I). The dotted line is the response time when only one cluster (i.e., one set of analyzer tasks) is running and no voting scheme is incorporated into the actuator node. The response time varies between 70 msec and 80 msec. The solid line shows the response time of FT-SUVS(I). Faults (node crash and link failure) are injected to Clusters 1, 2, and 3 at execution cycle numbers 50, 100, and 150, respectively. The timeout is set to 50 msec at each voting executive. As shown in the figure the response time varies between 120 msec and 140 msec before the fault is injected. This 60 msec overhead is mainly due to the communication cost between the sensor node and the analyzer clusters and between the analyzer clusters and the actuator node. Consequently, response time does not vary much even after the fault is injected.

In Figure 12, a fault (node crash in Cluster 1) is injected at cycle number 52 in FT-SUVS(II). The second fault (link failure) is injected to one node in Cluster 2 at cycle number

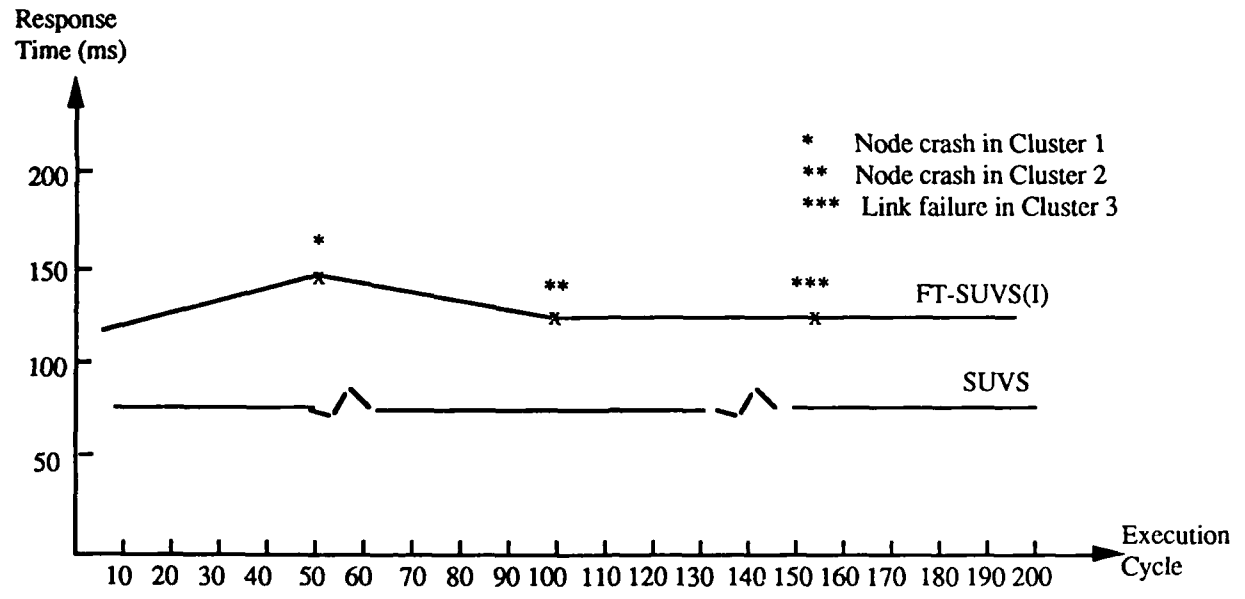


Figure 11. Response time of SUVS AND FT_SUVS(I).

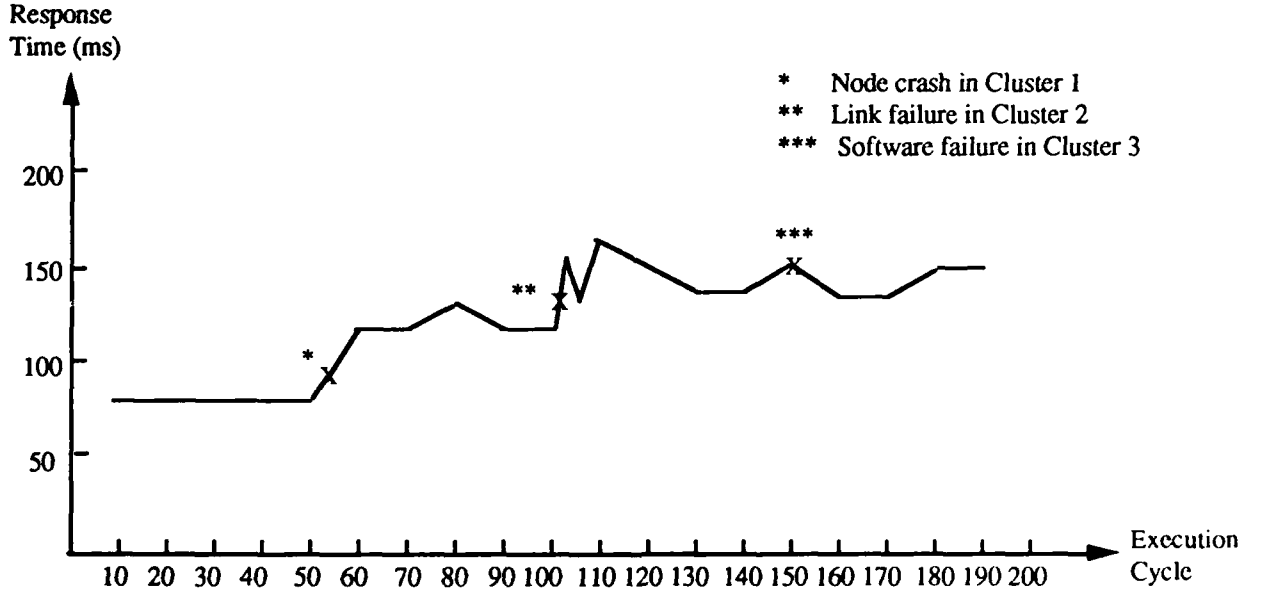


Figure 12. Response time of FT_SUVS(II).

101. Finally, at cycle number 150 the third fault (software error) is injected to one node in Cluster 3. As shown in the figure, the response time when no fault is injected is about 85 msec. Therefore, if there is no fault the overhead of FT-SUVS(II) is much smaller than that of FT-SUVS(I) because of less communication overhead. However, the response time increases

after the fault is injected to a node in Cluster 1. This is due to the waiting time at Cluster 2. Currently, the timeout is set to 20 msec in Cluster 2. That is, Cluster 2 waits for the completion signal of each execution cycle from Cluster 1. If the signal does not come within 20 msec (after the completion of computation) Cluster 2 forwards its output to the actuator. Waiting times in Cluster 3 and 4 are 30 msec and 40 msec, respectively. After all three faults are injected the response time stays about 140 msec.

5.3 Future Experimentation

We have established SUVS testbed and experimented with two fault-tolerant configurations: one with the voting scheme and one with the conversation scheme. We plan to use the testbed for clinical study including evaluation of techniques and methods for design of reliable distributed real-time systems as follows.

(1) Further experimental study on fault tolerance schemes such as voting, comparison and DRB (Distributed Recovery Block) will be conducted. For ultra reliable computing we will also study how the different schemes can be incorporated effectively into the systems at various stage of computing For example, Figure 13 shows one possible process configuration using three error detection and recovery schemes. In the upper half of the figure there are four identical analyzer processes. These processes vote on a message before it is transmitted to other analyzer process(es). At the end of every conversation the CAT is performed on outputs that go to the actuators. In the bottom half of the figure, there is another set of processes with the same configuration but run different versions of processes. The results from both sets (i.e., primary and alternate), if both have passed their CAT's, are compared before the results are finally sent to the actuators. By comparing these results we may detect errors that have not been detected by the CAT. If one of them fails the CAT or does not produce the results within a specified time (i.e., missed deadline or timeout) then the results produced by the other set are immediately used.

(2) In [Yan89] various implementation approaches of the conversation scheme have been proposed. Those approaches, e.g., synchronous and asynchronous conversations as well as centralized, decentralized, and semi-centralized acceptance tests, will be validated by experimental study.

(3) One of the major factors to be considered for incorporating redundant software is the communication and synchronization overhead. Since the HHRH architecture provides two different communication paths (i.e., shared memory and point-to-point connection), the communication cost are somewhat alleviated. Another design consideration is the possible incorporation of the timeout mechanism. The timeout mechanism is required for every

synchronization stage to avoid lockup of other processes. The proper timeout period should be determined based on an analysis of the system behavior. It is also important to consider the cases where no agreement is reached among processes (in the case of voting or comparison) or when no acceptable result is produced (in the case of CAT). Detailed analysis of different configurations and implementation strategies (i.e., design and implementation cost, runtime cost, timing aspects, and reliability trade-offs) will be discussed in later papers.

(4) Reliability modeling and estimation of FT-SUVS is another major area for the future research. The model should be able to reflect both hardware and sofware components.

The testbed will also be used for clinical study of the specification, design, and implementation methods studied at UTA [Gra89].

6. SUMMARY

This chapter reports implementation details and the experimental result of a distributed real-time system testbed developed at UTA. This is part of our continuous effort to investigate development methods and techniques for ultra reliable real-time computing. This particular work focuses on experimental validation of fault tolerance techniques. The testbed is named SUVS and is implemented on a network of eight SUN Workstations.

We have implemented two fault-tolerant SUVS configurations. The first is FT-SUVS(I) which incorporates both multiple identical copies of software and a voting scheme. The second is FT-SUVS(II) which uses two versions of the software (and two copies of each version) and a set of acceptance test routines. Both configurations are implemented on a sixteen-node HHRH architecture. The response time is measured for both cases by injecting various faults. The preliminary measurement exposes the temporal behavior of the two fault-tolerant configurations of SUVS.

We believe that the major obstacle to realizing fault-tolerant computing concepts within practical technologies has been the cost of implementing multiple versions of software. In addition, the acceptability check routines for "real" applications has been another obstacle in the application of the fault tolerance schemes which are approval based. In this work we have demonstrated that multiple versions can be generated in a systematic manner that is both cost effective and conceptually simple when used within this application domain. The acceptability check routine is incorporated into FT-SUVS(II) as a form of conversation acceptance test (CAT). Although there may be no general way of designing acceptability check routines, we believe the approach used here in designing the CAT for FT-SUVS(II) can be applicable to many applications.

Software Configuration with Three Error Detection Schemes

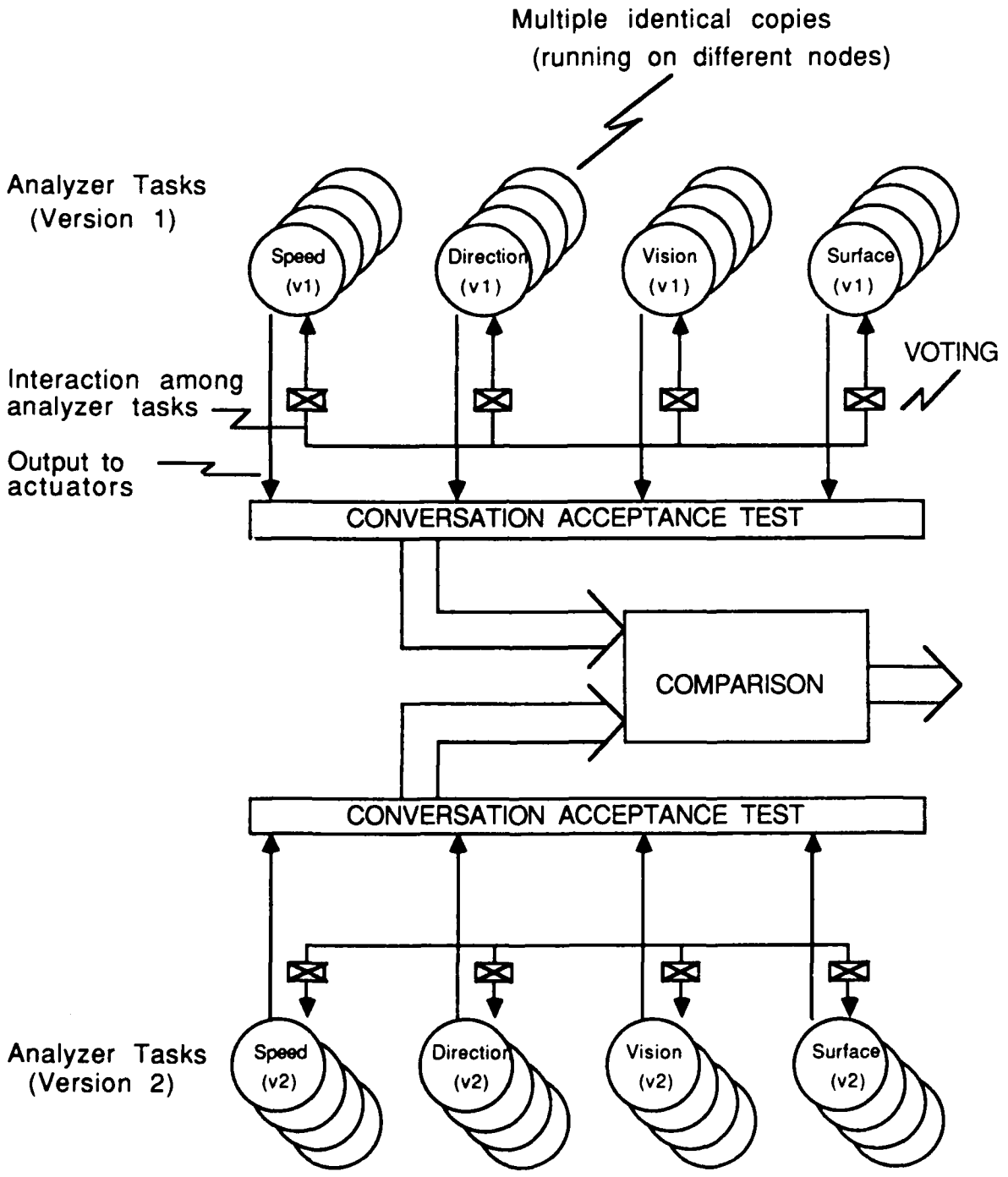


Figure 13. Three levels of error detection in FT-SUVS.

We plan to refine the timing measurement capabilities of our testbed. So far, we have only measured turn-around times. We plan to measure the exact overhead at various stages of each cycle. We also plan to implement and experiment with various fault-tolerant configurations (e.g. see Figure 13). This experimental work along with the theoretical work on timing and reliability analysis will provide a valuable contribution towards developing ultra reliable real-time systems in a cost-effective manner.

7. REFERENCES

[Aga86] Agarwal, D.P., Janakiram, V.K., and Pathak, G.C., "Evaluating the Performance of Multicomputer Configuration", *IEEE Computer*, May 1986, pp. 23-37.

[Avi85] Aviziennis, A., et al., "The UCLA DEDIX System: A Distributed Testbed for Multiple-Version Software", *Proc. FTCS-15*, June 1985, pp. 126-134.

[Bha88] Bhargava, *'Implementation and Analysis of a Hybrid Higher Radix Hypercube Architecture'*, MS Thesis, The University of Texas at Arlington, April 1988.

[Bha89a] Bhargava, B. and Reidl, J., "The Raid Distributed Database System", *IEEE Trans. Software Eng.*, Vol. 15, No. 6, June 1989, pp. 726-736.

[Bha89b] Bhargava, B., and Lian, S.R., "Independent Checkpointing and Concurrent Rollback for Recovery in Distributed Systems - an Optimistic Appreoach", *IEEE Proc. of 7th Symposium on Reliable Distributed Systems*, Oct. 1988, pp. 3-12.

[Bhu84] Bhuyan, L. and Agrawal, D.P., "Generalized Hypercube and Hyperbus Structures for a Computer Network", *IEEE Trans. on Computers*, Vol. 34, April 1984, pp. 323-333.

[Chu87] Chu, W., Kim, K.H., and McDonald, W.C., "Testbed-based Validation of Design Techniques for Reliable Distributed Real-Time Systems", *Proc. IEEE (Special Issue on Distributed Databases)*, Vol. 75, No. 5, May 1987, pp. 649-667.

[DCS89] "Practicality of Distributed System Design Tools", *Panel Discussion of the 9th Int'l Conf. on Distributed Computing Systems*, June 1989.

[Gra89] Grabow, P.C. and Yang, S.M., *A Development Mothodology for Distributed Real-Time Software Systems*, Proposal to the Texas Advanced Research Program, July 1989.

[Hoa85] Hoare, C.A.R., *Communicating Sequential Processes*, Prentice-Hall International, Englewood Cliffs, New Jersey, 1985.

[IEE86] IEEE Computer Soc., *The Workshop on Design Principles for Experimental Distributed Systems*, Purdue Univ., Oct. 1986.

[Jah86] Hahanian, F., and Mok, A.K.L., "Safety Analysis of Timing Properties in Real-time Systems", *IEEE Trans. on Software Engineering*, Vol. 12, No. 9, September 1986, pp. 890-

904.

[Kim82] Kim, K.H., "Approaches to Mechanization of the Conversation Scheme Based on Monitor," *IEEE Trans. on Software Eng.*, May 1982, pp. 189-197.

[Kim85] Kim, K.H. and Yang, S.M., "Fault Tolerance Mechanisms in Real-Time Distributed Operating Systems: An Overview," *Proc. PCCS 85*, Oct. 1985, pp. 220-229.

[Kim88a] Kim, K. H., "An Approach to Experimental Evaluation of Real-Time Fault-Tolerant Distributed Computing Schemes", *IEEE Trans. on Software Eng.*, June 1989, pp. 715-725.

[Kim88b] Kim, K.H., "Programmer Transparent Coordination of Recovering Concurrent Processes: Philosophy and Rules for Efficient Implementation", *IEEE Trans. on Software Engineering*, Vol. 14, No. 6, June 1988, pp. 810-821.

[Koh86] Kohler, W. and Jeng, B.P., "Performance Evaluation of Integrated Concur- rency Control and Recovery Algorithms using a Distributed Transaction Processing Testbed", in *Proc. 6th Int. Conf. Distributed Computing Systems*, May 1986, pp. 130-139.

[Kop89] Kopetz, H., et al., "Distributed Fault-Tolerant Real-Time Systems: The Mars Approach", *IEEE MICRO*, Feb. 1989, pp. 25-40.

[McD82] McDonald, W.C. and Smith, R.W., "A Flexible Distributed Testbed for Real-Time Applications", *IEEE Computer*, Vol. 15, No. 10, Oct. 1982, pp. 25-39.

[Mil80] Milner, R., *A Calculus of Communicating Systems*, Spring-Verlag, Lecture Notes in Computer Science, No. 92, 1980.

[Mus90] Musa, J.D., and Everett, W.W., "Software-Reliability Engineering: Technology for the 1990s", *IEEE Software*, Nov. 1990, pp. 36-43.

[Ram88] Ramanathan, P., and Shin, K.G., "Checkpointing and Rollback recovery in a Distributed System Using Common Time Base", *IEEE Proc. of 7th SRDS*, Oct. 1988, pp. 13-21.

[Ram89] Ramsey, N., "Developing Formally Verified Ada Program", *Proc. 5th Int'l Workshop on Software Specification and Design*, May 1989, pp. 257-272.

[Ran75] Randell, B., "System Structure for Software Fault Tolerance," *IEEE, Trans. on Software Eng.*, June 1975, pp. 220-232.

[Yan89] Yang, S.M. and Kim, K.H., "Implementation of the Conversation Scheme in Loosely Coupled Distributed Computer Systems", *Proc. Int'l Conf on DCSs*, June 1989, pp. 570-578.

CHAPTER VIII

SUMMARY AND A NEW DEVELOPMENT MODEL FOR FUTURE REAL-TIME APPLICATIONS

1. Summary

In this report we have discussed several aspects of designing fault-tolerant parallel/distributed computer systems for real-time applications, which include parallel architecture, software fault tolerance schemes, performance modeling, real-time task scheduling, real-time communication protocols, and a testbed for experimental validation. These works must be integrated into a stream of development process.

The traditional waterfall model seems inadequate for developing fault-tolerant real-time systems in distributed/parallel environments, which require rigorous design and verification of timeliness and reliability; this is because (1) design time verification has limitation, especially in distributed/parallel computing environments and (2) due to the high cost and long development times it is often too late when problems in software are discovered at system integration time.

We now propose a new model for the development of large, complex and distributed real-time embedded systems. It is our premise that since software costs are high compared to those of hardware (which include development costs) and in many cases the complexity of software is much higher than hardware, that a better approach is to develop the system's software functionality first. This premise is somewhat contrary to the typical approach of finding the best hardware solution and then meeting the remaining system requirements with software. Our model defines five phases of development.

2. Five Phases of a New Development Model

Phase I: Problem Conception. Target system is characterized. A domain specific framework is necessary to support better communication between the customer and the developer such that a clear and precise specification can be written. Time and reliability requirements must be specified along with logical requirements. The formal approach is preferable [Ost92], yet some form of compromise seems inevitable.

Phase II: Decomposition. Software driven decomposition of system's functionality is done based on an object-oriented design (OOD) approach [Boo91]. The OOD approach seems natural for large complex real-time systems: sensor and actuator objects and in between these objects are control objects. Moreover, reusability and maintainability among others are the positive attributes of the approach. Phase II focuses on logic correctness only, and excludes consideration of timing or architectural elements. However, timing criteria, inherited from the requirements specification are carried forward into the decomposition framework.

The output from phase II consists of a set of communication objects, each of which is the smallest reusable and schedulable unit. It is often the case that the system results in many objects (orders of hundreds to thousand). These objects must be clustered to minimize the overheads due to communications, duplicated functionality, coupling and locality of resources etc.

Phase III: Software Instantiation. This phase begins by clustering of objects. Clustering is combining the objects based on their connectivity (i.e., coupling or data sharing) and functionality. This reduces the complexity and overhead due to too many objects. The runtime kernel is another object that must be instantiated in this phase. We propose a generic, framed kernel designed for such applications. The frame must be optimized and/or adjusted for a particular application (i.e., specific functions and services including communications) and then instantiated. Therefore, all of the *system* functionality (including hardware) is instantiated as software which constitutes the output of this phase.

In many cases an integrated software system which is constructed from reusable objects may lead to inefficient implementation. For this reason, we believe that the system should go through a "Post Integration Optimization." In this phase the system will be converted into conventional process (task) graphs. These task graphs will be partitioned for implementation on a generic multiprocessor system. The generic processors can be based either on a shared memory or distributed memory model. Optimization decisions such as scheduling, load balancing, placement of data, context switching overhead, and interprocess communication are performed in the next phase.

Phase IV: Design Evaluation. The task of this phase is to determine the "optimal" candidate target architecture which satisfies the system's performance characteristics, i.e., timeliness and reliability. First step is to map one task to one processor (1:1). Then combine (i.e., further mapping) more than one task into one processor. The performance is affected by three major factors which are (1) architecture, (2) mapping of tasks to processors, and (3) scheduling of tasks and messages. Therefore, these factors should all be considered simultaneously (i.e. many different combinations must be evaluated). This must be done by three analysis approaches, namely, mathematical (or theoretical) analysis, simulation-based analysis [Fuj90], and testbed-based analysis [Yan92]. These analyses are not exclusive, and often supplementary. Although we do not mention other limitations such as hardware cost, maximum number of processors, and various topologies still, in some cases these factors should also be considered.

(1) Mathmatical Analysis: Mathemetical proof, albeit restricted is necessary for reliability and timeliness guarantee. For example, we should guarantee the timeliness of very critical tasks, to meet the execution deadlines. That is, schedulability of some tasks must be guaranteed before runtime. Real-Time Logic and/or deadline scheduling algorithms can be used. Reliability modeling with the consideration of both hardware and software is yet another area to be studied.

(2) Simulation-based Analysis: The simulation-based technique can be used to analyze the performance whenever the mathematical approach is impossible. (Many mathemetical models are not applicable for analyzing complex real-time systems due to the approximation or simplification of many parameters.) The simulation model should realistically reflect the actual physical environment or system, i.e., inputs to the systems and feedback from the outputs, as well as system behavior. One challenging issue is parallel real-time simulation.

(3) Testbed-based Analysis: The testbed is also utilized to evaluate the candidate configuration. The major focus here is to confirm the simulation result. The software including the kernel are executed on a hardware emulator. Various timings are actually measured. Real faults are injected into the system and the recoverability is also evaluated. Another major task is software refinement including debugging. Software features which heavily depend on architecture and networking are refined. For example, synchronization and communication mechanisms must be refined depending on whether the shared memory or message passing mechanism is used. The development of a flexible and truthful hardware emulator is another challenge.

Phase V: System Integration. The target architecture is developed while the software can continue to be refined using the testbed. The integration test is performed after the hardware part is complete.

Figure 1 shows the five phases of our development model. The major difference between our model and the traditional waterfall model is that software decomposition and instantiation are done before the target architecture is considered. This provides several advantages: (1) software reuse is easier, (2) performance analysis is more accurate, and (3) software cost is reduced.

3. Current Work

The current research at the University of Texas at Arlington (UTA) is as follows:

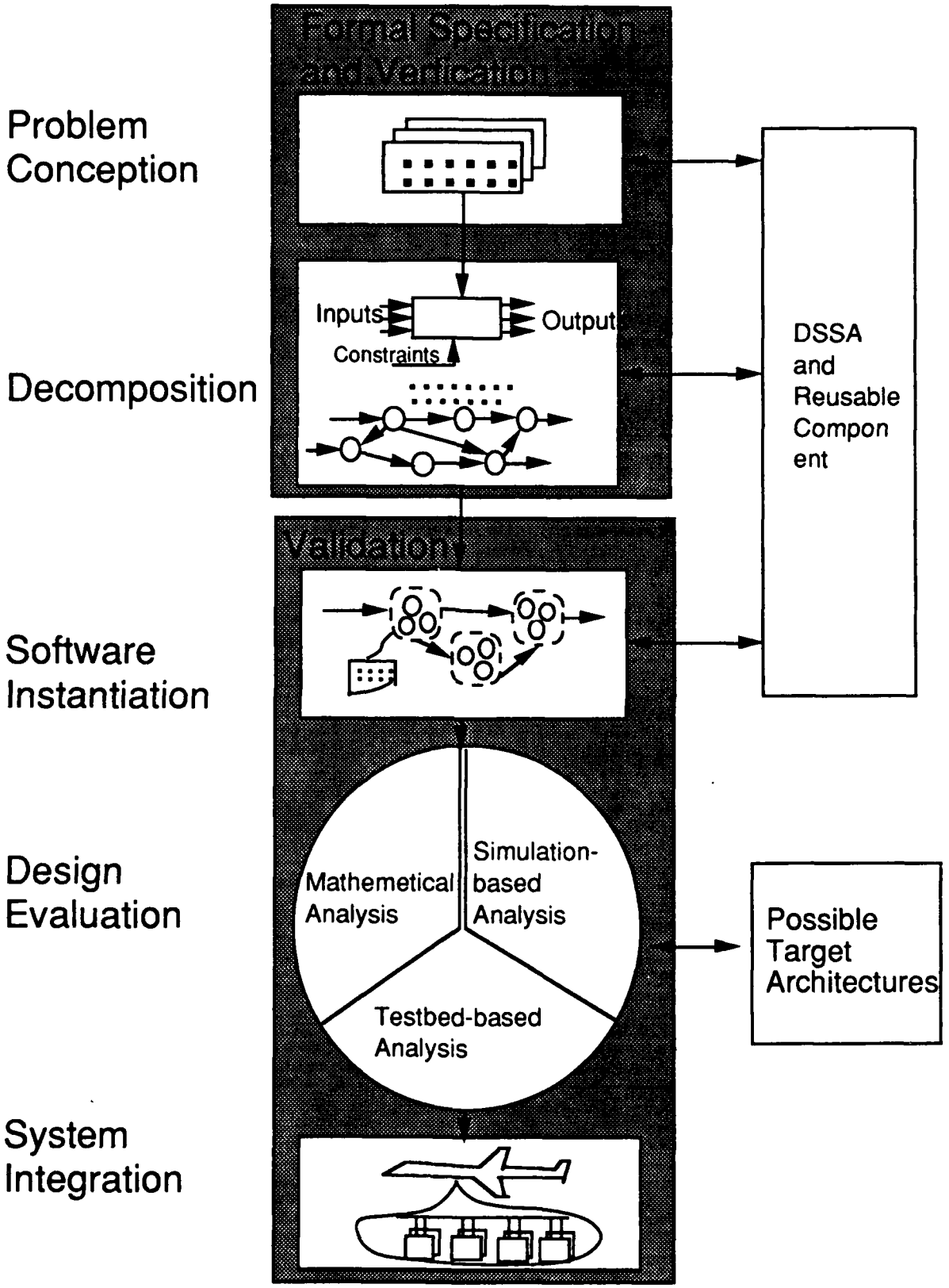


Figure 1. A Development Model

(1) HHRH Architecture: More theoretical and practical study on HHRH (Hybrid Higher Radix Hypercube) architecture is carried on. Especially, fault-tolerant routing algorithms and various fault-tolerant configuration for ultra reliable computing are under study.

(2) Real-time task and message scheduling: Time-value-function based scheduling as well as the deadline-driven scheduling algorithms are under study. We have proposed a dynamic execution time estimation technique for effective scheduling. Performance of various real-time communication protocols is under study.

(3) Fault-tolerant model: Software fault-tolerance model for interacting tasks is under investigation. The model will be extended with the consideration of hardware configuration.

(4) A parallel simulation technique: We will focus on stochastic discrete event simulations for performance evaluation and study of meaningful ways to measure the validity of simulation models.

(5) Testbed and real-time kernel: SUVS (Simplified Unmanned Vehicle System) testbed is being used to conduct experimental evaluation of techniques and methods for design of reliable distributed real-time systems. We will extend the testbed with a generic, framed real-time kernel for large real-time systems. The major parts of the kernel are real-time scheduling, intertask and internode communication, and fault-tolerant support.

(6) Integrated environment: An integrated environment and tools will be developed for the three different analysis components, i.e., mathematical, simulation-based and testbed-based analysis. This will provide an integrated (i.e., mutually supplemented) development environment.

4. REFERENCES

[Boo91] G. Booch, 'Object Oriented Design', Benjamin/Cummings, 1991.

[Das89] A.K. Deshpande and K.M. Kavi, " A Review of Specification and Verification Methods for Parallel Programs, Including the Dataflow Approach", IEEE proceeding, Vol. 77, No. 12, Dec. 1989, pp. 1816-1828.

[Fuj90] R.M. Fujimoto, "Parallel Discrete Event Simulation", Communications of The ACM, Vol. 33, No. 10, Oct. 1990, pp.31-53.

[Ost92] J.S. Ostroff, "Survey of Formal Methods for the Specification and Design of Real-Time Systems", to appear in IEEE Tutorial on Specification of Time.

[Yan92] S.M. Yang, et. al., "SUVS: A Distributed Real-Time System Testbed for Fault-Tolerant Computing", Proceedings of the 1992 Symposium on Applied Computing.