THE DESIGN OF FAULT TOLERANT SOFTWARE FOR LOOSELY COUPLED DISTRIBUTED SYSTEMS.

Andrew Martin Tyrrell

Submitted for the degree of Doctor of Philosophy.

The University of Aston in Birmingham

April 1987.

THE UNIVERSITY OF ASTON IN BIRMINGHAM

The Design of Fault Tolerant Software for Loosely Coupled Distributed Systems.

Andrew Martin Tyrrell

Submitted for the degree of Doctor of Philosophy 1987.

Summary.

Requirements for systems to continue to operate satisfactorily in the presence of faults has led to the development of techniques for the construction of fault tolerant software. This thesis addresses the problem of error detection and recovery in distributed systems which consist of a set of communicating sequential processes.

A method is presented for the 'a priori' design of conversations for this class of distributed system. Petri nets are used to represent the state and to solve state reachability problems for concurrent systems. The dynamic behaviour of the system can be characterised by a state-change table derived from the state reachability tree.

Systematic conversation generation is possible by defining a closed boundary on any branch of the state-change table. By relating the state-change table to process attributes it ensures all necessary processes are included in the conversation. The method also ensures properly nested conversations.

An implementation of the conversation scheme using the concurrent language occam is proposed. The structure of the conversation is defined using the special features of occam. The proposed implementation gives a structure which is independent of the application and is independent of the number of processes involved.

Finally, the integrity of inter-process communications is investigated. The basic communication primitives used in message passing systems are seen to have deficiencies when applied to systems with safety implications. Using a Petri net model a boundary for a time-out mechanism is proposed which will increase the integrity of a system which involves inter-process communications.

Keywords: Fault tolerant software, Petri nets, Occam, Concurrent processes, Conversations.

ACKNOWLEDGEMENTS.

There are a number of people who have helped during the course of this work and in the production of this thesis.

I would like to thank Helen Turner for typing the mathematical equations, Peter Miller for getting systems working when I could not and to Mike Spann for refreshments. A special thanks is due to Geof Carpenter for many stimulating discussions and his help in the preparation of this thesis. I would also like to thank the SERC for their funding over the past 3 years. Finally I am indebted to my supervisor Dr David Holding without whose constant encouragement and searching questions, this work would not have been possible.

LIST OF CONTENTS.

LIST OF FIGURES AND TABLES.

Table 4.5 Partition of Table 4.2 from t10 to t27 102

Chapter 1.

Introduction.

1.1 Introduction.

Throughout this thesis the terms fault, error and failure are used. It is important that the discussions are conducted with a defined terminology for the relevant concepts. The definitions given here are derived from those given in [1] :

a) a failure occurs whenever the external behaviour of a system does not conform to that prescribed by the system specification,

b) an error is a state of the system which, in the absence of any corrective action by the system, could lead to a failure which would not be attributed to any event subsequent to the error,

c) a fault is the adjudged cause of an error.

Due to the complexity of computer systems, it is generally impossible to obtain a system which is completely free from faults [2]. Failure of a system may

be caused by either hardware or software faults.

The concept of hardware fault tolerance has been studied for a long time [3]. Hardware structures have been developed which will cope, with a high degree of probability, with these faults. Hardware reliability has increased as component reliability improves, while due to the increased complexity of software systems, software faults have become more prevalent.

All software failures result from design faults [4]. The relative frequency of software errors compared with hardware errors reflects the increased logical complexity of software [4]. This complexity is due to the fact that machines used for hardware design have a relatively small number of possible internal states, making it usually possible to consider the hardware design as correct. In comparison to this even a small software system has an enormous number of different possible states, making it very difficult to justify the assumption that the software design is correct. Methods are being developed for increasing the correctness of a design [5], for ascertaining the correctness of a design using correctness proofs [6], and for introducing fault avoidance methods [7]. These methods are in an early stage of development and can at present be applied only to a limited set of tasks, such as proving the logic for a specific

operating system function. It is not thought that these methods could be applied to complete systems at present [8], although work on automating theorem proving may provide useful gains in productivity. Work in software fault tolerance has increased over the last 10 years in an attempt to prevent the increase in software complexity increasing software faults.

The requirement for systems to continue to operate satisfactorily in the presence of faults has lead to techniques for the construction of fault tolerant software systems. A fault tolerant system detects errors created as the effects of a fault and applies error recovery provisions in the form of abnormal or exceptional mechanisms and algorithms to continue operation and restore normal computations. These methods must be based on useful redundancy, this redundancy must be a redundancy of design [9].

Once error detection has taken place, the fault tolerant methods for software systems are usually classified into backward or forward recovery techniques [10]. Forward error recovery is achieved by making corrections to a system state containing errors so that normal operation can be resumed [10]. Backward error recovery restores the system to an error free state which occurred prior to the manifestation of the fault. Using this earlier state, the function

of the system is then provided by an alternative algorithm [11]. Forward error recovery techniques are generally used for recovering from predictable faults. In contrast, backward error recovery is used for unpredictable faults [12]. This thesis is concerned with the development of design methods for backward error recovery systems.

The recovery block mechanism [11,4] provides a backward error recovery scheme for general sequential systems. It uses a similar mechanism to the standby spares used in hardware systems [13]. If a fault is detected by an acceptance test, the system is restored to a previous correct state and control is transferred to a spare component.

In distributed concurrent systems the software can be partitioned into a number of processes, the partitioning often being performed on a functional basis [14,15]. Inter-process communications or information flow will take place through defined interfaces to the processes. These information flows are essential to the operation of the complete system. However, under fault conditions, errors may propagate through the inter-process channels. It is therefore essential to limit and control such communications. One way of limiting the extent of information flow is by the use of atomic actions [16]. An atomic action

can be defined as follows [17] : "The activity of a group of components constitutes an atomic action if there are no interactions between that group and the rest of the system for the duration of the activity". The conversation [4] is a backward error recovery mechanism using the idea of atomic actions to provide a fault structure for distributed concurrent systems.

The conversation is an extension of the sequential fault tolerant mechanism mentioned earlier, the recovery block. It encompasses a set of interacting concurrent processes by coordinating the recovery activity of the interacting processes into a common recovery structure. The faults this fault tolerant mechanism will deal with are unpredictable design faults which generate errors detectable with logical tests.

The conversation provides an error recovery mechanism which allows a specific , predefined, subset of a set of processes to be included within the fault tolerant mechanism. The conversation is usually used to protect a specific function or part of the system. The boundary of the conversation must be designed with care to ensure that all required processes are included by the conversation. If conversations are nested, it must be ensured that this nesting is designed properly.

In this thesis the problem of designing the conversation mechanism for sets of distributed processes is related to the problem of identifying and protecting certain sequences of states of the set of processes. A solution is proposed in which : (i) the system is modelled as a set of communicating sequential processes and described in the occam programming language, (ii) the model is transformed into a Petri net and the network state and state reachability space are determined and (iii) a method is presented for identifying conversation boundaries within the state reachability space of the system. An implementation of the conversation scheme is then shown and is described in the occam programming language.

A further problem in distributed concurrent systems concerns the inter-process communications [18]. In a system consisting of a set of communicating processes, as outlined above, if one of the communicating processes fails to reach a communication point, the other process involved in the communication will become deadlocked [19]. In a distributed system, the conversation scheme relies on communications for its correct operation. It is therefore essential to prevent processes becoming deadlocked. By using Petri nets to model the communication primitives, a scheme is discussed which decreases the likelyhood of processes deadlocking [20].

1.2 Summary of Thesis.

Chapter 2 of the thesis sets out the objectives of the research presented here. It presents a survey of previous work in this field of research. It also identifies a number of tangible goals which were achieved to obtain the set objectives.

A crucial concept in the subsequent chapters of this thesis is the idea of the state of a process or system. The design method presented uses the state of a system to identify the placement of a fault tolerant mechanism within that system. A method was therefore required for identifying the state of a system. In chapter 3 Petri nets [21] are used for the state definition of a system consisting of a set of communicating sequential processes. The concurrent programming language occam [22] is described and related to a Petri net model, thus making it possible to model a concurrent system described in occam, including information about the state of the interacting processes.

A formal definition of Petri nets and the Petri net structure is given. The concept of net state is introduced and mapped onto the reachability tree of a Petri net. Petri net techniques are then used to model sequential software systems. The state of the Petri net is mapped to the state of the software system and the transition from one state to the next

shown on the reachability tree. The model for the sequential system is then extended to incorporate the additional features required in concurrent software systems.

In chapter 4 the fault tolerant mechanism for concurrent processes, the conversation, is introduced and a number of design problems identified. Before a conversation can be constructed the boundaries of the conversation must be identified. It is shown that analysis of process state and state transitions can be used to identify conversation boundaries automatically [23]. The model developed in chapter 3 is used to determine the states within a conversation. Particular emphasis is placed on identifying the state of a conversation because these states are used directly in the implementation of the recovery mechanism. The method which is developed in this chapter will identify all processes within a specific conversation. Perhaps more important from the point of view of design, the method can be used to provide fault tolerance for a particular function or subsystem, because it allows the designer to identify the boundary of a conversation, or properly nested set of conversations which enclose the particular feature.

An implementation of the conversation scheme is given in chapter 5. The special facilities of the

language occam can be used to generate a conversation framework. This is independent of the application and due to the nature of the language is independent of the number of processes involved. By using the design rules developed in the previous chapter, the conversation framework is incorporated into a distributed concurrent control example.

When processes are involved in communications with other processes in the system there is always a possibility of a communication failure. In certain circumstances these failures could lead to a loss of system integrity. Chapter 6 considers some of the problems involved in designing reliable communications for concurrent systems. The requirements for reliable communications are described for the different types of communication primitives, and their relative advantages and deficiences are highlighted. A system is discussed which gives a higher degree of reliability than existing systems [24]. This system is modelled again using Petri nets and implemented in the concurrent programming language occam.

Chapter 7 summarises the achievements of the research and draws a number of conclusions about these. Also in this chapter a number of areas for further research are suggested.

Chapter 2.

Aims and Objectives of the Research.

2.1 Introduction.

The major objective of the research in this thesis was to produce a design method for the production of fault tolerant software for distributed systems. The approach used to recover from unanticipated faults is state restoration; this ensures the comprehensive removal of errors. The system must be reset to a state which has already occurred during the operation of the system. If the system can be restored to a state which it occupied prior to the occurrence of a fault then errors resulting from that fault will have been removed.

To attain the objective of this thesis, a number of goals had to be achieved. Firstly a mechanism had to be identified which could provide fault tolerance in a distributed system. The recovery block scheme [4] is a well known and proven fault tolerant mechanism for sequential systems. The conversation scheme proposed by Randell [4] is an extension to the recovery block scheme, for concurrent systems. Backward error recovery using a conversation is relatively straightforward since it uses a planned recovery line

[17]. This method of recovery does not suffer from the disadvantages of unplanned recovery line methods. Unplanned recovery line methods require a complex mechanism to locate the recovery lines, even with this mechanism non-identification of a recovery line is possible and the possibility of the domino effect occurring is high.

A conversation limits the extent of the migration of an error between a number of interacting concurrent processes [4]. This is achieved by placing a boundary around a set of these interacting processes. The boundary has four edges: an entry line prohibits a process from rolling too far back when in recovery and holds a set of correct prior states, which are used for state replacement. The exit line is a synchronising line between the processes within the conversation; the state of each process is checked at this line and if found in error will cause all the processes in the conversation to roll back to the entry line. The two side walls prohibit the passing of information (either into or out from the conversation) to processes outside the conversation.

When designing a conversation to increase the fault tolerance of a system two major of problems arise: identification of constituent processes and identification of conversation boundaries. For a

given set of events a system consisting of a set of concurrent processes will have a subset of these processes interacting with each other. Thus, between any two specific events in a system only a subset of the processes within the system will be interacting with each other. If a fault tolerant boundary is required between these two events, only those processes interacting with each other need to be included within the boundary. The remaining processes in the system will need to be excluded from the fault tolerant activity. Each conversation will thus contain a characteristic subset of processes. In general, the processes in one conversation will not be the same subset as those in another conversation. A method is required which identifies these processes for any given conversation.

To increase the usefulness of a fault tolerant mechanism nesting should be possible. Conversations do allow nesting. However, if conversations are not properly nested one or more processes could leave an inner conversation making it impossible for an outer conversation to recover fully [4]. Care must therefore be taken when designing a system with nested conversations [25].

2.2 Placement of Conversations.

This thesis develops a method of identifying conversations by examining the state of the processes in the system. For an error recovery technique to be able to restore a prior state of a system, a record of that state must have been preserved. State restoration is certainly a possible recovery method for software systems, since the notion of state is inherent in such systems.

A number of papers have been published on the recording of state information in a dynamic manner, that is, where the state of a system is recorded as the system executes. In [26] the state of the processes is defined using occurrence graphs [21]. These graphs are generated by the system itself as it executes. Each process keeps a record of that part of the growing occurrence graph in which it is involved. If recovery is to be performed a process must send a fail message to other processes in the system determined by its part of the occurrence graph. A process receiving a fail message must stop its normal operation and send fail messages determined by its graph. Due to the independent nature of the processes and their restorable places on the occurrence graph, the probability of multiple rollbacks occurring is high [17] and the searching for a set of restorable states back to the beginning of the software is possible; this is the domino effect [4].

Barigazzi et al. [27] avoid the domino effect by keeping only one copy of previous states for each process. This does however, have the disadvantage of not allowing fault tolerant blocks to be nested. In their proposal states are saved either when a local counter reaches zero count or because states have been saved in another process and a communication has occurred between the two processes. The communications in the latter case, for saving process state, does bring a further source of unreliability into the system. It is probable that the recovery mechanism will be required to protect certain functional aspects of the processes. In this method no consideration is made of the functional aspects of the processes.

Russell [28], has discussed state restoration in systems with the restriction that process interactions are unidirectional. It further postulates that a system which is not domino-free may still not exhibit the domino effect.

A shared memory system is considered by Kim [29]. Here a monitor [30] is used as the sole control of process interactions. It assumes that there is a centralising process which manages restoration of monitors and coordination of process rollbacks. The domino effect is eliminated by placing an additional constraint on the system, that is, suspicion of received

messages being in error is prohibited. That is s-propagations [29] are not allowed.

In this thesis a method for identification of planned recovery lines is proposed. A method is developed in which the state behaviour of the system is mapped onto a state reachability tree. By using this idea of state the boundary lines are defined. The exit line is the set of states belonging to the processes prior to leaving the conversation. By relating the states between the entry and exit lines to the processes, the minimum set of processes required for a given conversation can be determined. It is also shown that by using the state reachability tree of the system it can be determined whether two or more conversations are properly nested or not.

It is therefore proposed in this thesis that the design of fault tolerant distributed software, using the conversation scheme, can be simplified by consideration of the system state.

The simplification can not be considered until a method for defining the state of a distributed system is specified. A system state is a point of state space defined by a vector of values assumed by system variables. Any assignment or communication operation corresponds to a transition from one system state to another. The state representation used in this thesis

is achieved using Petri net techniques [31]. Petri nets provide several advantages as a system modelling technique. First, the overall system structure and behaviour is easy to understand due to the precise and graphical nature of the representation scheme. Second, the behaviour of the system can be analysed using Petri net theory and analytical tools [32,33].

A model of the distributed system must be defined before the state of the system can be defined. To model a system it is essential to have an complete specification of the system. The specification should be complete and unambiguous [5]. Work on formal methods for system specification [5,34,35] is intended to satisfy these objectives, but these methods are at an early stage of development and not widely used. In this thesis the description of the distributed system is achieved using the concurrent programming language occam [22]. Although it is a programming language, occam can also be used to specify distributed systems [36].

The modelling tool should have a formal definition, enabling the model of a system to be constructed in a rigorous manner and, in addition, allowing analysis of a system. Before the model for a concurrent system can be constructed modelling techniques for a sequential system are needed. These modelling

techniques were again constructed using Petri nets from a general specification of sequential primitives and constructs. This was constructed by firstly modelling the basic primitives which are the most fundamental features of a computational machine; assignment, input and output.

Thus, by using occam as a specification language for concurrent systems and modelling the specification using Petri nets, a state-transition model of a concurrent system was developed. This model was then used to identify boundaries for the placement of conversations.

2.3 Implementation of Conversations.

Once the conversation has been designed the inevitable step is to implement the design. A number of implementations have been proposed for the conversation scheme.

In [37], the name-linked recovery block is introduced. The paper looks at both asynchronous and synchronous conversations. Multiprocess recovery blocks are also considered. The ideas presented are only slight advances towards a full implementation over those of the original proposal [4], and are proposed constructs rather than implementations.

Kim [25], assumes that interprocess communications take place through monitors, i.e. a shared memory system, and presents a number of possible implementations, based around the language Concurrent Pascal [38]. Extensions to the language Concurrent Pascal are proposed which it is suggested will help in the structuring of recoverable process interactions. This method restricts the design to a shared memory system and has the disadvantage of requiring specific extensions to be made to the language Concurrent Pascal before any design can be implemented.

This thesis proposes an implementation of the conversation which uses the concurrent programming language occam, with no extensions. Although occam assumes that interprocess communications take place by message passing through channels [39], this is not a real restriction on the design since it is possible to change a message passing system into one which uses shared memory [40].

The first aim of the implementation was to provide a system which was independent of the application; that is, once the structure for the conversation was constructed, this could be used for any application by changing the algorithms inside the structure, but not the structure. Secondly, it should be clear where a conversation starts and finishes for each pro-

cess in the conversation. The acceptance tests for the conversations should be identified clearly, allowing changes to be easily made; this is the only part of the conversation structure which may require alteration from application to application. The implementation obviously must allow synchronised exit from the conversation. Finally, it should be possible to nest conversations.

It is shown here that these aims are meet by the proposed implementation. An example is given to highlight the main features of the implementation.

2.4 Communications Failures.

A tacit assumption for the system considered is that communications do not fail. This may not always be the case [41]. The final part of this thesis is concerned with this subject. The problem considered here is that of communication failure due to non-receipt of message, i.e. processes failing to reach communication points, failure of communication medium. The main objective for this is to ensure that processes do not deadlock [19] due to communications failure.

To achieve this objective each of the communication types, for message passing, are investigated and are shown to be deficient when applied in a fault

tolerant situation. A suggestion is made for the placement of a timeout mechanism [42] which it is argued gives a higher reliability system from those proposed before. Each communication primitive used in message passing is analysed using a Petri net state-transition model. By using state reduction on the state-transition model a boundary for a timeout mechanism is identified.

2.5 Discussion.

The systematic design of a fault tolerant system requires a method for the placement of the fault recovery mechanism within the system. Previous designs for fault tolerant distributed systems do not consider explicitly the placement problem. In the method proposed in this thesis the distributed system is described using the concurrent language occam, this description is mapped onto Petri nets allowing the state of the system to be defined. It is shown that from the definition of state the boundaries of conversations can be identified and thus placed in the system.

The implementation of a fault tolerant structure should be easy for the designer to incorporate into systems, it should not add complexity to the system making the probability of design errors greater.

Implementations previously proposed have required extensions to languages, put restrictions of the final system structure and in some cases increased the complexity of the system. The implementation given in this thesis requires no extensions to the language; it is application independent and is structured such that system complexity is not increased too much.

Chapter 3.

System Model.

3.1 Introduction.

In many fields of study, a phenomena is not studied directly but indirectly through a model of the phenomena. A model is a representation of what are felt to be the important features of the system under study. By the manipulation of the representation, it is hoped that new knowledge about the modelled phenomena can be obtained without the danger, cost or inconvenience of manipulating the real phenomena itself.

Computer systems are often complex, large, systems of many interacting components. Each component itself can be complex, as can its interactions with other components in the system. Thus, one fundamental idea is that systems are composed of separate interacting components [43]. Each component may itself be a system, but its behaviour can be described independently of other components of the system, except for well-defined interactions with other components. These components may exhibit concurrency. In order to model computer systems a tool is required which will cope with the interacting components of the system and allow concurrency to be represented.

The model used throughout this thesis is a Petri net [44,31] description of the system.

A formal definition for the basic Petri net has been specified [31] together with the Petri net graph allowing analysis of the system to be carried out. A formal definition for the state of the Petri net graph is given. Thus, by using Petri nets it is possible to represent the state of the system being modelled. Using this formal definition, the structures present in sequential and concurrent software systems are modelled.

Modelling a concurrent system requires a number of additional constructs not required for modelling sequential systems such as parallelism and communications [45]. These features are incorporated in the concurrent notation C.S.P. [46], and concurrent language occam [22]. It is shown here that these additional constructs can be modelled using Petri nets.

The models are analysed using the reachability tree of the graph [31]. The reachability tree is built up from the formal definitions of Petri nets.

3.2 Petri Net Structure and Graph.

Petri nets are composed of two fundamental com-

ponents : a set of places, P, and a set of transitions, T. To define the relationship between the places and the transitions two functions connecting transitions to places are defined: I the input function, and O, the output function. The Petri net is defined completely by its places, transitions, input function, and output function [31].

The Petri net can be used to model a computer program by representing sequences of statements (actions) by transitions, the points between actions by places, and the value of a program counter by the location of a Petri net token [47]. These actions can be local to a process, such as assignment, or more complex actions involving more than one process, such as communication.

3.2.1 Petri Net Structure.

Peterson [31] defines a number of important properties of Petri nets which enable them to be analysed. The definitions are based on bag theory [48], an extension of set theory. A Petri net structure, C, is a four-tuple, C = (P,T,I,O).

$P = \{p_1, p_2, \ldots, p_n\}$ is a finite set of places, $n >= 0$. $T = \{t_1, t_2, \ldots, t_m\}$ is a finite set of transitions, $m >= 0$. $I : T \longrightarrow P$ is the input function, a mapping from transitions to bags of places. $O : T \longrightarrow P$ is the output function, a mapping from transitions

to bags of places. The set of places and the set of transitions are disjoint, $P \cap T = 0$. (An example of a Petri net structure is given in fig 3.1.)

$C = (P,T,I,O)$

$P = \{p_1, p_2, p_3, p_4, p_5\}$

$T = \{t_1, t_2\}$

$I(t_1) = \{p_1\}$ $\quad O(t_1) = \{p_2\}$

$I(t_2) = \{p_2, p_4\}$ $\quad O(t_2) = \{p_3, p_5\}$

Fig 3.1 Petri net structure consisting of two transitions and five places.

3.2.2 Petri Net Graph.

From the definitions of the Petri net structure given above a diagram of the modelled system can be specified, the Petri net graph.

A Petri net graph G is a bipartite directed multigraph (multiple arcs between the two kinds of nodes), $G = (V,A)$, where $V = \{v_1, v_2, \ldots, v_s\}$ is a set of vertices and $A = \{a_1, a_2, \ldots, a_r\}$ is a bag of directed arcs, $a_i = \{v_j, v_k\}$, with $v_j, v_k \in V$. The set V can be partitioned into two disjoint sets P and T such that $V = P \cup T$, and $P \cap T = 0$, and for each directed arc, $a_i \in A$, if $a_i = (v_j, v_k)$, then either $v_j \in P$ and $v_k \in T$ or $v_j \in T$ and $v_k \in P$. The graph contains two kinds of node, place nodes and

transition nodes. Places (p) are represented by circles. Transitions (t) are represented by bars. The Petri net graph for fig 3.1 is shown in fig 3.2.

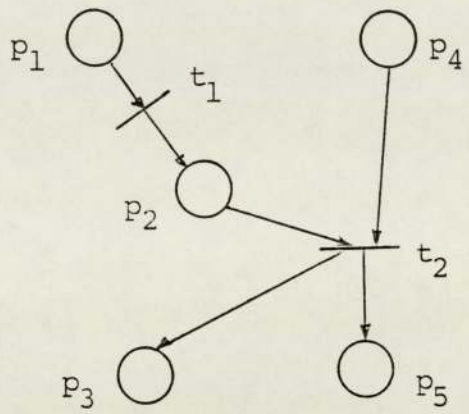


Fig 3.2 Petri net graph for fig 3.1.

3.3 Representing State.

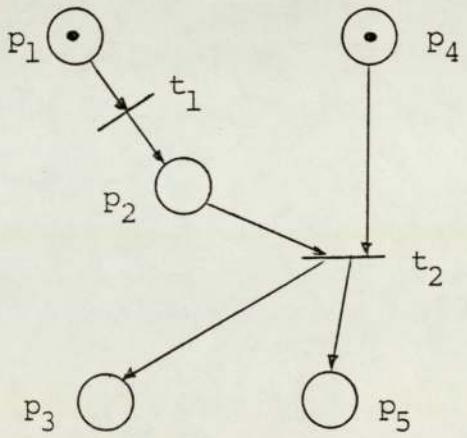
Each component of a system has its own state. The state of a component may change with time. The state of a component thus depends on the past history of that component. The concept of state is very important to modelling systems. The state gives information which enables future actions to be predicted. The concept of state is mapped onto Petri nets by marking the places on a Petri net graph with tokens. This can then be related back to the state of the system the Petri net is modelling.

3.3.1 Petri Net Marking.

A marking $\bar{\mu}$ is an assignment of tokens to the places of a Petri net. A marking μ of a Petri net C = (P,T,I,O) is a function from the set of places P to the nonnegative integer N.

$$\mu : P \longrightarrow N.$$

The marking μ can be defined as an n-vector, $\mu = (\mu_1, \mu_2, \ldots, \mu_n)$, where $n = |P|$ and each $\mu_i \varepsilon N$, $i = 1,\ldots,n$. The state of a Petri net is defined by its marking. A marking for the Petri net graph of fig 3.2 is shown in fig 3.3.



$\mu = \{1,0,0,1,0\}$

Fig 3.3 Marking of fig 3.2.

The marking μ can also be defined as a vector $\mu = \{p_1, p_b, \ldots, p_n\}$, where $p_i \varepsilon P$ and these are the only

places marked at the particular instant. Thus, for fig 3.3 above $\mu = \{1,4\}$.

3.3.2 Execution Rules.

An action in software can only take place when all required information is available, i.e. x := y+z must have both y and z before the assignment can take place. Similarly, for a transition on a Petri net graph, a transition $t_j \in T$ in a marked Petri net $C = (P,T,I,O)$ with marking μ is enabled if for all $p_i \in P$:

$$\mu(p_i) \geq \#(p_i, I(t_j))$$

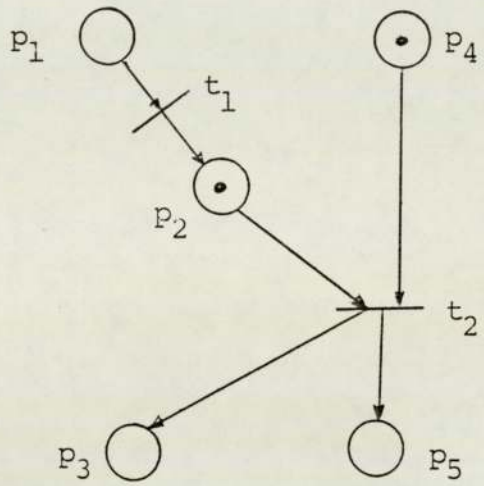
A transition fires by removing all of its enabling tokens from its input places and depositing into each of its output places one token for each arc from the transition to the place.

A transition t_j in a marked Petri net with marking μ may fire whenever it is enabled. Firing an enabled transition t_j results in a new marking μ' defined by :

$$\mu'(p_i) = \mu(p_i) - \#(p_i, I(t_j)) + \#(p_i, O(t_j))$$

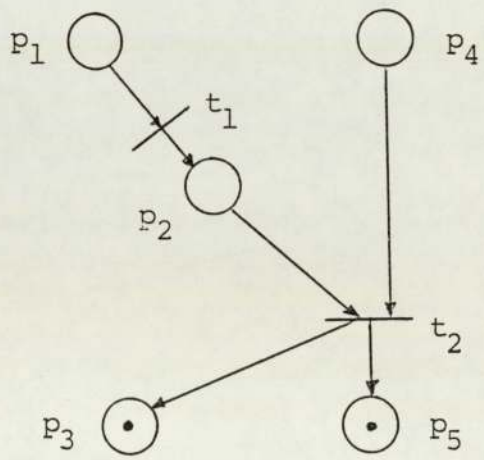
The firing of a transition represents a change in the state of the Petri net by a change in the marking of the net. The state space of a Petri net with n

places is the set of all markings, that is, N^n. An example of transitions firing is shown in figs 3.4 and 3.5.



$\mu = \{2,4\}$

Fig 3.4: Transition t_2 enabled.



$\mu = \{3,5\}$

Fig 3.5: Result of t_2 firing.

A description and formal definition of the

modelling tool, Petri nets, have been given. It has been shown that from a mathematical definition of the net a graph may be constructed of the model. Using this graphical model of the net, state was introduced and the rules for changing the state were given.

In the following sections Petri nets are used to model software systems. It is shown that both sequential and concurrent systems may be modelled using Petri nets.

3.4 Sequential Processes.

Sequential programs can be written using six primitive processes [49]:

INPUT

OUTPUT

ASSIGNMENT

SEQUENCE

SELECTION

REPETITION

Each basic primitive can be modelled as an element of a Petri net, such that the state of the process is represented by the marking of the places of the net, μ.

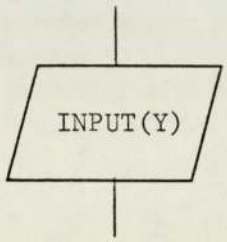
The Petri net models for these primitive constructs are shown in figure 3.6. It follows that any

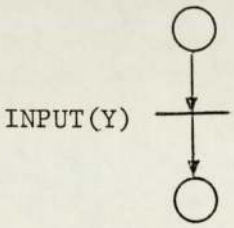
sequential program can be modelled as a Petri net by combining a number of these primitives.

Process Program Flow Diagram Petri Net Model

a) INPUT

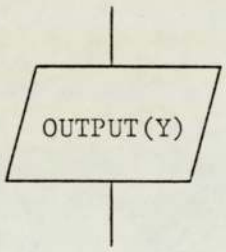
INPUT(Y)

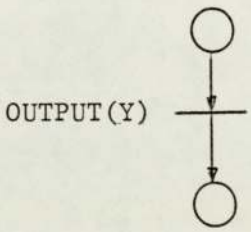




b) OUTPUT

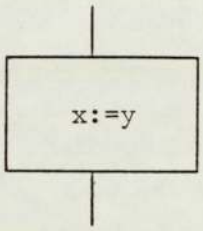
OUTPUT(Y)

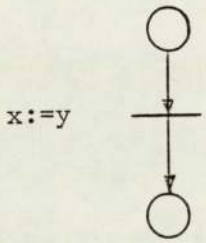




c) ASSIGNMENT

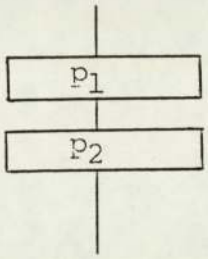
x := y

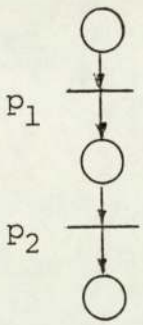




d) SEQUENCE

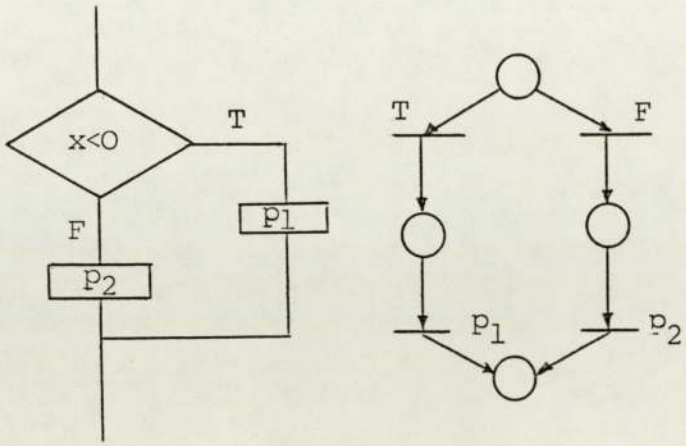
SEQ
P1
P2





e) SELECTION

IF x<0 THEN P1
ELSE P2



f) REPETITION

SEQ
k:=1
WHILE k<100
SEQ
P1
P2

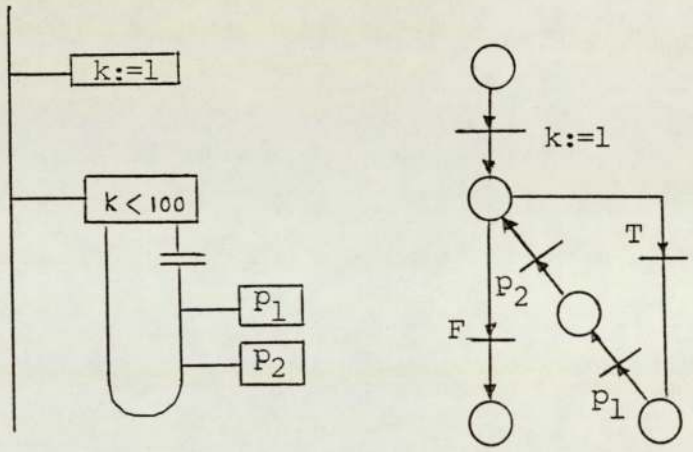


Fig 3.6 Petri Net Models of Sequential Software Constructs.

From the above it can be seen that every place has a unique output transition, except for places which precede decisions (e,f); these places have two output transitions corresponding to TRUE and FALSE

outcomes of the decision predicate. The choice as to which arc to take can be made non-deterministically or by some outside influence (such as the designer).

3.5 Concurrent Systems.

The sequential constructs described above are sufficient to describe sequential systems and the sequential parts of concurrent systems. However, concurrent systems can not be fully described using the sequential concepts alone. Additional constructs must be introduced to describe parallelism, inter-process communications and inter-process synchronisation [38].

Hoare has introduced the notation C.S.P. [46,50], which allows the sequential and parallel composition of communicating processes. This notation also unifies input, output and inter-process synchronisation in a simple mechanism for internal (inter-process) or external (input,output) communications. The problem of inter-process communication is handled by inter-process message passing.

3.5.1 Axioms of C.S.P. :

The basic axioms of C.S.P. are as follows :-

a) A parallel command based on Dijkstra´s parbegin [51] is used to specify concurrent execution. All processes start simultaneously, the command is only

completed when all processes have completed their execution. Communication via updating global variables is not possible.

b) A simple form for input and output commands is used; it is also used for communications between concurrent processes.

c) For communication between two processes to take place there must be a logical pairing of input and output in which :

i) the receiver process must identify the transmitter process,

ii) the transmitter process must identify the receiver process,

iii) the data object to be transmitted must be of the same type as the data type expected in the receiver,

iv) there is no buffering in the data channel: both transmitter and receiver must be ready for communication. This enforces synchronisation between the processes and either process may be delayed until the other process is ready.

d) Dijkstra's guarded commands [52] are used as sequential control structures and the means of introducing and controlling non-determinism between processes.

e) Input commands may appear in guards. This then acts as an alternative constructor. A process which is guarded by an input is not executed unless the process at the other end of the communication is waiting to output. If several input guards of a set of alternatives have ready sources, only one is selected arbitrarily.

By using the above proposals the programming language occam has been developed, which is relatively simple to understand and introduces input, output and concurrency as explicit primitives.

3.5.2 Occam.

Occam enables a system to be described as a collection of concurrent communicating processes. The communications is achieved via channels.

In occam each primitive process and construct occupy a single line. The components of the constructs are indented.

3.5.2.1 Primitive Processes.

Occam has three primitive processes: assignment,input and output :

v := e assign an expression e to a variable v.

c ! e output expression e on channel c.

c ? v input variable v from channel c.

3.5.2.2 Constructs.

The primitive processes can be combined in a number of ways by using the constructs available in occam.

3.5.2.2.1 Sequential Constructs.

Sequential (SEQ): primitives following this construct are executed in sequence one after the other.

Conditional (IF): this construct is followed by a condition. If the condition is true the primitives encompassed by the construct will be executed.

Repetition (WHILE): a condition follows the WHILE and the primitives encompassed by the construct are executed until the condition is false.

3.5.2.2.2 Parallel Constructs.

Parallel (PAR): all processes in the scope of the construct are performed concurrently. The construct terminates when all constituent components have terminated.

Alternative (ALT): the alternative constructor chooses one of its components for execution. Each component

process has a guard which is an input, with an optional condition. The earliest process which is ready to be executed is chosen; the guard is executed followed by the guarded process. If more than one guard is satisfied the choice as to which alternative is taken is arbitrary.

Replicator: replicators are used to describe collections of similar processes. They can be used with the constructs PAR, SEQ and ALT.

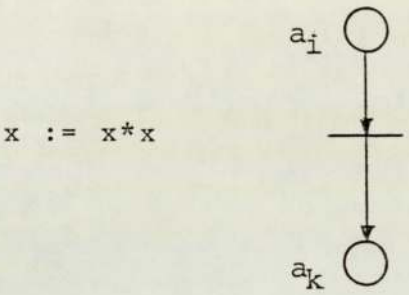
3.5.3 Modelling Concurrent Software.

Concurrent programming languages such as occam allow the user to model concurrent systems. Such systems are composed of separate, interacting components. Each component may itself be a process, and its behaviour can be described independently of the other components of the system, except for well-defined interactions with other components. In this section a state model of concurrent software is derived using Petri net techniques: Petri nets have been previously used to generate state models of sequential software [53]. To deal with concurrent systems Petri net models have to be developed for concurrent constructs, such as parallel processes, synchronised communications and asynchronous ALT processes. By incorporating these concurrent constructs a unified Petri net-

based state model of sequential and concurrent software was developed.

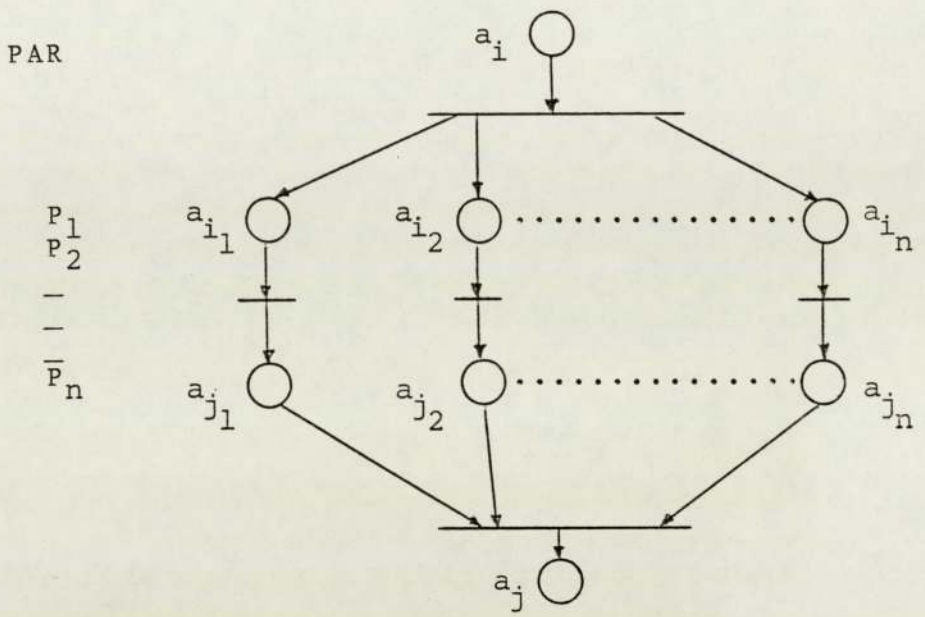
Assignment.

Assignment is an action which involves a single process only: it can be modelled as a transition with single input and output arcs.



Parallel.

In the parallel construct all actions are initiated simultaneously. The construct does not terminate until all parallel processes have terminated.

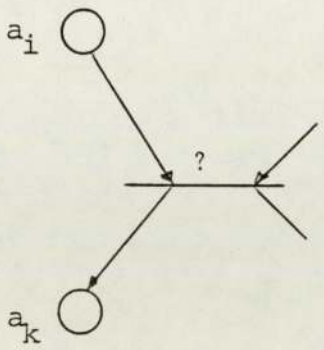


Communications.

For the action of communication two processes are involved. One sends the information and one receives it. Thus a transition modelling a communication requires at least two input arcs, one from each process involved in the communication and at least two output arcs, again one to each process. To distinguish input and output actions the occam notation for input and output on the transitions is used.

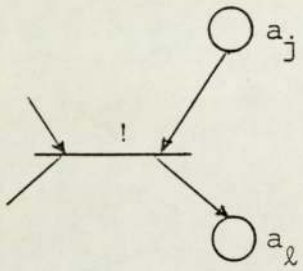
Input.

comm.chan ? var



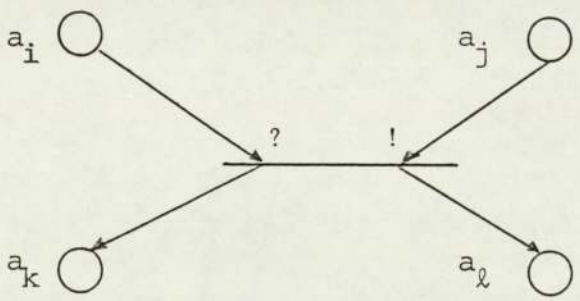
Output.

comm.chan ! exp



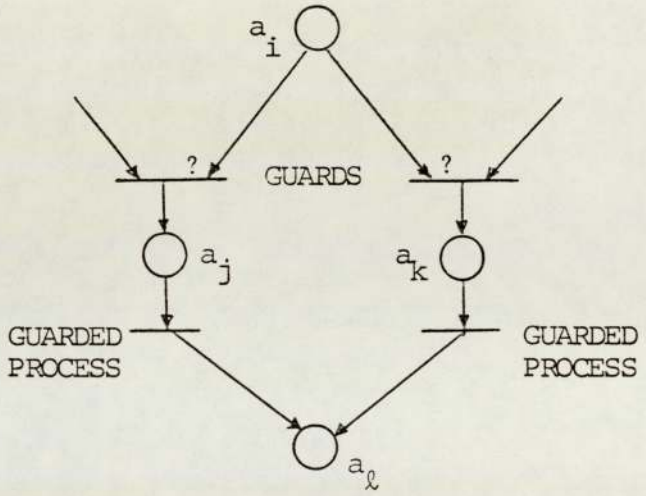
These two actions, input and output, always appear in pairs in the system. In occam type systems where the communications is synchronised, the same transition will be shared by both processes involved in the communication.

Process A Process B



Alternative.

The alternative constructor consists of a number of guarded processes. The guard of each component process comprises an input, and an optional input expression. When an input is ready, the corresponding guard will be satisfied and the guarded process executed. If more than one guard is satisfied simultaneously the choice as to which alternative is taken is arbitrary.



The previous sections have developed a model, firstly for sequential systems and then for concurrent systems described as a set of communicating sequential processes, using Petri nets. It has been shown that parallelism is inherent within the model, interprocess communications can be represented and that asynchronous alternatives may be included within the model.

Section 3.3 showed how state could be introduced into the model. To be able to interpret the changing

sequence of state through which a system evolves, a means of representing the evolving state must be used. The next section introduces such a technique, the reachability tree.

3.6 Reachability Tree.

It has been shown that by using the basic definitions of Petri nets and a formal concurrent language, occam, it is possible to model the constructs required for concurrent systems. A model can be constructed of the concurrent software for a complete system, by simply connecting the nets for each component construct. In such a network each place on the Petri net will have an associated state, and each transition will correspond to a component process. The state of the software is the marking of the places on the Petri net.

The state of a Petri net is defined by its marking. To analyse the dynamics of the system the markings (states) of the model must be mapped as they are changed. In what follows a number of functions are formally defined which enable the reachability space to be constructed. The method used here is the reachability tree [31].

The reachability tree consists of nodes which represent markings of the Petri net connected by arcs

which represent the firing of transitions. Each node on the tree will have an associated state, and each arc represents the transition of the corresponding component process. Each node on the reachability tree is labelled with a marking, arcs are labelled with transitions.

The initial node (root of the reachability tree) is labelled with the initial marking. Given a node x in the tree, additional nodes are added to the tree for each marking that is directly reachable from the marking of the node x. This is given by the next state function δ . In the modelling of software systems the reachability tree can be transformed to : nodes represent the state of the system and arcs represent the possible changes in state resulting from performing actions.

3.6.1 State Dynamics of a Petri Net.

The next-state function $\delta : N^n \times T \longrightarrow N^n$ for a Petri net $C = (P,T,I,O)$ with marking μ and transition $t_j \in T$ is defined if and only if :

$$\mu(p_i) \geq \#(p_i, I(t_j))$$

for all $p_i \in P$.

If $\delta(\mu, t_j)$ is defined, then $\delta(\mu,t_j) = \mu'$, where :

$$\mu'(p_i) = \mu(p_i) - \#(p_i, I(t_j)) + \#(p_i, O(t_j))$$

for all $p_i \in P$.

From figs 3.4 and 3.5, $\mu = \{2,4\}$ and $\delta(\mu,t_2) = \{3,5\}$.

For a Petri net C = (P,T,I,O) with marking μ, a marking μ' is immediately reachable from μ if there exists a transition $t_j \in T$ such that $\delta(\mu,t_j) = \mu'$.

For each transition t_j on the Petri net which is enabled in the marking for node x, a new node on the reachability tree with marking $\delta(x,t_j)$ is created, and an arc labelled t_j is directed from the node x to this new node. This process is repeated for all new nodes. Continuing this process will create the entire state space. This is known as the reachability set $R(C,\mu)$. In practice repetitive constructs will usually bound the reachability tree.

The reachability set $R(C,\mu)$ of a Petri net C with marking μ is defined to be all markings which are reachable from μ. A marking μ' is in $R(C,\mu)$ if there is any sequence of transition firings which will change marking μ into marking μ'.

The reachability set $R(C,\mu)$ for a Petri net C = (P,T,I,O) with marking μ is the set of markings defined by :

1. $\mu \in R(C,\mu)$

2. If $\mu' \in R(C,\mu)$ and $\mu'' = \delta(\mu',t_j)$ for some $t_j \in T$, then $\mu'' \in R(C,\mu)$.

$R(C,\mu)$ for fig 3.3 = {(1,4),(2,4),(3,5)}. The corresponding reachability tree is shown in figure 3.7.

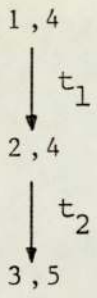


Fig 3.7 Reachability Tree for fig 3.3.

A path from the initial marking (root) to a node in the tree corresponds to an execution sequence and can be defined by the extended next-state function $\delta(\mu,\lambda)$. The extended next-state function is defined for a marking μ and a sequence of transitions $\sigma \in T^*$ by :

$$\delta(\mu, t_j\sigma) = \delta(\delta(\mu, t_j), \sigma) = \delta(\mu,\lambda)$$

For fig 3.3, let $\sigma = t_1,t_2$ and $\mu = \{1,4\}$
then $\delta(\mu,\lambda) = \{3,5\}$.

The definitions given above are concerned with the dynamics of the Petri net. They enable the state

space of a Petri net to be calculated given an initial state. These states can be used to construct the reachability tree of the net. The reachability tree represents the whole reachability set of a Petri net.

3.7 Discussion.

This chapter has built up a number of important ideas used throughout the rest of this thesis. It has demonstrated that Petri nets can be used as a notation for state space and space reachability in asynchronous concurrent systems. It has examined software constructs for SEQ, PAR and asynchronous processes. A state-transition description of occam processes was derived. Concurrent system models were mapped onto state-transition models and the state reachability was determined. Finally, the state of software was mapped onto the state reachability tree of the Petri net model.

Chapter 4.

The Structured Design of Conversations.

4.1 Introduction.

A fault is the mechanical or algorithmic cause of an error in a system [10]. Despite the adoption of fault avoidance techniques, faults still occur in constructed systems. These faults can occur, for example, due to the unavailability of fault-free hardware components, or because of the complexity in system software. The non-deterministic nature of concurrent systems, consisting of a number of asynchronous components interacting with each other make exhaustive testing of software for such systems impossible [17]. Yet in many applications it is essential that the system continues to operate correctly, even in the presence of faults [54]. There is therefore a need for fault tolerant software.

Fault tolerance should be based on the provision of useful redundancy [9]. In a hardware system this is achieved by duplicating system components and diverse design. Tolerance to faults in software can only be achieved by redundancy of design : replacing a faulty software module by an identical module would

just cause the same fault [8].

A further problem in designing error detection and recovery capabilities into a concurrent program structured in the form of a collection of cooperating asynchronous processes, arises from the possibility of error propagation through process interaction [25]. An error in one process may produce a fault in another process and lead to an error in the second process. It is therefore necessary to bound the extent of error propagation and to introduce a coordinated recovery in all processes involved.

In this chapter the recovery mechanism for fault tolerant systems is considered. This is divided into two major sections: one for sequential systems (recovery blocks) and one for concurrent systems (conversations). The mechanisms for error detection and error control are considered (test line, acceptance test and roll back) and a need for desgin tools identified.

It is the aim of this chapter to develop a method of identifying conversations to protect either processes or functions within a system and to provide analysis and design tools so that the designer can use such a method. The object was to develop a method for the systematic design of conversations for that class of systems which can be modelled using the SEQ, PAR

and asynchronous constructs available in occam.

The goals which were satisfied to meet this objective were: the derivation of a state model and state reachability model for the systems using Petri net techniques (as developed in chapter 3), the identification of local transitions, sequences of which can be protected by recovery blocks and inter-process communications/transitions, which can only be protected by conversations, the identification of conversation boundaries (by reducing the reachability tree), identification of processes within a conversation (by relating processes to state-transition attributes), the identification of conversation boundaries to the control mechanisms (test line etc.) and a specification of how to design a conversation to protect a particular function.

4.2 Error Detection and Recovery.

4.2.1 Sequential Systems.

The recovery block scheme [4,55] has been proposed as a method of introducing redundancy into the software of computing system, in the form of stand-by spares, in order to provide tolerance against faults.

The recovery block scheme for error recovery is based on the idea that programs are written in functional blocks. Although these blocks, which are

assumed to be non-redundant, may have been designed carefully and tested to some extent, design faults could still be present. It is therefore necessary to design around these blocks, known as the 'primary blocks', a mechanism which will deal with these faults. A recovery scheme for such a system will be designed in a number of steps.

The first step in this process is to provide a means of detecting the errors caused by the block. A process known as an 'acceptance test' is incorporated into the system to check on the correctness or reasonableness of the results calculated by the primary block. Thus:

primary block

acceptance test

The acceptance test will consist of a sequence of statements which will raise an exception if the state of the system is not acceptable.

The next stage in designing a structure to provide fault tolerance is to consider a suitable method of error recovery if the primary block fails its acceptance test. Since the precise time at which the errors will be generated is not known the most suitable prior state for restoration is the state of the

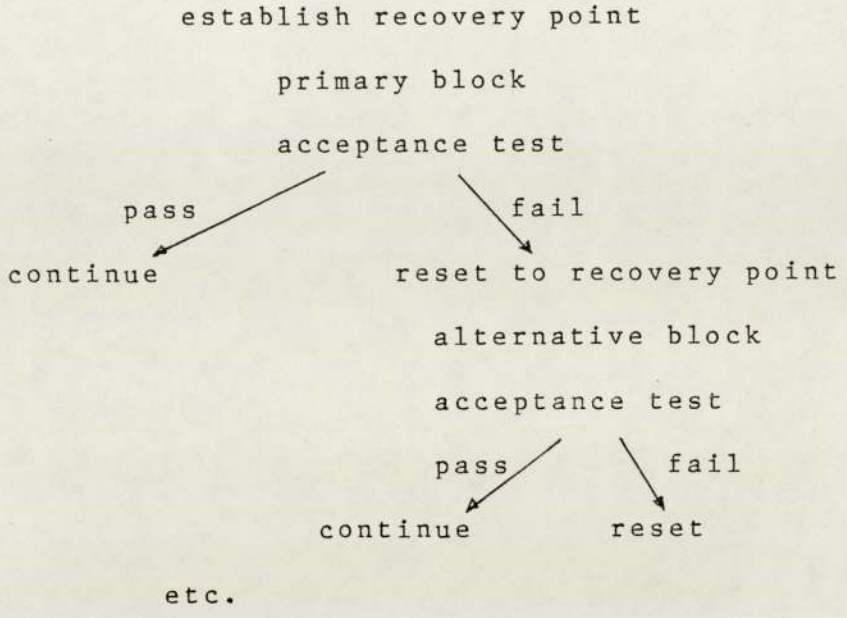
process that existed just before entry to the primary block. A recovery point can be established at this point by the introduction of an additional process 'establish recovery point' before the primary block.

establish recovery point

primary block

acceptance test

If an error is detected by the acceptance test the primary block will be restored to the state at the recovery point, a retry of the primary block may be useless since the same fault could result in an exception being raised again. What is required is a secondary block which will produce results that pass the same acceptance test as designed for the primary block, but which has a different design which it is hoped will not be prone to the same fault. This secondary block can be thought of as an 'alternative block' to the primary block.



The recovery scheme is not restricted to a single alternate block; a number of alternative blocks may be added, to be executed in turn if previous blocks have failed the acceptance test.

This recovery block scheme is described by the syntax given in fig 4.1. The common acceptance test is designated by the ENSURE statement and is placed at the beginning of the recovery block. Following the acceptance test is the primary block (BY) and the alternate blocks (ELSE BY).

```
ENSURE    <acceptance test>
BY        <primary block>
ELSE BY   <alternate block 1>
ELSE BY   <alternate block 2>
  -
  -
ELSE ERROR
```

Fig 4.1. Recovery Block Outline.

On entry to the recovery block the recovery point is established and the primary block entered. On completion of the primary block the acceptance test is executed: if the test does not raise an exeception the recovery block is exited. However, if an exception is raised the recovery point is restored, the next alternate block is executed and above procedure is repeated.

Recovery blocks can be nested so that one recovery block can form part of a primary block of an enclosing recovery block [4].

4.2.2 Concurrent Systems.

The recovery block scheme for error detection and recovery in single sequential process systems cannot be used directly for networks of communicating

sequential processes [4]. In concurrent systems consisting of a set of communicating processes, when one process raises an exception it is not sufficient to perform recovery actions on just that single process. The extent the recovery of that process impinges upon the other processes in the system must be considered. In communicating processes information flows between the processes and faults migrate. This influences the nature of the required recovery process.

For example, if a process has just sent information to a second process and an exception is raised, both processes should undergo recovery since the transmitted information could be in error. Similarly, if a process has received information from another process and then an exception is raised, it may require the information to be sent again (or another form of it); and thus both processes must be recovered.

A further problem in the design of error detection and recovery mechanisms for a system of communicating sequential processes is that the recovery points of the processes cannot be chosen arbitrarily. If recovery points of interacting processes are not properly coordinated, then an intolerably long sequence of rollback propagations can occur. This is termed the domino effect [4]. In the most extreme case

the roll back sequence would continue to the beginning of the program. This effect can be illustrated by considering a system of three processes as shown in figure 4.2. The three processes, P1,P2,P3, have autonomously established four recovery points. The dotted lines indicate process interactions. If Process 2 fails, it will be backed up to its fourth recovery point past an interaction with Process 1; this must therefore also be backed up to the recovery point immediately prior to this interaction. However, if Process 3 should fail all the processes will have to be backed up right to their starting points.

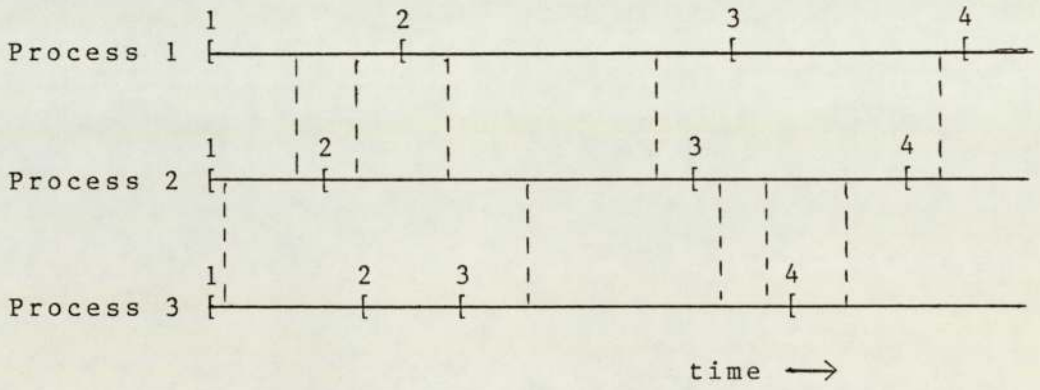


Fig 4.2 Example of the Domino Effect.

The domino effect can occur when two particular circumstances exist in combination [4]:

1) The recovery block structure of the various processes is uncoordinated, and take no account of

process interdependencies caused by their interactions.

2) The processes are symmetrical with respect to failure propagation - either member of any pair of interacting processes can cause the other to roll back.

4.3 Conversations.

An abstract construct termed a conversation was proposed [4] as an aid to the structuring of properly coordinated error detection and backward recovery actions of interacting processes. A conversation attempts to prevent the domino effect by dealing with circumstance 1 above. A conversation [37,25] is an extension of the recovery block technique, to two dimensions (i.e. time and processes). Like recovery blocks, conversations provide boundaries which serves to limit the damage caused to a system by errors.

4.3.1 Basic Structure of a Conversation.

The boundary of a conversation consists of a recovery line, a test line and two side walls. The boundary encloses the set of communicating (interacting) processes which are party to the conversation. The recovery line is the part of the boundary which

defines the start of the conversation. It consists of a coordinated set of states (recovery points) for the interacting processes. At the start of a conversation, the state of each entry process is stored for use if recovery is necessary. The entry to a conversation need not be a synchronous event.

The test line is a coordinated set of acceptance tests for the set of interacting processes. Each test line process is required to pass an acceptance test. A conversation is successful only if all test line processes pass their acceptance tests. Processes must exit from a conversation synchronously. If any acceptance test is failed, recovery is achieved by rolling back the conversation to the recovery line, restoring the process state to that on entry to the conversation, and executing the alternate blocks. Thus, processes in the conversation cooperate in error detection. The side walls of the conversation prohibit the passing of information to processes not involved in the conversation (prevent information smuggling). A representation of a conversation consisting of three processes is shown in fig 4.3.

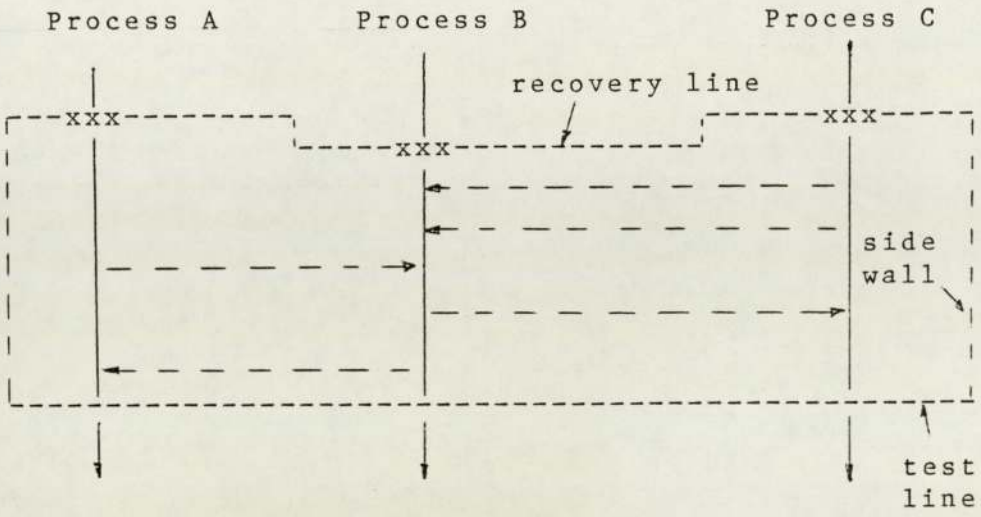


Fig 4.3. Conversation Scheme.

As with recovery blocks, conversations can be nested within other conversations, so as to provide additional possibilities for error detection and recovery. However, it is possible to envisage conversations which intersect and are not strictly nested . Such a structure is shown in fig 4.4.

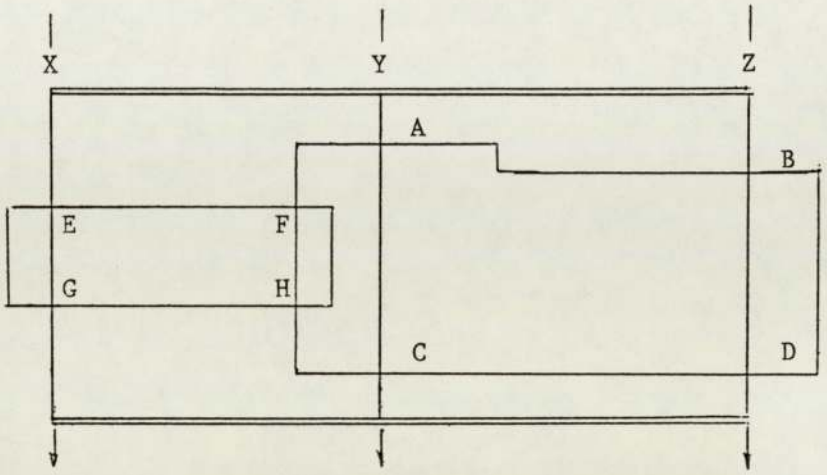


Fig 4.4. Example of Two Conversations which are not Strictly Nested.

Here if Y or Z fail their acceptance test at C and D it would not be possible to roll back X if it had passed its acceptance test at G. Thus conversations which intersect and are not strictly nested cannot be allowed.

4.3.2 Problems with Conversation Design.

The conversation scheme is a good fault tolerant mechanism for a system consisting of a set of communicating sequential processes, but a number of problems do exist in the design of such a scheme for a given system. These problems can be related with the boundary of the conversation.

Firstly, in order to define the boundary of a

conversation it is necessary to have some definition of system state. Without this definition of state it is obviously not possible to identify the entry and exit states (i.e. the boundary) of the conversation.

When defining the recovery states of the conversation it is crucial that the states be a consistent set. If the entry states do not form a consistent set the domino effect could occur. A consistent set can be precisely defined as follows [17]:

Consider a subset of processes P1 ... Pn, which establish recovery points at times t1 ... tn. Then a set is said to be consistent at some later time t iff

(i) in the period ti to tj processes Pi and Pj do not communicate

(ii) in the period ti to t, process Pi does not communicate with any process not in the subset.

Part (ii) of this definition of consistent set brings out a further problem of atomicity [16,12] when designing conversations: that is ensuring that all processes interacting within the space of the conversation must be party to the conversation. If this is not the case information smuggling will occur and the conversation mechanism will be useless.

4.4 A Possible Solution to Conversation Design.

In sequential single-process systems, the state of the process can be ascertained and saved, allowing the restoration of these states during fault recovery, as in the recovery block method.

The design problem is more complex in distributed systems which consist of a set of processes which operate automonously between synchronising inter-process communications [26]. This means that, between communications, the state of each process is independent of the state of the other processes and this leads to a considerable increase in the possible states of the system.

By using the Petri net model of a distributed system given in chapter 3, the states of the system are identified. Using the associated reachability tree it is possible to identify the entry and exit states of conversations [23,56] and to identify all processes involved in the conversation.

4.5 System State and Petri Nets.

The reachability tree of the system is used in a new design procedure to incorporate a fault tolerant mechanism in the form of conversations into a distributed system.

In chapter 3 the state of a Petri net was defined

by its marking at a given instance in time [31]. The marking, and thus, the state of the net is changed by the firing of enabled transitions. The net state will continue to change until the net terminates, deadlocks or falls into an infinite loop. The most convenient way of representing these state changes in the net is by the use of a reachability tree. All of the possible states of a Petri net can be obtained by traversing all branches of the reachability tree.

In chapter 3 a state-transition description of occam processes was derived. The state of software was mapped onto the state reachability tree of a Petri net model. By placing the initial markings onto the Petri net, the operation of the program can be simulated.

As each transition fires the marking of the Petri net will change. At each decision point, each branch must be considered seperately to obtain the complete reachability tree. The set of markings for the Petri net form a set of states for the program from which the Petri net was derived. Each node on the reachability tree defines the state of the whole system at that time.

It follows that, for any system modelled in occam, one can determine the "entry" state (or subset of states) and the set of possible reachable "exit"

states for each primitive process (or combination of processes).

It is also possible to distinguish between changes of state caused by local operations and changes of state caused by communication with other processes in the system by looking at the states changed by a given transition (local operations cause only one state change, communications cause more than one change). The complete reachability tree gives all possible combinations of system state which can exist with the given transitions.

4.6 Identification of Fault Tolerant Boundaries.

4.6.1 Construction of State Change Table.

A table which identifies which states change when a transition fires is constructed. The system dynamics are characterised by the evolution of the system states through a sequence of state transitions. This can be defined by a state-change table which lists the state changes for each transition in the reachability tree.

From the reachability tree it is possible to identify the present state and next-state for each transitions.

Present State $\mu = \{pa..pm\}$ 4.1

Next State $\mu' = \delta(\mu,t_j) = \{pc..pn\}$ 4.2

Equations 4.1 and 4.2 give the complete state of the system before and after the transition t_j has fired . If a state is unchanged after the firing of a transition, that state is independent of the particular transition. Since process state change is of interest, the states which do not change can be eliminated. This table (the state-change table) can be built by using the reachability tree, the present state of the system and the next-state function.

By taking the two relative complements [57] (relative complement of a and b [a-b] is the set of values which are in a but not in b) it is possible to determine only those states which change during the transition t_j.

$\mu - \mu' = \{pe..ps\} = I_j$ 4.3

$\mu' - \mu = \{pg..pt\} = E_j$ 4.4

The set I represents the subset of the initial states which are altered by the transition. The set E represents the subset of the final states which are created by the firing of the transition. For example, from table 4.1:

	I	E
t10	11,19	12,20

Thus, for transition t10 states 11 and 19 are changed and 12 and 20 created, also 11 $\longrightarrow$ 12 and 19 $\longrightarrow$ 20.

By determining the state changes for each transition in the reachability tree, a state-change table can be constructed, which lists the evolution of the system states as a function of the state transitions. As an example table 4.1 is the state-change table of the reachability tree in figure 4.7. The two lower-halves of the state-change table correspond to the two branches on the reachability tree.

Going from left to right on the same row in the table shows the change in system state caused by the transition on that particular row. If a state appears on the left hand side of the table, it is changed by that particular transition. If a state appears on the right hand side of the table, it has been changed to that state by the transition. The transition rows as they proceed down the table correspond to increasing time. For example, from table 4.1, state 1 is changed to state 2 by transition t1.

4.6.2 Identification of Communications.

When designing conversations inter-process communications are of interest, rather than intra-process communications. The state-change table may be reduced to a communication state-change table consisting of only communication transitions by removing all intra-process transitions.

Forming equivalent relationships between states created by intra-process actions (since these form local states between communication transitions) simplifies the state-change table. This table will be known as the communication state-change table. Inter-process communications can be identified by examining the process-identifier attributes of the relative complements in either equation 4.3 or 4.4.

If only one state has changed after a transition firing, then this is a local change and not a communication. A communication must cause at least two states to change, one in each of the communicating process. If two states have changed but both states belong to the same process, then again this is a local change.

However, if two states have changed and they belong to two different processes, then this is a state change caused by a communication. i.e. consider

a transition t_j ,

Let $p_1, p_2 \in I_j$ or $p_1, p_2 \in E_j$

where $p_1 \in$ PROCq and $p_2 \in$ PROCr

then the transition is an inter-process communication if $q \neq r$. When $q = r$ the transition can be classified as an intra-process action.

For example, the state-change table, table 4.1, can be reduced to the communication state-change table, table 4.2.

Transitions t10,t19,t25 can be combined to form a single row of the state-change table, because the associated ALT constructs are embedded within a replicator statement and the three ALT statements must all fire before the replicated ALT can terminate, i.e. the REP construct implies a replication of structure, which allows reduction. This can be seen by the fact that all three transitions share the same present state (11) and next state (12). Similarly, transitions t13,t21,t27 are condensed to a single row of the state-change table.

4.6.3 Identification of Conversations.

The underlying reason for a conversation is the interactions (communications) between the processes.

It is essential for proper placement of conversations to look at the interactions between the processes. If there are no interactions between processes, no conversation is required.

If guaranteed recoverability is to be provided for a set of processes which by interacting have become mutually dependent on each others progress, it must be arranged that processes cooperate in the provision of recovery points, as well as in the interchange of ordinary information [17]. It is shown here that a conversation can be constructed by generating systematically the entry and exit lines of the conversation such that no process interaction takes place through the side walls of the conversation.

The method partitions the reachability tree to form boundaries within the state space of the system. Any two transitions on the same branch of the reachability tree can be considered to form a boundary or partition within the communication state-change table. To determine a complete set of states which are initially marked at the chosen boundary and a set of states created at the end of the boundary two sets S and F, are formed from the union of all present and next states within the partition boundary, the relative complements of these sets are then taken to eliminate states created and destroyed within the boun-

dary.

$$S = \{I1 \cup I2 \cup .... In\} \qquad 4.5$$

$$F = \{E1 \cup E2 \cup .... En\} \qquad 4.6$$

Taking the relative complements of these sets:

$$S - F = \{p1,p2,...pn\} = K \qquad 4.7$$

$$F - S = \{pr,ps,...py\} = J \qquad 4.8$$

4.6.4 Entry and Exit States.

The set K represents the subset of the initial states which are altered by inter-process transitions and the set J represents the subset of final states which are created by inter-process transitions which fire within the boundary, table 4.3.

The two sets, J and K, can be considered to be the entry and exit states of a conversation. The partition or boundary of the communication state-change table then forms the boundary of the conversation. It follows that, by partitioning the state-change table at required transitions, the entry and exit points are defined by the sets K and J.

4.6.5 Processes in Conversation.

Each state in K and J can be identified with a process through the process-identifier attributes of the state;

$p_1 \in$ PROCq,

$p_2 \in$ PROCr.

Since the communications state-change table gives all the communications in a given state space, these are all the processes that are interacting during this state space and are thus the only processes required in the conversation.

4.7 Design of Conversations.

The design problem involves specifying a conversation boundary which will protect a particular part or function of the system. Such a specification will be expressed in terms of the functional processes, PROCcontrol, etc. The transitions and states associated with these functions can be identified through their process-identifier attributes. It is therefore possible to identify the corresponding transition and states in the communication state-change table. If the function is to be protected, then all the associated states must lie within the conversation. The entry state will coincide with the generation of the states, and the test line with the termination of these states. The set of states within the conversation can be found by identifying the entry and test lines which enclose the identified states (and a minimum set of the states) along the same branch of

the reachability tree. Since the sets K and J have as their members the entry and exit states for a given partition, only a single element of these sets is required to determine the complete set of entry and exit states for the particular partition.

4.7.1 Demonstrator Example.

Consider the example of a 3-axis robot arm controller [58]. The controller consists of an ´operator´ process which inputs the coordinates from the keyboard, checks for the reserved value, i.e. final position (0,0,0); passes the input values to process ´control´; and waits to receive further inputs. Process ´control´ accepts inputs from process ´operator´; calculates the new value for direction and distance for each motor: and passes the computed values to the ´motor´ processes. An occam solution to this problem is given in fig 4.5. There are five concurrent processes in this solution. There are three motor processes one for each axis which input values for distance and direction from process ´control´ and move the motors.

The robot arm control program of figure 4.5, can be translated into a Petri net graph using the transformations described in chapter 3. The complete Petri net graph for the robot program of figure 4.5 is shown in figure 4.6 and is partitioned into five

functional processes which correspond the actual processes in the program. The repetitive construct in each functional process gives rise to cyclic structures in the Petri net graph which serve to bound the graph. The closure of the cyclic loops is signified in figure 4.6 by the primes on the state identifiers (p1´, p9´, etc.).

The functional process boundaries (operator,control,motors) associated with the distributed system can be mapped onto occam and Petri net models of these systems. The transitions and states of the Petri net can therefore be associated with specific processes and assigned a process-identifier attribute; PROCi = {ti,pi}. Where ti = {ta..tg} and pi = {pa..ph}. Using the Petri net graph each state and transition can be assigned to a process-identifier attribute.

PROCoperator = {t1,t2,t3,t4,t5,t6,t7,t8,t16,t18,t22,
t24,t28,t30,p1,p2,p3,p4,p5,p6,p7,p8}
PROCcontrol = {t2,t3,t5,t8,t9,t10,t11,t12,t13,t14,
t19,t21,t25,t27,p9,p10,p11,p12,p13,p14,p15,p16,p17,
p18}

PROCmotor1 = {t10,t13,t15,t16,t17,t18,p19,p20,p21,
p22,p23,p24}

PROCmotor2 = {t19,t20,t21,t22,t23,t24,p25,p26,p27,
p28,p29,p30}

PROCmotor3 = {t25,t26,t27,t28,t29,t30,p31,p32,p33,
p34,p35,p36}

The reachability tree for the Petri net graph is shown in fig 4.7. There is only one decision point in this case (corresponding to the IF clause (t4)), therefore there are only two branches in this reachability tree : the main branch which is concerned with normal input variables and a second branch if the input is the final position of 0,0,0 in which case all processes are stopped.

Consider the robot control example and its associated communication state-change table (table 4.2). The main conversation protects the control process from the point at which new coordinates are input to the point at which all axial control motor processes have reported correct execution of the axial movement commands output by the control process. This specification is associated with the process control between the entry line state ´9´ and the test line state ´15´. Therefore, the communication state-change table can be partitioned on the main branch to enclose states 9 and

15 as shown in table 4.3. From this partition, the start and finish states, S and F, the entry line and the exit line states, K and J can be determined.

Although in this example, the conversation partitions have been applied to the communication state-change table, the partitions can also be applied to the state-change table. This may be necessary if a conversation is required to protect a particular function hidden, by the equivalence relationships, within the communication state-change table. The procedure is exactly the same as above.

4.8 Proof of Nesting.

Conversations can occur within other conversations, so as to provide additional possibilities of error detection and recovery. The partitioning of the state change table can be performed a number of times for a given problem as in tables 4.4 and 4.5. Every time a new partition is made, new conversation boundaries will be produced. Any or all of these conversations can be used on the system in question, as in figs 4.8 and 4.9.

If a system involves more than one conversation it may be the case that these conversations overlap each other. This overlapping will only occur if the

conversations are not disjoint with respect to both time and processes. If this overlapping does occur it is essential for the proper operation of the conversations that these conversations are strictly nested [25]; i.e. the inner conversation starts last and finishes first, and is completely enclosed by the outer conversation, as shown.

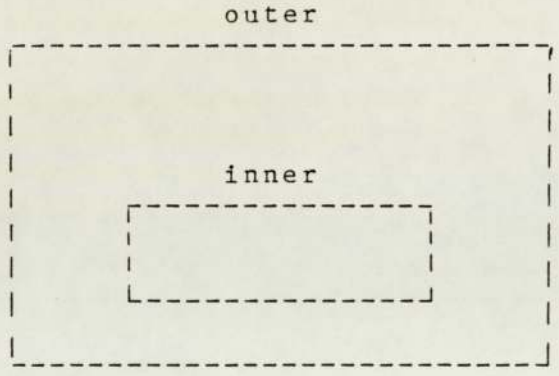


Fig 4.10. Example of Nested Conversation.

If the outer conversation does not fully enclose the inner conversation then the recovery mechanism may be useless. A process may leave the inner conversation and continue its execution. If the outer conversation acceptance test is then failed, the processes within it will be rolled back. However, since the process in the inner conversation has left the bounds of the outer conversation it is impossible to roll it back. An example of bad nesting is shown below.

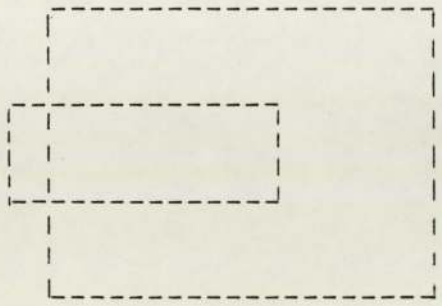


Fig 4.11. Example of Bad Nesting.

In all but trivial examples it is difficult to decide by inspection alone whether the conversations are properly nested or not. However, by using a state space analysis it is possible to determine if they are strictly nested or not.

Let the conversations be denoted by :

Ca, Cb etc

A conversation consists of two or more processes performing a number of actions. These actions may be local to a process or an interaction with another process in the conversation.

A conversation can thus be defined as a set of two sets: a set of transitions which take place during

it and a set of processes involved in it. Hence,

Ca = {ta,Pa}

Cb = {tb,Pb}

where ta = {t1,t2,...}

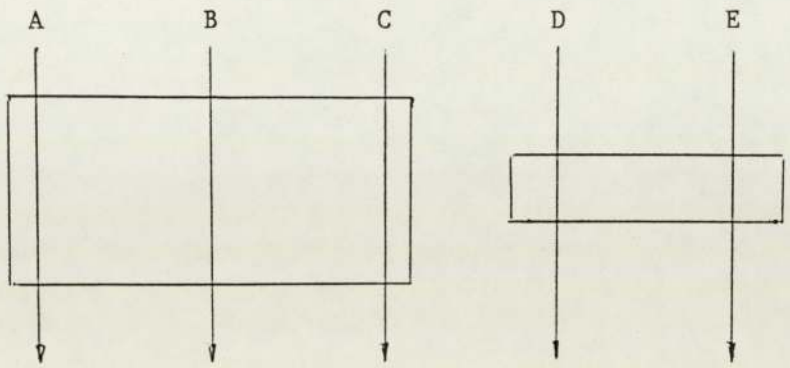
and Pa = {PA,PB,...}

Two conversations are said to be disjoint if either: they do not share any processes or if they do share processes one conversation has finished before the second has started. If the conversations are disjoint no nesting will be required. To decide if the conversations are disjoint, tests must be performed.

1) Taking the intersection of the sets of processes shows if the conversations are independent with respect to processes :

Ca ∩ Cb = {P}

If the intersection produces an empty set, {0}, then no communications take place between these conversations in this time and no nesting will be required.



2) It is possible that the above test fails, i.e. each conversation involves one or more common processes in their operation, but these may be disjoint in time.

If

$$\forall\, tx,ty : tx \in ta < ty \in tb$$

where ta is the set of transitions in conversation Ca and tb is the set of transitions in conversation Cb.

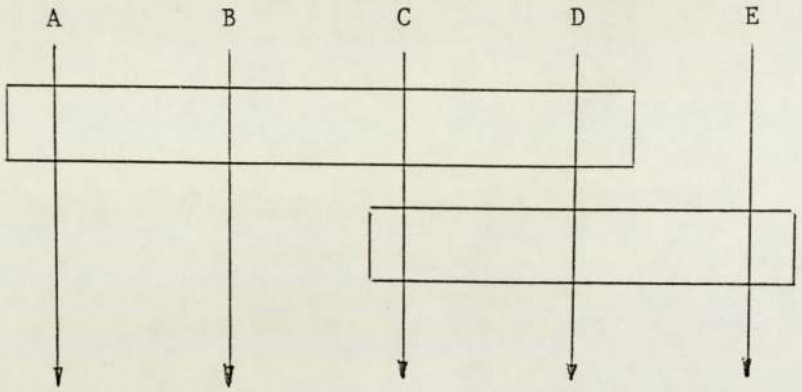
In this case conversation Ca finishes before conversation Cb starts.

if

$$\forall\, tx,ty : tx \in ta > ty \in tb$$

then, conversation Cb finishes before conversation Ca starts.

In either of the case the conversations are disjoint in time, i.e. one conversation is finished before the next starts and no nesting will be required.



If both of these tests prove negative then the conversations are not disjoint and further investigation must be made to determine if they are strictly nested or not.

For two conversations to be strictly nested, the outer conversation must fully enclose the inner conversation. Hence, the outer conversation must include all states present within the inner conversation.

The full set of states for a given conversation can be calculated using equations 4.5 - 4.8.

Equation 4.5 gives the set of all present states in the conversation.

Equation 4.7 gives the entry states of the

conversation, i.e. the states before the conversation was entered, thus :

$$(S - K) = L \qquad 4.9$$

gives all the present states enclosed by the conversation.

Similarly:

$$(F - J) = M \qquad 4.10$$

gives all the next states enclosed by the conversation.

Now, the complete set of states enclosed by the conversation can be determined by taking the union of equations 4.9 and 4.10

$$L \cup M = C \qquad 4.11$$

This is the set of all states which are changed during the conversation.

Let ΔCn be the change in states during conversation Cn.

$$\Delta Ca = \{p1, p2, ...\}$$

$\Delta Cb = \{pn, pm, ...\}$

For a conversation to operate successfully all processes involved in the conversation must be rolled back to the beginning of the conversation if an acceptance test is failed. To roll back after a failure has been detected all states in the inner conversation must be present in the outer conversation since the inner conversation will be re-entered on subsequent tries.

$\therefore \Delta Ca \cap \Delta Cb = \Delta Cb$ for successful nesting.

where Cb is the inner conversation.

Example of improperly nested conversations.

Looking at fig 4.12 :

$\Delta Ca = \{3, 10, 11, 12, 13, 14, 15\}$

$\Delta Cb = \{12, 13, 14, 20, 21, 26, 27, 32, 33\}$

$\Delta Ca \cap \Delta Cb = \{12, 13, 14\} \neq \Delta Cb$

$\therefore$ these conversations are not properly nested.

4.9 Discussion.

When errors occur a process should be rolled back to a previous state. It is critical that it is rolled back to a unique and well defined state on the entry boundary of the conversation. Once a conversation boundary has been identified it is important that all processes that are interacting within the state space of the conversation must be included in that conversation.

This chapter has addressed the problem of specifying and designing error detection and recovery mechanisms for a class of distributed systems. A method was described for the systematic identification of conversation boundaries.

The formalised definition of system state and reachability using Petri net techniques has been used. The properties of the state reachability tree were exploited in the development of a method for the design of proper conversations. The functional attributes of the system were used to identify conversations which would protect a particular part of a system (the conversation placement problem). The conversations designed using this method automatically enclose all processes which are party to the conversation.

The design method reduced the complexity of the problem by systematically reducing design

considerations to only those system states which are changed through interfunctional actions. These states provided the minimum set required for the design procedure and the identification of the recovery and test lines.

By using the same tools a technique was developed for testing if two or more conversations are properly nested.

```
Robot3.OCC
-- Occam program for 3-axis robot arm controller.
-- Declaration of inter-process channels.
CHAN request,return,motion[3],finished[3],stop[4],go[4]:
-- Declaration of process 'operator'.
PROC operator (CHAN send,receive) =
  VAR x,y,z,run :
  SEQ
    run := TRUE
    WHILE run
    SEQ
...   input x,y,z from keyboard.          --(t1)
--
      send ! x  --send to control process. (t2)
      send ! y
      send ! z
--
      receive ? ANY             --motors moved.(t3)
--
      IF
        (x=0)AND(y=0)AND(z=0) --check for finish. (t4)
          SEQ
            PAR i = [0 FOR 4]
              stop[i] ! ANY    --finish.(t5,t16,t22,t28)
            run := FALSE       -- (t6)
        TRUE                   -- (t7)
          PAR i = [0 FOR 4]
            go[i] ! ANY :  --continue.(t8,t18,t24,t30)
--
-- Declaration of process 'motor'.
PROC motor (CHAN motion,finished,stopi,goi) =
  VAR step,direction,going :
  SEQ
    going := TRUE
    WHILE going
      SEQ
        motion ? step  --get from control.(t10,t19,t25)
        motion ? direction
--
...     move motor          --(t15,t20,t26)
--
        finished ! ANY     --(t13,t21,t27)
--
        ALT
          stopi ? ANY       --finish.(t16,t22,t28)
            going := FALSE --(t17,t23,t29)
          goi ? ANY         --continue.(t18,t24,t30)
            SKIP :
```

Fig 4.5a. Occam Program for 3-Axis Robot Arm Controller.

```
-- Declaration of process 'control'.
PROC control (CHAN receive,send,stopi,goi) =
  VAR xold,yold,zold,xnew,ynew,znew,
  count,step[3],direction[3],going :
  SEQ
... initialise xold,yold,zold
    going := TRUE
    WHILE going
      SEQ
        receive ? xnew   --input from operator. (t2)
        receive ? ynew
        receive ? znew
--
...     calculate distance and direction      --(t9)
--
        PAR i = [0 FOR 3]
          SEQ
            motion[i] ! step[i]  --send to each motor.
            motion[i] ! direction[i] --(t10,t19,t25)
--
...     update xold,yold,zold        --(t11,t12)
--
        count := 0
        WHILE count <> 3
          ALT i = [0 FOR 3]
            finished[i] ? ANY  --check all motors moved.
              count := count + 1 --(t13,t21,t27)
        send ! ANY           --(t3)
--
        ALT
          stopi ? ANY        --finish. (t5)
            going := FALSE   --(t14)
          goi ? ANY          --continue. (t8)
            SKIP :
--
-- main program.
PAR
  PAR i = [0 FOR 3]
    motor(motion[i],finished[i],stop[i],go[i])
  control(request,return,stop[3],go[3])
  operator(request,return)
```

Fig 4.5b. Occam Program for 3 - Axis Robot Arm Controller.

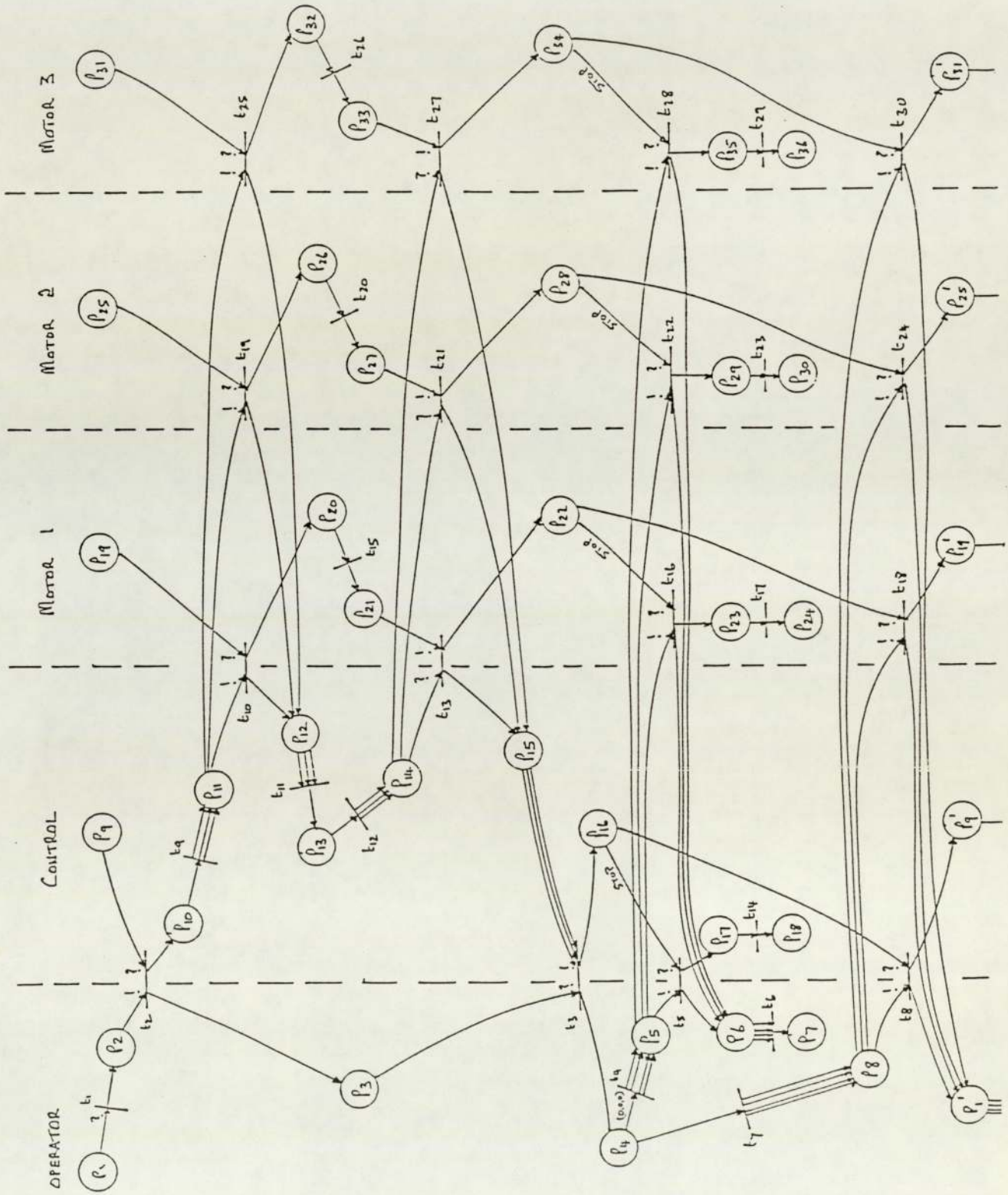


Figure 4.6. Petri Net of Occam Program in Fig 4.5.

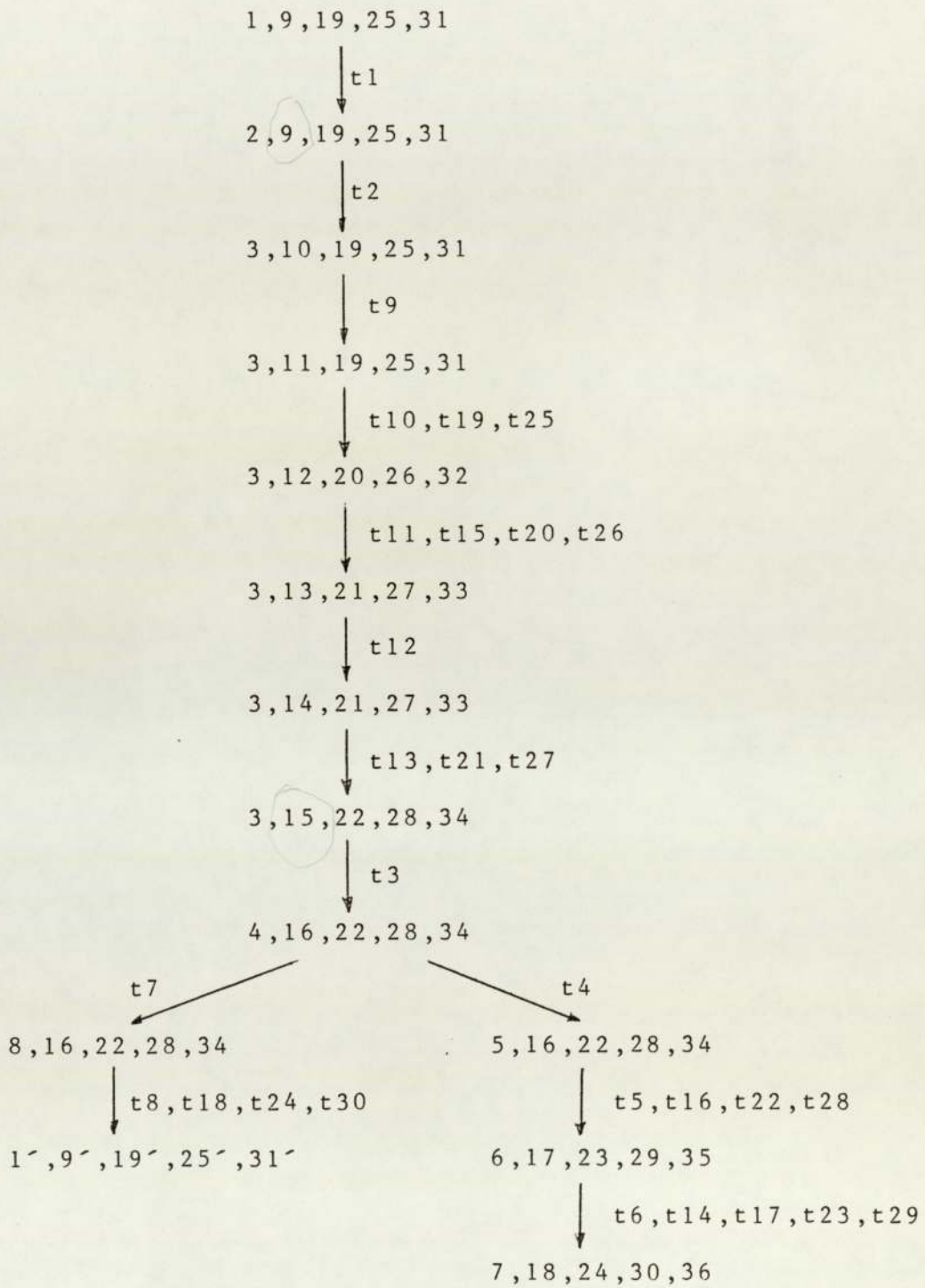


Fig 4.7. Reachability Tree of fig 4.6.

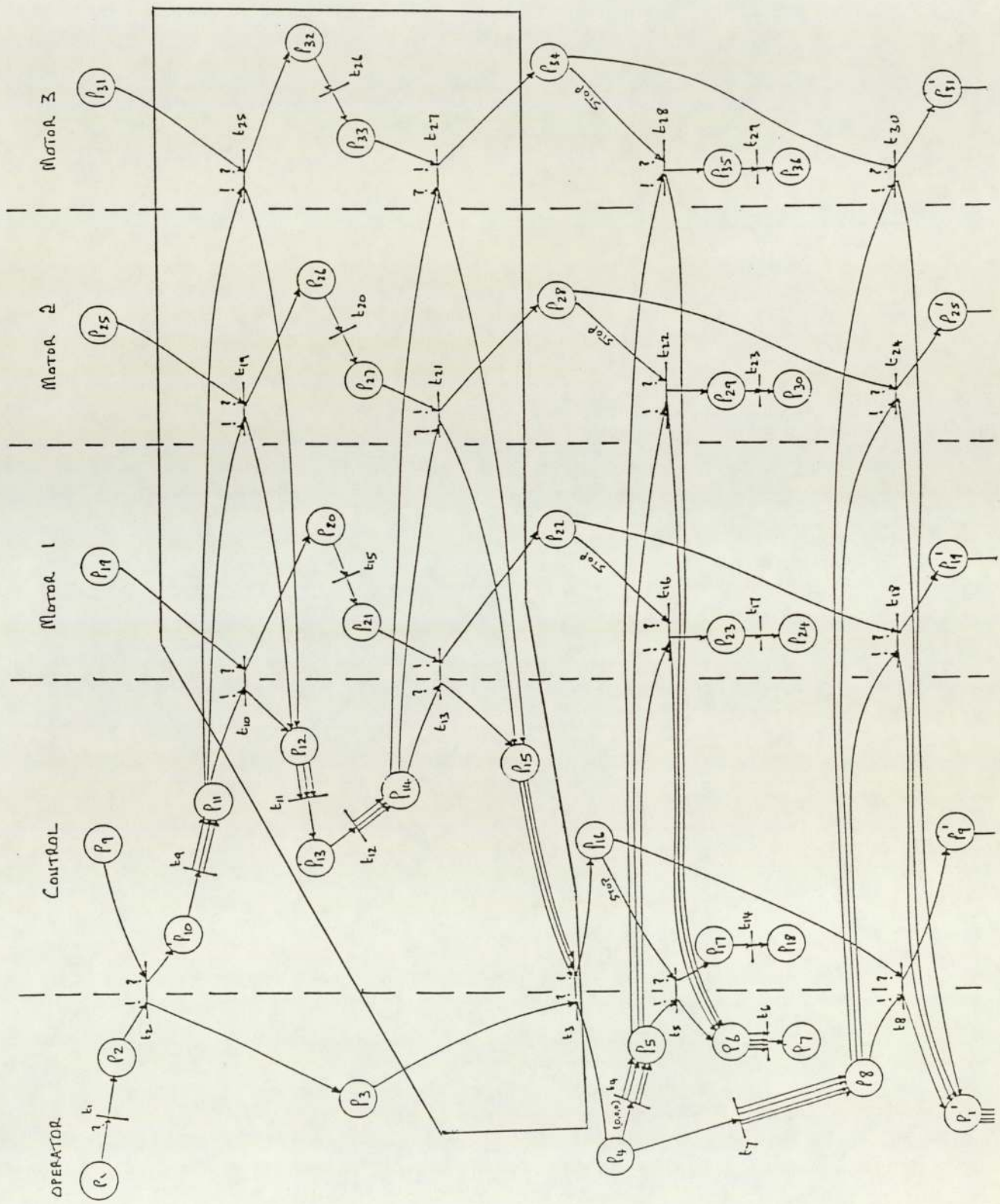


Figure 4.8. Partition from Table 4.4.

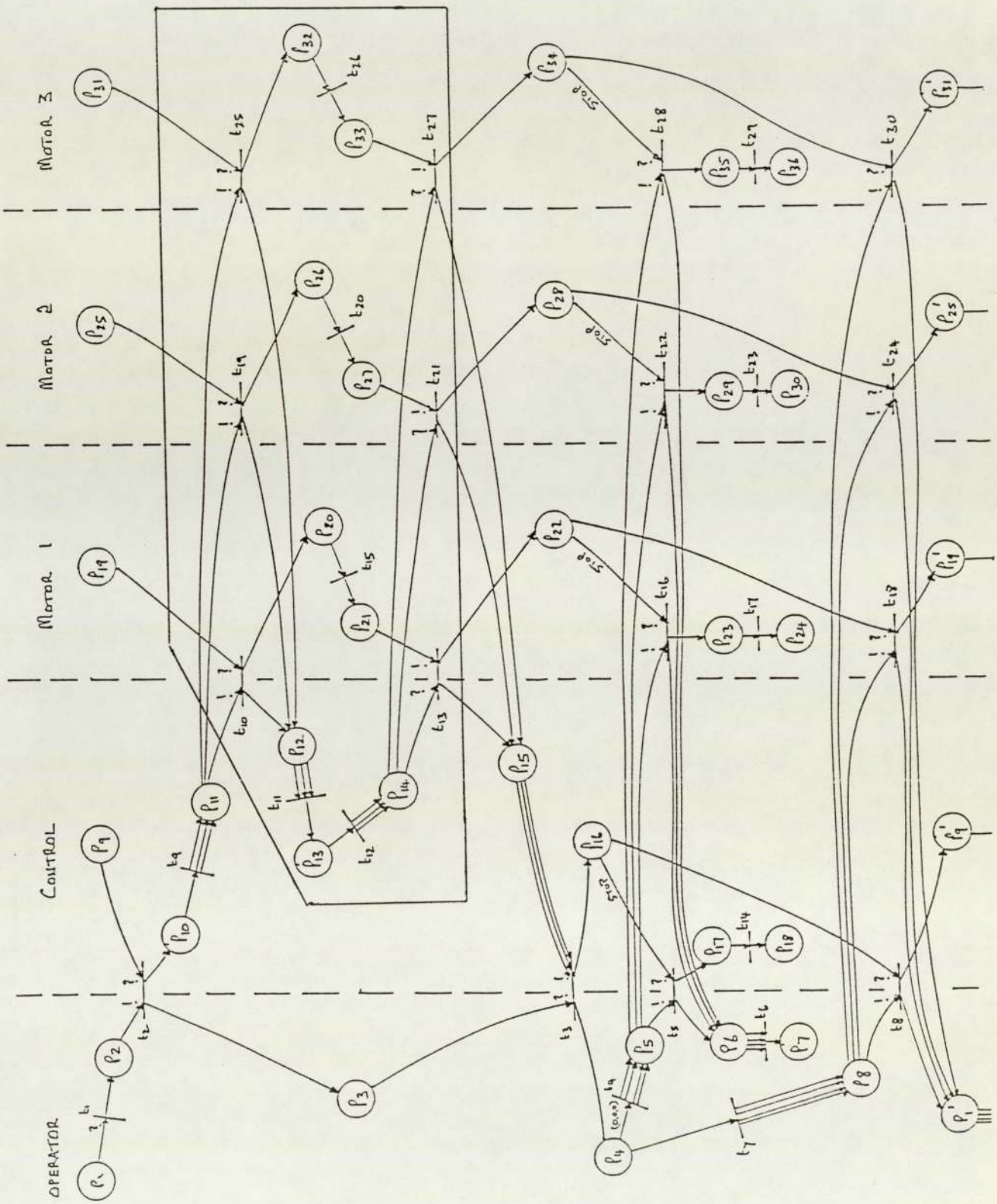


Figure 4.9. Partition from Table 4.5.

Transitions	I	E
t1	1	2
t2	2,9	3,10
t9	10	11
t10	11,19	12,20
t19	11,25	12,26
t25	11,31	12,32
t11	12	13
t15	20	21
t20	26	27
t26	32	33
t12	13	14
t13	14,21	15,22
t21	14,27	15,28
t27	14,33	15,35
t3	3,15	4,16
t7	4	8
t8	8,16	1´,9´
t18	8,22	1´,19´
t24	8,28	1´,25´
t30	8,34	1´,31´
t4	4	5
t5	5,16	6,17
t16	5,22	6,23
t22	5,28	6,29
t28	5,34	6,35
t6	6	7
t14	17	18
t17	23	24
t23	29	30
t29	35	36

Table 4.1. State Change Table of fig 4.7.

t2	2,9	3,11
t10,t19,t25	11,19,25,31	14,21,27,33
t13,t21,t27	14,21,27,33	15,22,28,34
t3	3,15	4,16
EITHER		
t8,t18,t24,t20	4,16,22,28,34	1´,9´,19´,25´,31´
OR		
t5,t16,t22,t28	4,16,22,28,34	7,18,24,30,36

Table 4.2. Communication State-Change Table of table 4.1.

t2	2,9	3,11
t10,t19,t25	11,19,25,31	14,21,27,33
t13,t21,t27	14,21,27,33	15,22,28,34

Boundary t2 - t27

Entry States K = {2,9,19,25,31}

Exit States J = {3,15,22,28,34}

Table 4.3. Partition of table 4.2 from t2 to t27.

t10,t19,t25	11,19,25,31	14,21,27,33
t13,t21,t27	14,21,27,33	15,22,28,34
t3	3,15	4,16

Boundary t10 - t3

Entry States K = {3,11,19,25,31}

Exit States J = {4,16,22,28,34}

Table 4.4. Partition of table 4.2 from t10 to t3.

t10,t19,t25	11,19,25,31	14,21,27,33
t13,t21,t27	14,21,27,33	15,22,28,34

Boundary t10 - t27

Entry States K = {11,19,25,31}

Exit States J = {15,22,28,34}

Table 4.5. Partition of table 4.2 from t10 to t27.

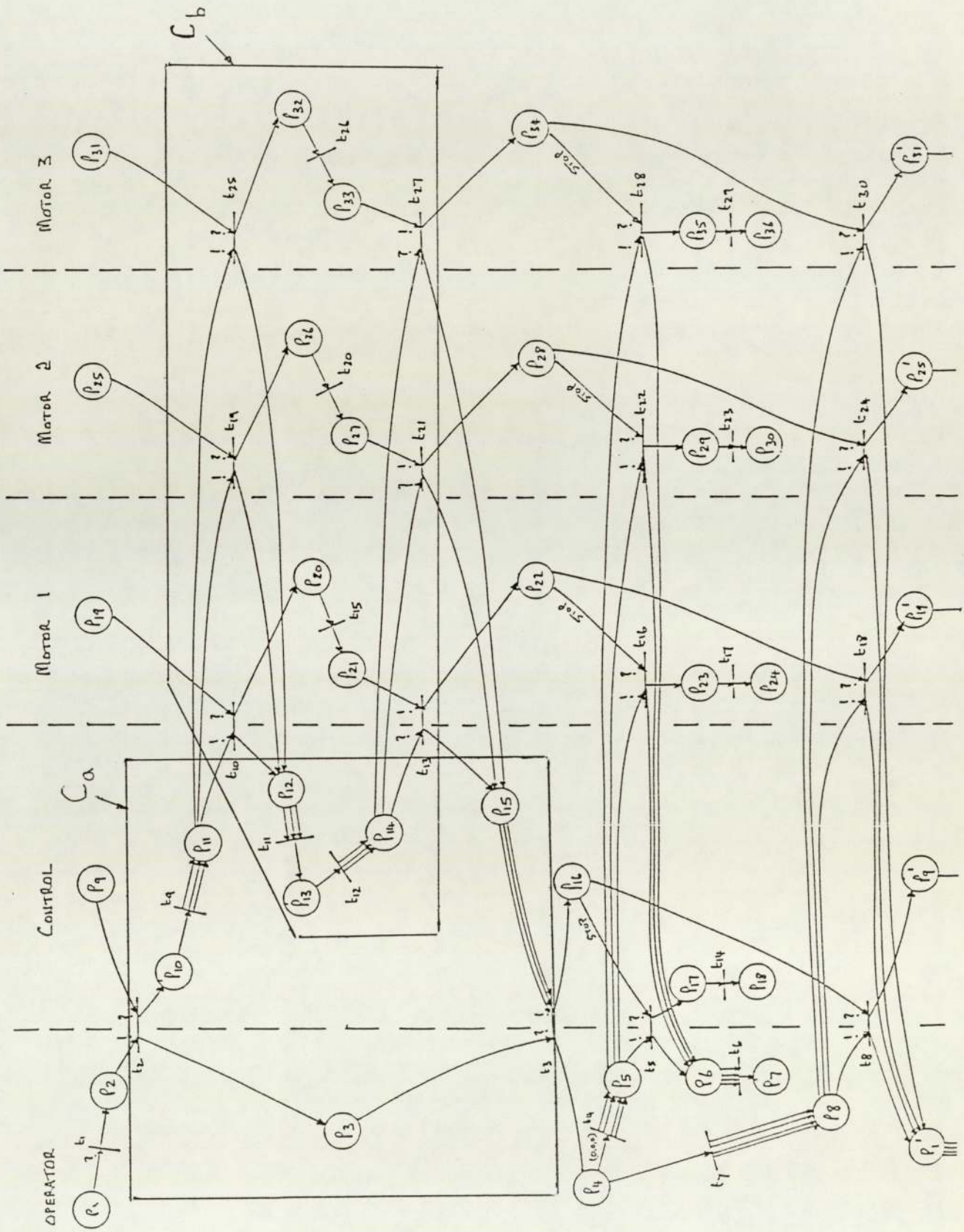


Figure 4.12. Petri Net with Bad Nesting.

Chapter 5.

Implementation of the Conversation Scheme.

5.1 Introduction.

In the previous chapter a design method was developed which enabled the designer to place conversation boundaries systematically. The method also gave properly nested conversation boundaries when two conversations were not disjoint. The aim of this chapter is to present an implementation for the conversation scheme using the concurrent programming language occam [36]. To be able to develop an implementation of the conversation scheme, once its boundaries have been identified, a number of subgoals must be met. Methods must be presented for the recovery roll back mechanism, if the conversation is failed, a structure for alternative blocks is required, the acceptance tests should be performed by the processes within the conversation and their results relayed to all other processes in the conversation and finally information smuggling should be prevented.

The method described here shows how these structures may be built without extensions to the language occam. An illustrative example is given to demon-

strate the use of these methods.

5.2 Features of Occam Support Environment.

In chapter 3 the primitives and constructs of the concurrent language occam were described. Some features of the support environment are now outlined, which prove useful in the implementation of conversations.

5.2.1 Initialisation and Termination of Processes.

Processes can be initialised and activated at the beginning of a program by the operating system. Processes may also be created during the operation of the program using process calls which may involve passing channels, variables and values as parameters to the process. The system terminates only when all processes within it have terminated. If the processes in the system contain loops or are part of races then termination messages passed along channels may be required to ensure proper termination of all processes in the system.

5.2.2 Folds.

The occam program support environment [22,59] provides a folding editor. The folds can be used to accommodate a hierarchical set of program elabora-

tions. The folds do not alter the execution of the program, however, they clarify the structure of the program and are used in the implementation presented here to highlight the structure of the programs and the structure of the conversations used for error detection and recovery. Each fold, denoted by "..." and a fold name, can contain program statements and/or other folds. An opened fold is denoted by "{{{ }}}" and the fold name.

5.3 An Implementation of the Conversation Scheme.

A conversation may be implemented using either centralised control with a test line coordinator process or distributed control with a distributed coordination mechanism. In the following both centralised and distributed mechanisms are developed and implemented using the occam programming language.

5.3.1 Features of the Conversation Scheme.

The structure of a conversation is described in chapter 4. In summary:

- A recovery line is defined at the entry to the conversation, which processes may not cross during roll back.

- A test line is defined at the exit from the conversation, which is an acceptability criterion for the

processes leaving the conversation.

- No information should be passed to or from processes outside the conversation. Thus, two side walls are postulated to prevent information smuggling.

- Processes can enter a conversation asynchronously.

- Processes must leave a conversation synchronously.

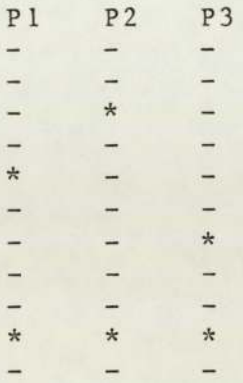
- Processes cooperate in error detection.

5.3.2 Design and Implementation of a Centralised Conversation Mechanism.

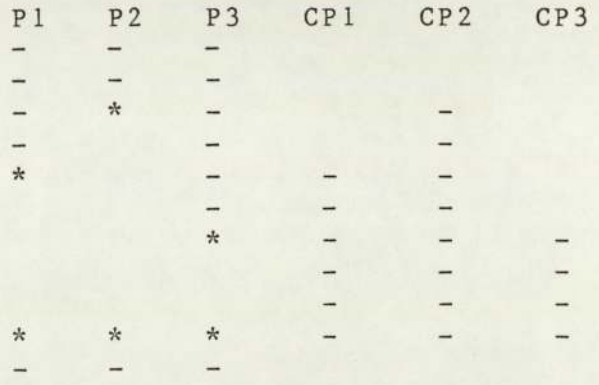
Consider 3 processes (P1,P2,P3) which interact with each other by message passing, as shown below. Where process flow is denoted by -.

```
P1   P2   P3
-    -    -
-    -    -
-    -    -
-    -    -
-    -    -
-    -    -
-    -    -
-    -    -
```

A conversation is required to protect a critical part of the system. To place the conversation in the correct position across the processes the techniques developed in chapter 4 are used. The boundary is shown below by *.



The parts of processes included in the conversation can be enclosed within a fold associated with the conversation. This clarifies the program structure. To make the conversation structure easier to identify, and to increase checks on information smuggling, the parts of the processes involved in the conversation can be put in "conversation processes" (CP1,CP2,CP3 below) and initiated by the main processes themselves. All these "conversation processes" can then be folded away in a conversation fold.



In the centralised case a conversation consists

of its constituent processes and a conversation control process which acts as a test line coordinator for the conversation. When a conversation is started, a nominated member of the set of entry processes initialises the conversation coordinator. The coordinator exists for the duration of the conversation. When all acceptance tests have been passed all processes involved in the conversation are terminated. Below the test line process is shown by TLP and will be included into the conversation fold with the "conversation processes".

```
P1   P2   P3   CP1   CP2   CP3   TLP
-    -    -
-    -    -
-    *    -          -           -
-         -          -           -
*         -     -    -           -
          -     -    -           -
          *     -    -     -     -
                -    -     -     -
                -    -     -     -
*    *    *     -    -     -     -
-    -    -
```

Since the processes in the conversation are grouped together in folds easy identification of side wall violations is achieved.

5.3.3 Implementation Example.

Consider, for example, the three axis positional robot discussed in chapter 4. This comprises operator, control and axial positioning processes (fig 5.1)

which are elaborated in figure 5.2(a,b,c). A centralised coordinator is used for each conversation which provides fault tolerant control of a distributed system.

```
{{{ process 3 axis robot

...         operator process
...         control process
...         3-axial position processes
}}}
```

Fig 5.1. 3 Axis Control Robot.

The conversations are determined by partitioning the reachability tree of the system. The main conversation is required to protect the process which calculates new values of axial coordinates and the motor process implements these coordinates. This conversation involves all main processes: operator, control, motors and its boundary is shown clearly in the listing for each process (fig 5.2a,b,c). Alternatively, folding can be used to clarify the structure of the conversation. For example, the program can be refolded such that all constituent parts of the conversation (including the test line process) lie within a fold, as in figure 5.3.

The structure of the control part of the conversation including the control test line and recovery procedure is shown in fig 5.4. The primary and alternative blocks are again folded away to reveal the structure of the recovery mechanism. On entry, the recovery variables are saved, for use if the process is rolled back. The control variable, "enter", controls the forward path and initially points to the primary block, which is then executed. On exit from the primary block the local acceptance test is executed on the control part of the conversation. The result of this acceptance test (pass or fail) is reported to the test line process. The test line process (fig 5.5) collects the results of all local acceptance tests and determines whether the conversation has succeeded. If all tests are passed the test line process notifies all exit processes in the conversation and the conversation is terminated. If one or more of the acceptance tests fails the test line process notifies all processes that recovery roll back is to be executed. The processes then roll back and restore the saved entry states and the block entry variable is updated, the next block is then executed. The above procedure is then repeated.

The test line process (fig 5.5) uses the ALT construct to receive notification of the results of local acceptance tests. The acceptance process does not

therefore assume any particular order for the termination of the constituent processes; nor does it impose any timing constraints on the systems performance.

Thus, the figures 5.2a - 5.5 show a conversation design using occam which includes a recovery roll back mechanism, a structure to choose the alternatives, an acceptance test procedure and easy identification of information smuggling. In the next section an implementation of nested conversations is considered.

5.3.4 Nested Conversations.

Properly nested conversations may be designed using the methodology outlined in chapter 4, by identifying partitions which are totally enclosed within other partitions; the inner nested conversation is initiated by processes which are themselves within a conversation. Again these processes can be folded away into a conversation fold. An example of this type is shown (fig 5.6) in an elaboration of the primary block of the control process (fig 5.2b). Here the conversation protects the procedure by which operator passes new input values to process control, and the controller calculates new coordinates. This conversation involves the processes: operator, control and test line.

5.3.5 Global Acceptance Tests.

The processes in a system of communicating processes will inherently not have the complete information relating to the whole system. To provide a higher degree of reliability for the system, information from a number of processes may be required to enable an acceptance test to be performed. In such circumstances the local acceptance tests are not sufficient to protect the system. An additional global acceptance test will be required to ensure certain conditions between processes do not exist. In the implementation presented here an acceptance test process is used to coordinate the processes leaving the conversation, there by providing a global acceptance test. When each process sends its local acceptance test results to the acceptance test process they also send the data required for the global acceptance test. This data are collected by the acceptance test process which performs the global acceptance test and reports the result to the other processes in the conversation. This extension to the previous example are shown in figs 5.7, 5.8 and 5.9.

5.4 Implementation of a Distributed Acceptance Test Process.

The centralised control of a conversation, such as those described above, can be removed by introduc-

ing a mechanism in which each process transmits the result of its local acceptance test to all other exit processes. Each exit process will contain an individual version of the test line and will autonomously decide whether to continue or recover. An example of this method using the two processes in conversation b, is shown in figs 5.10 and 5.11.

5.4.1 Disadvantages of this Method.

Communications between processes as well as design faults are a possible source of errors. It is therefore advantageous to reduce the number of communication channels required in the system.

Consider the number of channels required in a distributed system:

assuming there are N processes in the conversation, Process(i) will require (N-1) channels to communicate to all other processes, therefore the total number of channels required is N(N-1).

For the centralised case Process(i) will require 2 channels, one to the test line coordinator and one to receive the return message, again assuming N processes the total number of channels required is 2N.

Thus, the number of channels required for more

than three processes is much larger than that for the centralised conversation structure. For example, a conversation involving 5 processes requires 10 channels in the centralised case and 20 channels in the decentralised case.

In addition the processes have to communicate with each other in a specific order. This becomes necessary because in occam outputs can not be used as guards to processes. This adds further complexity to the implementation of this type of conversation, and means the structure of the processes will change with the number of processes involved in the conversation. The complexity is increased with an increase in the number of processes in the conversation.

A further example to emphasise this point is shown in figs 5.12a,b,c, where the program following the acceptance test is illustrated for a conversation involving three processes. It can be seen from this that the processes are asymmetric and careful design of the distributed test line process is necessary if deadlocks are to be avoided.

Finally, a global acceptance test with this type of implementation is not easily performed. One of the processes within the conversation would have to be nominated to perform such a test and all results would have to be sent to it.

Thus, a centralised test line process is to be preferred in many applications. The relative simplicity of the centralised test line process implementation is due to the constructs available within the language occam.

5.5 Advantages Gained using occam.

A channel in occam can only belong to two processes, one of which is the input and the other the output of the channel [60,61]. This limitation is very useful when constructing conversations. For example, one of the major rules of conversations is that no information smuggling should occur. That is, only processes involved in the conversation should interact with each other. Within a conversation smuggling will not occur if no channel belongs to a process outside the conversation.

The alternative construct (ALT) makes it possible for a process to accept inputs from any number of channels in a nondeterministic way. This simplifies construction of the test line process. In addition the operation of accepting inputs from all other processes is simplified by the use of the replicator on the alternative construct (ALT i = [0 FOR n]). The resulting part of the program consists of only two statements and its length is independent of the number

of processes involved in the conversation.

Similarly the acceptance process transmits to all other processes in the conversation by using the replicator with the parallel construct (PAR i = [0 FOR m]).

Although the technique of folds does not affect the actual construction of the program it does help clarify the structure of the program. By using the folds in appropriate places it is possible to reveal the important features of the conversation. For example, the processes involved in each conversation can be seen clearly, as in fig 5.3. Similarly, the structure for roll back and entry into alternatives are highlighted in fig 5.4. The acceptance test for each process involved in the conversation can be seen easily also in fig 5.4.

Finally, the structure of the recovery mechanism is independent of the primary and alternate blocks. The recovery mechanism proposed here considers the alternatives as blocks which are entered and then produce results. How this is achieved is of no consequence to the recovery mechanism. Each block within a conversation can therefore be considered independently, as in the primary block of the control process, fig 5.6. These blocks may be changed without affecting the structure of the conversation mechanism

in any way. An extra alternative can be added easily to the structure by the addition to the IF clause (i.e. enter = 4) and the new alternative block.

By using the centralised test line process the constituent processes in the conversation do not require knowledge of the number of other processes within the conversation. This makes the mechanism for the acceptance test independent of the number of processes.

The mechanism proposed thus forms a generalised design and implementation of the conversation mechanism.

5.6 Discussion.

It has been shown that by using the features of the concurrent language occam a conversation design can be implemented easily and is independent of the application. Further, the characteristics of the communications in occam, in which communication channels belong to two processes only, was exploited to solve the problem of information smuggling by ensuring that no channel in the conversation belongs to a process outside the conversation. The design procedure was aided by the use of folding editors which facilitate functional elaboration. These were used to highlight

the structure and extent of the conversation; in effect folds were used as a design notation for the boundary of conversations.

```
{{{ process operator
PROC operator (CHAN send,receive,dis0) =
  VAR x,y,z,run :
  SEQ
    run := TRUE
    WHILE run
--
      SEQ
...     get inputs x,y,z
--
-- Initiate conversation a.
--**** Recovery Line ******************
        PAR
          acceptest
          operatortocontrol(send,x,y,z)
--***** Test Line **********************
-- Conversation a ends.
--
        receive ? ANY
-- receive move confirmed
        out(dis0,"arm moved to position*n*c")
        IF
          (x=0)AND(y=0)AND(z=0)
-- then finish
            SEQ
              out(dis0,"finished arm at 0,0,0*n*c")
              PAR i = [0 FOR 5]
                stop[i] ! ANY
              run := FALSE
          TRUE
-- else continue
            PAR i = [0 FOR 4]
              go[i] ! ANY :
}}}
```

Fig 5.2a. Process operator

```
{{{ process control
PROC control (CHAN receive,send,stopi,goi,disl) =
  VAR going :
  SEQ
    xold := 0
    yold := 0
    zold := 0
    going := TRUE
    WHILE going
      SEQ
--
-- Initiate conversation a.
--***** Recovery Line ********************
        controltomotor(receive,send,disl)
--***** Test Line ************************
-- Conversation a ends.
--
        send ! ANY
        ALT
          stopi ? ANY
            going := FALSE
          goi ? ANY
            SKIP :
}}}
```

Fig 5.2b. Process Control.

```
{{{ process motor
PROC motor(CHAN motion,finished,stopi,goi,
disi,alrighti,oki,VALUE n) =
  VAR step,direction,going :
  SEQ
    going := TRUE
    WHILE going
      SEQ
--
-- Initiate conversation a.
--***** Recovery Line ******************
        movemotors(motion,finished,disi,
                   alrighti,oki,n)
--***** Test Line **********************
-- Conversation a ends.
--
        ALT
          stopi ? ANY
            going := FALSE
          goi ? ANY
            SKIP :
}}}
```

Fig 5.2c. Process Motor.

```
{{{  conversation a.
--***** Conversation Fold Start *******
...  conversation a, (operator part).
--
...  conversation a, (control part).
--
...  conversation a, (motor part).
--
...  test line (coordinator).
--***** Conversation Fold Finish ******
}}}


... remainder of operator process

... remainder of control process

... remainder of motor process
```

Fig 5.3. Axis Control Robot with Refolding to show Conversation.

```
{{{ conversation a, from control.
PROC controltomotor (CHAN receive,send,dis1) =
  DEF limit = 10, pass = 1, fail = 0 :
  VAR going,enter :
  SEQ
    going := TRUE
    enter := 0
... save recovery variables
    WHILE going
      SEQ
-- recovery line.
... restore variables if roll back occurs
--
-- Recovery loop.
--
        enter := enter + 1
        IF
          (enter = 1)
...primary block
          (enter = 2)
...2nd block
          (enter = 3)
...3rd block
--
-- Acceptance test for control part of
-- conversation a.
        IF
          ((xold < limit)AND(yold < limit)AND
           (zold < limit))
--
-- Inform test line process result of
-- acceptance test.
--
            allright[0] ! pass
          TRUE
            allright[0] ! fail
--
-- Get result of combined acceptance test from
-- test line process.
--
        ok[0] ? test
        IF
          (test = pass)
            going := FALSE
          TRUE
            SKIP
    SKIP :
}}}
```

Fig 5.4. Acceptance Test and Recovery Structure.

```
{{{ test line.
PROC acceptest =
  DEF pass = 1, fail = 0, total = 5 :
  VAR flag,num,going,test :
  SEQ
    going := TRUE
    WHILE going
      SEQ
        flag := pass
        num := 0
--
-- Get results from all processes in conversation.
--
        WHILE num < total
          ALT i = [0 FOR total]
            allright[i] ? test
              IF
                (test = pass)
                  num := num + 1
                TRUE
                  SEQ
                    flag := fail
                    num := num + 1
--
-- Communicate result of all accptance tests
-- to processes in conversation.
--
        IF
          (flag = fail)
            PAR i = [0 FOR total]
              ok[i] ! fail
          TRUE
            SEQ
              PAR i = [0 FOR total]
                ok[i] ! pass
              going := FALSE
    SKIP :
}}}
```

Fig 5.5. Test Line Process.

```
-- Initiate conversation a.
--
{{{  primary block of control process
            DEF total = 3 :
            VAR count,step[3],direction[3] :
            SEQ
--
-- Initiate conversation b.
--***** Recovery Line ************************
              PAR
                acceptestb
                calstepdir(receive,send,dis1)
--***** Test Line ****************************
-- conversation b ends.
--
              PAR i = [0 FOR 3]
                SEQ
                  motion[i] ! step[i]
                  motion[i] ! direction[i]
--
              xold := xnew
              yold := ynew
              zold := znew
--
              count := 0
              WHILE count <> total
                ALT i = [0 FOR 3]
                  finished[i] ? ANY
                    count := count + 1
-- conversation a ends.
}}}
```

Fig 5.6. Primary Block of Control Process.

```
{{{ conversation b, from control.
PROC calstepdir (CHAN receive,send,dis1) =
  DEF limit = 10, pass = 1, fail = 0 :
  VAR going,enter :
  SEQ
    going := TRUE
    enter := 0
... save recovery variables
    WHILE going
      SEQ
-- recovery line.
... restore variables on roll back
--
-- Recovery loop.
--
        enter := enter + 1
        IF
          (enter = 1)
... primary block
          (enter = 2)
... 2nd block
          (enter = 3)
... 3rd block
--
-- Acceptance test for control part of
-- conversation b.
        IF
          ((xnew < limit)AND(ynew < limit)AND
           (znew < limit))
--
-- Inform test line process result of acceptance
-- test, and send data for global acceptance test.
--
            SEQ
              allright[5] ! pass
              allright[5] ! xnew
          TRUE
            allright[5] ! fail
--
-- Get result of combined acceptance test from
-- test line process.
--
        ok[5] ? test
        IF
          (test = pass)
            going := FALSE
          TRUE
            SKIP
    SKIP :
}}}
```

Fig 5.7. Acceptance Test and Recovery Structure of control part of Conversation b with global acceptance test.

```
{{{ conversation b, from operator.
PROC optocontrol (CHAN receive,send,disl) =
  DEF limit = 10, pass = 1, fail = 0 :
  VAR going,enter :
  SEQ
    going := TRUE
    enter := 0
... save recovery variables
    WHILE going
      SEQ
-- recovery line.
... restore variables on roll back
--
-- Recovery loop.
--
        enter := enter + 1
        IF
          (enter = 1)
... primary block
          (enter = 2)
... 2nd block
          (enter = 3)
...,3rd block
--
-- Acceptance test for operator part of
-- conversation b.
        IF
          ((x < limit)AND(y < limit)AND(z < limit))
--
-- Inform test line process result of acceptance
-- test, and send data for global acceptance test.
--
            SEQ
              allright[6] ! pass
              allright[6] ! x
          TRUE
            allright[6] ! fail
--
-- Get result of combined acceptance test from
-- test line process.
--
        ok[6] ? test
        IF
          (test = pass)
            going := FALSE
          TRUE
            SKIP
    SKIP :
}}}
```

Fig 5.8. Acceptance Test and Recovery Structure of operator part of Conversation b with global acceptance test.

```
{{{ test line for conversation b.
PROC acceptest =
  DEF pass = 1, fail = 0, total = 2 :
  VAR flag,num,going,test,data[7] :
  SEQ
    going := TRUE
    WHILE going
      SEQ
        flag := pass
        num := 0
--
-- Get results from all processes in conversation.
--
        WHILE num < total
          ALT i = [5 FOR total]
            allright[i] ? test
              IF
                (test = pass)
                  SEQ
--
-- Input data for global acceptance test.
--
                    allright[i] ? data[i]
                    num := num + 1
                TRUE
                  SEQ
                    flag := fail
                    num := num + 1
--
-- Communicate result of all accptance tests to
-- processes in conversation.
--
        IF
          (flag = fail)
            PAR i = [5 FOR total]
              ok[i] ! fail
          TRUE
--
-- Global acceptance test.
--
            IF
              (data[5] <> data[6])
                PAR i = [5 FOR total]
                  ok[i] ! fail
              TRUE
                SEQ
                  PAR i = [5 FOR total]
                    ok[i] ! pass
                  going := FALSE
    SKIP :
}}}
```

Fig 5.9. Test Line Process for Conversation b with global acceptance test added.

```
{{{ conversation b, from control.
PROC calstepdir (CHAN receive,send,dis1) =
  DEF limit = 10, pass = 1, fail = 0 :
  VAR going,enter,test :
  SEQ
    going := TRUE
    enter := 0
... save recovery variables
    WHILE going
      SEQ
-- recovery line.
... restore variables on roll back
--
-- Recovery loop.
--
        enter := enter + 1
        IF
          (enter = 1)
... primary block
          (enter = 2)
... 2nd block
          (enter = 3)
... 3rd block
--
-- Acceptance test for control part of
-- conversation b.
        IF
          ((xnew < limit)AND(ynew < limit)AND
           (znew < limit))
--
-- Inform other process result of
-- acceptance test and receive theirs.
--
            SEQ
              allright[5] ! pass
              allright[6] ? test
              IF
                (test = pass)
                  going := FALSE
                TRUE
                  SKIP
          TRUE
            SEQ
              allright[5] ! fail
              allright[6] ? test
    SKIP :
}}}
```

Fig 5.10. Acceptance Test and Recovery Structure of control part of Conversation b with decentralised acceptance test.

```
{{{ conversation b, from operator.
PROC optocontrol (CHAN receive,send,disl) =
  DEF limit = 10, pass = 1, fail = 0 :
  VAR going,enter,test :
  SEQ
    going := TRUE
    enter := 0
... save recovery variables
    WHILE going
      SEQ
-- recovery line.
... restore variables on roll back
--
-- Recovery loop.
--
        enter := enter + 1
        IF
          (enter = 1)
... primary block
          (enter = 2)
... 2nd block
          (enter = 3)
... 3rd block
--
-- Acceptance test for operator part of
-- conversation b.
        IF
          ((x < limit)AND(y < limit)AND(z < limit))
--
-- Receive results from other process
-- and relay own results.
--
            SEQ
              allright[5] ? pass
              allright[6] ! test
              IF
                (test = pass)
                  going := FALSE
                TRUE
                  SKIP
          TRUE
            SEQ
              allright[5] ? fail
              allright[6] ! test
    SKIP :
}}}
```

Fig 5.11. Acceptance Test and Recovery Structure of operator part of Conversation b with decentralised acceptance test.

```
PROC A
   recovery loop
IF
  (acceptance test)
    SEQ                          -- passed local test
      PAR i = [0 FOR 2]
        allright[i] ! pass  --inform other processes
      WHILE num < total
        ALT i = [2 FOR 2]
          allright[i] ? test  -- get their results
            IF
              (test = pass)   -- pass
                num := num + 1
              TRUE
                SEQ                    -- fail
                  fulltest := fail
                  num := num + 1
      IF
        (fulltest = pass)  -- if all pass
          going := FALSE   -- exit conversation
        TRUE
          SKIP
  TRUE
    SEQ                          -- failed local test
      PAR i = [0 FOR 2]
        allright[i] ! fail -- inform other processes
      WHILE num < total
        ALT i = [2 FOR 2]
          allright[i] ? test  -- get their results
            num := num + 1
```

Fig 5.12a. Acceptance Test Structure with three processes in the Conversation with decentralised acceptance test.

```
PROC B
   recovery loop
IF
  (acceptance test)
    SEQ                          -- passed local test
      allright[0] ? test         -- result from A
      IF
        (test = pass)
          SKIP
        TRUE
          fulltest := fail
      PAR
        allright[2] ! pass       -- infrom local to A
        allright[4] ! pass       -- infrom local to C
        allright[5] ? test       -- get result from C
      IF
        (test = pass)
          SKIP
        TRUE
          fulltest := fail
      IF
        (fulltest = pass)
          going := FALSE
        TRUE
          SKIP
  TRUE                           -- failed local test
    PAR
      allright[0] ? test         -- from A
      allright[2] ! fail         -- to A
      allright[4] ! fail         -- to C
      allright[5] ? test         -- from C
```

Fig 5.12b. Acceptance Test Structure with three processes in the Conversation with decentralised acceptance test.

```
PROC C
   recovery loop
IF
  (acceptance test)
    SEQ                          -- passed local test
      allright[1] ? test         -- local from A
      IF
        (test = pass)
          SKIP
        TRUE
          fulltest := fail
      PAR
        allright[3] ! pass       -- inform local to A
        allright[4] ? test       -- local from B
      IF
        (test = pass)
          SKIP
        TRUE
          fulltest := fail
      allright[5] ! pass         -- infrom local to B
      IF
        (fulltest = pass)
          going := FALSE
        TRUE
          SKIP
  TRUE                           -- failed local test
    PAR
      allright[1] ? test         -- from A
      allright[3] ! fail         -- to A
      allright[4] ? test         -- from B
      allright[5] ! fail         -- to B
```

Fig 5.12c. Acceptance Test Structure with three processes in the Conversation with decentralised acceptance test.

Chapter 6.

Reliable Communications.

6.1. Introduction.

In a system consisting of a set of communicating processes, if one of the communicating processes fails to reach a communication point the other process involved in the communication could become deadlocked [19]. If a process has become deadlocked it is not possible for the process to recover from faults by using the conversation scheme alone; since process flow has stopped. A deadlocked process may cause further processes to become deadlocked by its failure to reach future communication points. In a system using conversations for fault tolerance a process deadlocking could cause a complete failure of the system. The main objective of this chapter is to ensure that processes do not become deadlocked.

To achieve this objective each of the communication primitives, for message passing, are introduced. Message types for such systems are considered and two main types are identified. The requirements for reliable communications for each of these message types are given. Each of the communication primitives are

compared with the requirements for reliable communications.

The analysis of the communication primitives show them to be deficient in a fault tolerant situation. The message types are analysed using a Petri net state-transition model. State reduction of the state-transition model can be made. It is then possible to identify a boundary for a mechanism which will timeout the process. It is argued that by placement of a timeout mechanism around this boundary the process will not deadlock if a communication has failed.

Much research has been done on the implementation of distributed systems [22,46,62-68]. This has mainly concentrated on the development of languages for such systems. Less research has been done on the design of such systems. Generally in these designs [69-73] the issues of communication and synchronisation by message passing are not addressed explicitly. Communication mechanisms, such as the monitor [30], rely upon shared memory for passing messages. This limits the choice of hardware for the system. A message passing mechanism for interprocessor communications provides a more flexible choice of hardware for the final implementation, since this can use any hardware configuration from a completely distributed system to a shared memory system. Message passing also has the advantage

of mirroring better a physically distributed system.

6.2 Communication Primitives.

In this chapter a task which initiates a communication will be referred to as a source task. The task to which a communication is sent is called an object task. A channel is the medium used for such communications. Channels are unidirectional.

It is assumed that all interprocess communication is accomplished by message passing. The different communication primitives used in message passing can be classified into three types [63]:

a) Synchronous.

b) Asynchronous.

c) Remote Procedure Call.

6.2.1 Synchronous.

In a synchronous system the process that reaches the communication point first must wait for the other process before it can continue. The source task requires an acknowledgement from the object task before it can proceed. The source task may then issue an initiating message. On receipt of this message, the object task will issue an acknowledgement message.

When the source has received the acknowledgement both processes may continue autonomously. Process synchronisation is thus enforced through communication. This type of communication can be found in C.S.P., occam, L [46,22,62].

Synchronous communication has two primitives, shown below:

```
comms ! <message>;      --source task sends
                        --<message> to object task.

comms ? <message>;      --object task receives
                        --<message> from source task.
```

Since these are primitives of the language the acknowledge is implied and not shown in the notation.

6.2.2 Asynchronous.

Asynchronous communication systems do not require the object task to acknowledge the receipt of a message from the source. In this type of communication primitive the source task issues an initiating message and may then continue its operation. Since synchronisation is not enforced, some form of buffering is required to hold the initiating message should the object task not be ready to receive the message. Similarly, the object task may be required to wait if the message has not been initiated by the source. This type of communication primitive can be found in

CLU, PLITS [63,64].

The asynchronous communication also has two primitives; these are of the form:

```
SEND  <message>  TO  <object.task>;
-- source task sends <message> to object task

RECEIVE  <message>  FROM  <source.task>;
-- object task receives <message> from source task.
```

6.2.3 Remote Procedure Call.

The third type of communication primitive is the remote procedure call (remote invocation). The object task performs a specific function for the source task in a way similar to that of a subroutine. The object task is not initiated until it has received the source message, it completes local computation before issuing a reply. The source task waits for the object task to report completion of the procedure before continuing. Communication primitives of this type can be found in ADA, DP, CONIC [65-70].

The remote procedure call primitives have two parameters as shown below:

```
SOURCE TASK

object.request <in-parameters,out-parameters>;

OBJECT (PROCEDURE) TASK

accept.request <in-parameters>
            do <service request
                send out-parameters>;
```

6.2.4 Message Transactions.

Messages are used in distributed systems for the collection or distribution of data and to promulgate decisions or actions. Three main functional classifications or transfer categories are identified for messages in distributed computer systems [74] : command, status and alarm.

COMMANDS are messages which cause a change of state or action in the object task. These messages generally require a reply from the object task to signify completion of the action.

STATUS messages are sent by a source task to a number of object tasks and are used to convey source status. These message may be initiated by the source either periodically or when the state of the source task changes. Status messages may also be generated by an object task in response to a request from a source task.

ALARMS are messages initiated by the source to inform the other tasks that the controlled process is malfunctioning or is in an unsafe state.

These transfer categories can be divided into two message types [74]:

Command Message Type: Those requiring a reply to

their initial message: command and requested status messages.

Notify Message Type: Those requiring no reply: periodic status, event status and alarm messages.

6.3 Requirements for Reliable Communications.

As discussed in the introduction of this chapter tasks should not be allowed to deadlock. In this section the requirements this imposes for both source and object task are considered. Each of the message passing primitives are used to model the message types described above and related to the requirements for reliable communications.

The source task may fail to complete a request-reply transaction for a number of reasons: loss of the request message by the communications system; non-acceptance of the request by the object task due to processor failure or object task failure; failure of the object task while it is servicing the request; loss of the reply message by the communications system.

Any of the above failures will leave the source task suspended indefinitely, i.e. deadlock. Consequently, the requirement for a task initiating a request-reply transition is that it should have the

capability of detecting failure of a transaction and abandoning or aborting it.

While a task is awaiting completion of a request-reply transaction it cannot respond to other inputs from other tasks. A task must be able to set a maximum limit on the time it allocates to the request-reply transaction. Thus, a periodic interrupt from an independent time source is required to provide a time-reference event, independent of the software processes. The timing performance of an application process can be monitored by a real-time time-lapse counter. Both processes will be attempted concurrently; the first task to complete succeeds and the other attempt is withdrawn. This 'timeout' technique is known as a 'watchdog' [42] mechanism. The timeout can also be used to meet the requirement to detect and abandon failed transactions.

The integrity requirement for the object task is primarily that it should not be capable of suspending itself indefinitely waiting to receive a transaction which could fail to arrive. This requirement is partially met by the behavioural requirement for the object task to be able to wait on more than one potential transaction.

However, where a task is committed to performing an action regularly it must be able to limit the time

it is in the waiting state. For the object task this is the time it is prepared to wait to accept transactions. The object task must be able to timeout from awaiting transactions. This timeout covers the integrity requirement that the task should not be indefinitely suspended.

6.4 Implementation of Message Types.

The two groups of message types (command and notify) for distributed systems identified above can be modelled using the communication primitives of section 6.2. In this section analysis shows that when these two groups of messages are implemented directly with these primitives they contain ambiguities and deficiencies [24].

6.4.1 Command Message Types.

The command message can be modelled using the primitives described in section 6.2:

Command: Asynchronous Implementation.

```
SOURCE TASK                    OBJECT TASK

SEND <request>
TO <object>;      --------->   RECEIVE <request> FROM
                               <source>;
                               (service request)
RECEIVE <reply> <---------     SEND <reply> TO <source>;
FROM <object>;
```

The communication subsystem of a distributed

computer system may be subject to failure, either from external cause or because of faults in the object or source tasks. It is common to incorporate a timeout mechanism in such a system which will initiate the appropriate recovery mechanism. The timing requirements of such a system are only partially satisfied if the RECEIVE primitive is put in a SELECTIVE or ALTERNATIVE construct with a timeout.

```
SOURCE TASK

SEND <request> TO <object>;
SELECT
       TIMEOUT <period>
     OR
       RECEIVE <request> FROM <object>
```

When the select statement is executed both the communication and timeout tasks are attempted. Whichever task is completed first is defined as executed and the other attempt is withdrawn. However, such a model is still ambiguous because of the absence of any logical or notational paring between the two halves of the transaction. This could lead to a situation in which the object task sends a message to a timed-out source task.

Command: Synchronous Implementation.

```
SOURCE TASK                        OBJECT TASK

initiate ! <request>; -------> initiate ? <request>;
                                  (service request)
answer ?  <reply>;  <--------  answer ! <reply>;
```

In this case the logical clarity is good. However, the timing requirements are not easily satisfied. For example, if a timeout was placed on the receive primitive to break the wait-for-synchronisation then the source task could be suspended if the timeout was activated:

```
SOURCE TASK                OBJECT TASK

                                clock := NOW;
                                ALT
                                  WAIT NOW AFTER <clock +
                                                  timeout>
initiate ! <request>   ---->   initiate ? <request>
```

A time-out can not be placed on the send primitive because outputs are not allowed as guards in synchronous primitives.

The remote procedure call primitive of section 6.2 simulates the command message directly:

Command: Remote Procedure Call Implementation.

```
SOURCE TASK

  objecttask.request <in-parameter,
                      out-parameter>;

OBJECT (PROCEDURE) TASK

  accept request <in-parameters> do
           <service request - send out-parameters>;
```

Remote procedure calls of this type are used for communications in the Ada language [65]. The Ada implementation uses a timeout on the acceptance of the message by the object task and has the form :-

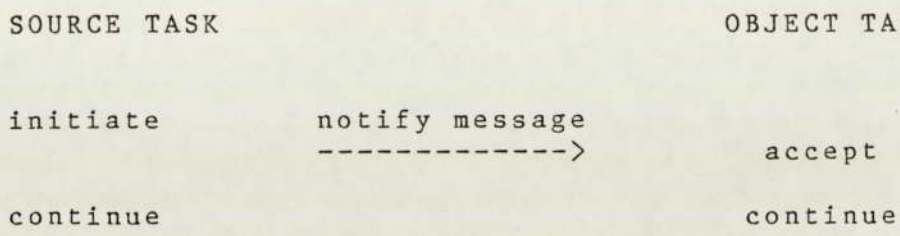
```
SOURCE TASK

SELECT
       TIMEOUT <period>
     OR
       object.request <in-parameters,out-parameters>;
```

This system is again ambiguous, for if the reply message is lost then the source task is suspended indefinitely. A solution to the problem of source task suspension was included in CONIC [70] in which the timeout is placed on the completion of the whole transaction.

6.4.2 Notify Message Types.

Notify transactions consist of one message conveying unsolicited information from source to object task. The information is unsolicited in the sense that it is not in reply to a specific request from the object task. Notify transactions can be between one source task and one or more object tasks. The notation below describes a notify transaction.



As with the command message type, some form of breakout mechanism is required around this transaction if process deadlock is to be prevented.

To achieve a reliable system the communication models outlined above must be assessed as formal notations for the expression of concurrency, the pairing of communication primitives, and the ability to incorporate timeout mechanisms in a satisfactory manner. None of the models outlined satisfy all of these objectives. The first two objectives are satisfied by the synchronous communication system which is developed in C.S.P. [50] and embedded as primitives in the derived language occam [22]. The formal semantics of C.S.P. allow certain proofs of program correctness and include correctness preserving transformations. However, the scope of the language does not include timeout mechanisms. The third objective is met by CONIC.

The two separate lines of development [22,70] have led to the language CONIC which satisfy the

transaction and timing requirments and to the synchronous communication systems such as that in occam which provides a formal notation and mechanism for the implementation of strictly synchronous systems. The following section explores the advantages to be realised by developing a formal construct which includes select or 'breakout' to provide the flexibility required to satisfy timing constraints for a synchronous communication system [20].

6.5 Modelling with Petri Nets.

Using the modelling techniques of chapter 3 it is possible to show where the boundary for a timeout mechanism must be placed if the communications are to be protected.

Starting with the lowest transaction level, it is possible to model a synchronous communications primitive (pair) using Petri net techniques (fig 6.1). Since the actions within the dotted line are atomic, i.e. to the outside world it looks like one action, this communication primitive may be re-drawn by making the atomic actions a single transition (fig 6.2). This procedure is similar to that of hierarchical Petri nets [75]. From fig 6.2 it can now be seen that the only position to put a breakout mechanism is around the transition, as in fig 6.3.

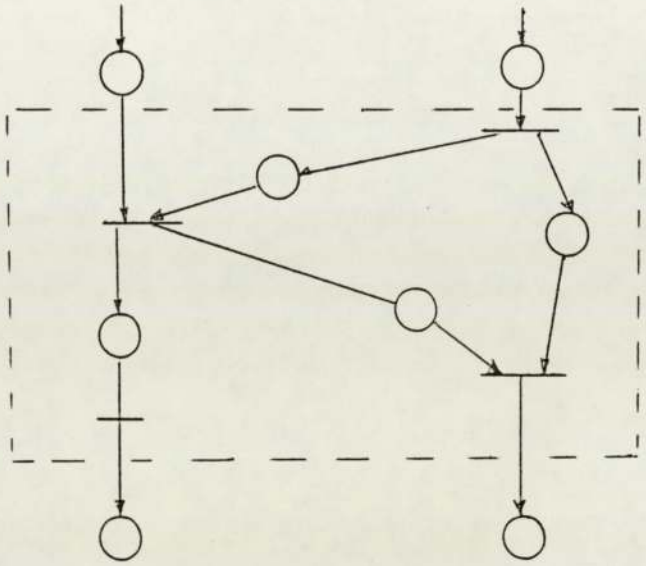


Figure 6.1. Model of Synchronous Communication Primitive.

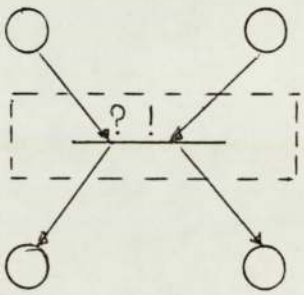


Figure 6.2. Model of Reduced Synchronous Primitives.

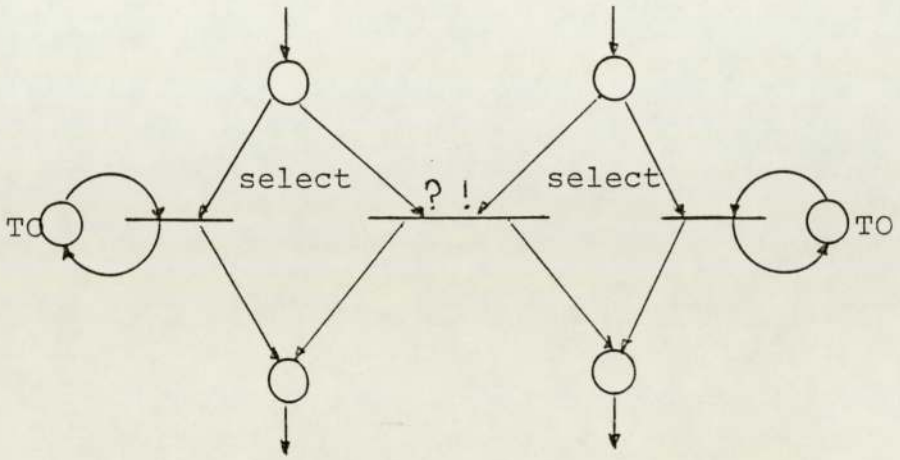


Figure 6.3. Model of Synchronous Communications with Breakout.

Considering the command type transaction: this can also be modelled using a Petri net graph (fig 6.4). Again looking at the graph the natural place to simplify the graph is to reduce the map from t4 to t6 into a single transition which can be thought of as atomic (fig 6.5). The double line indicates that the entry transition is an output for the source task and input for the object task, and the other way around for the exit transition.

Thus, when considering where to place a timeout mechanism the only place is around this transition (fig 6.6).

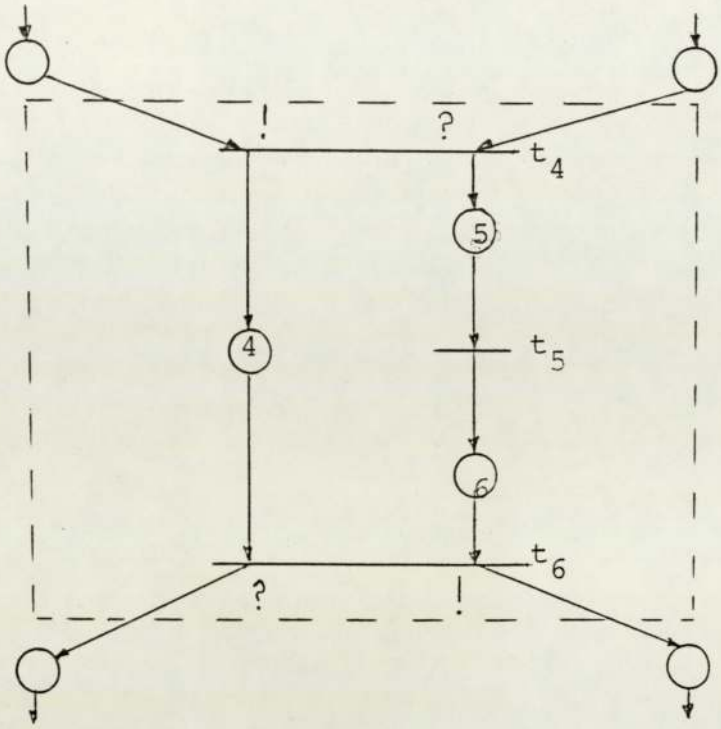


Figure 6.4. Model of Command Type Transaction.

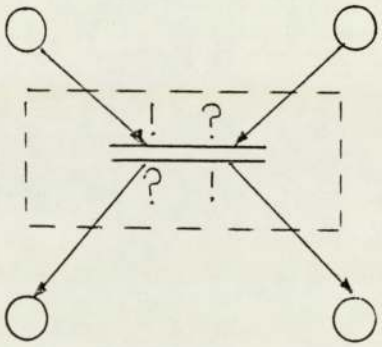


Figure 6.5. Model of Reduced Command Transaction.

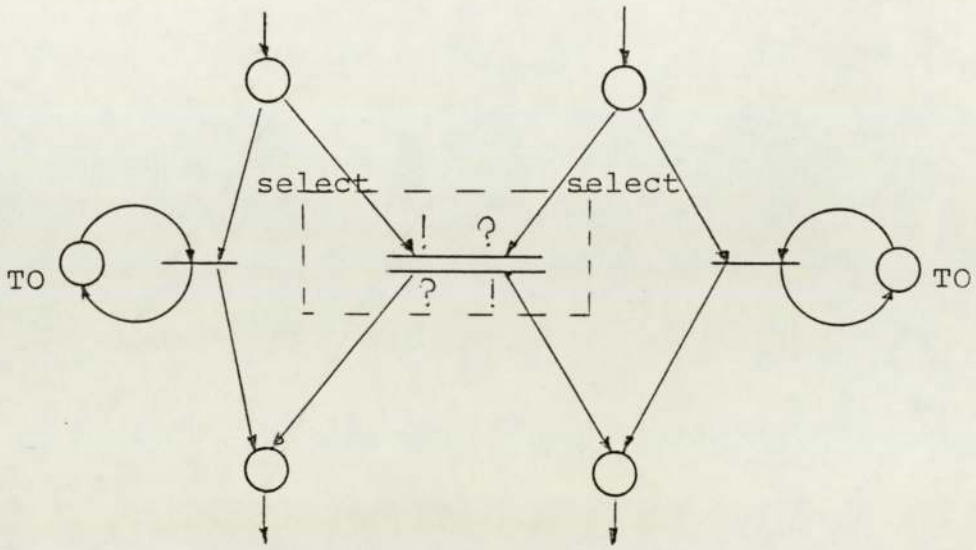


Figure 6.6. Model of Command Transaction with Breakout.

The command group message type can now be implemented by synchronous communications. However, the communication primitives have been enhanced by the inclusion of a select mechanism which provides an ability to break-out of the formal notation in the event of system behaviour conflicting with timing requirments. (This is necessary because the ALT construct does not allow the inclusion of the initiating output).

```
SOURCE TASK.

SELECT
  SEQ
    initiate ! <request>; --send request to
                          --object task.
    answer ? <reply>;     --wait reply from object
                          --task.
OR
  TIMEOUT (period);       --time-out if above not
                          --completed by period.


OBJECT TASK.

SELECT
  SEQ
    initiate ? <request>;   --receive from source
                            --request.
    SEQ
      (service request);    --service request.
      answer ! <reply>;     --send reply to source.
OR
  TIMEOUT (period);         --time-out if above not
                            --completed by period.
```

Under normal circumstances the synchronous communication will be successful. However, there are two cases when the timeout will be chosen:

i) Timeout occurring when either the source or the object task is waiting for the other task to reach the communication point. In this event the communication will not have been initiated.

ii) Timeout occurring after the initiation of the object task but before completion of the whole transaction.

In each case, both processes will have to be rolled-back to the start of the communication for recovery.

A timeout for each communication primitive is not required, since a timeout on the whole transaction will prevent either task from being suspended indefinitely.

An implementation of the notify type message with synchronous communication primitives will not meet the reliability requirements stated in section 6.3. Both the source and object tasks could be suspended indefinitely if its partner in the communication fails to reach the communication point. However, by using the SELECT construct introduced for the command message type, the integrity requirements can be met, ensuring neither task suspends itself indefinitely.

```
  SOURCE TASK

SELECT
  initiate ! <message>
OR
  TIMEOUT (period)

 OBJECT TASK

SELECT
  initiate ? <message>
OR
  TIMEOUT (period)
```

As with the command message type, under normal circumstances the synchronous communication will be successful. If however one of the tasks fails to reach the communication point the other task will be broken out of the transaction by the timeout process and will not be left suspended indefinitely.

6.6 Discussion.

This chapter examined some of the problems of designing software for distributed computer systems. It takes the view that in such systems communication and synchronisation should be addressed explicitly. It has shown that existing languages are ambiguous because they lack a logical paring between the two halves of the transaction, or have other deficiencies when applied to the design of distributed systems that are used in time-critical, real-time situations. However, the timing requirments of such systems are satisfied by CONIC while the formal notation of languages such as occam satisfy the concurrency and communication needs of strictly synchronous systems.

A notation has been proposed for the design of command and notify type messages in distributed computer systems. This notation provides strictly synchronous communication primitives in normal operation but includes break-out facilities necessary to satisfy timing constraints. This communication transaction built from occam-type primitives adds real-time communication to the notation.

Comparing the transactions proposed in this chapter with the implementation described in chapter 5, it can be seen that both the command and notify group message types are present. The notify type of

message is the most common where data are sent from one process to one or more processes and all processes continue. The command type of message can be identified in the acceptance test structure. Here each of the exit processes sends its acceptance test results to the test line process and then waits for a result back from this process.

Chapter 7.

Conclusions.

7.1 Conclusions.

In many situations the complexity of a system may prohibit a software design which is totally fault free. The use of design methodologies and correctness proofs will help towards the production of error free software but are impractical at the moment for large systems. In such cases the best that can be aimed for is a system which has a high probability of success in the presence of errors. Since the process of testing non-trivial software cannot be achieved fully provision for hidden design errors must be made.

This thesis has been concerned with the design of fault tolerant software for distributed computer systems consisting of sets of communicating sequential processes, which can only communicate with each other by message passing. A backward error recovery mechanism was used. The type of errors the technique described can cope with are unpredictable design errors which can be detected by logical acceptance tests.

A state-transition model for concurrent systems

was developed. The concurrent system described in the form of an occam specification is mapped onto a Petri net model. It is shown that all of the sequential and concurrent constructs present in occam can be modelled successfully. Given the Petri net model of the system it is shown that the state of the system can be represented by the Petri net and state evolution mapped onto a state-transition diagram of the net, i.e. the reachability tree.

Since the occam language is sufficient to represent concurrent system, the state-transition model developed should also be sufficient for a concurrent system. The Petri net model not only has the ability to model the control flow of concurrent systems, and map the state evolution of such a system, it is also possible to provide information about the concurrent system such as the possibility of process deadlock.

A systematic design of conversations was proposed in chapter 4. The design method uses the state-transition model developed in chapter 3. The design makes uses of the system state evolution. A state evolution (state-change) table was constructed from the state-transition model. This table enables the designer to identify boundaries to protect specific functional parts of the system. It provides a list of

the minimum set of states required for entry into and exit from a conversation. Using process attributes the method also allows the designer to identify a minimum set of processes which must be party to a conversation.

By partitioning the state-change table a number of times, nested conversation boundaries can be obtained. Since the state-transition model includes all state changes within the system, these nested conversations will always be correctly nested.

An implementation for the conversation mechanism using the concurrent programming language occam is presented. Again the system is considered to consist of a set of communicating sequential processes, communicating via message passing. The implementation is designed to be functionally independent, i.e. the conversation has the same structure whatever the application. This enables the designer to place the function requiring protection into the conversation structure and write the appropriate acceptance tests for each exit process in the conversation. Occam communication channels (each of which link two named processes only) are used to remove the problem of information smuggling by ensuring that no channel belongs to a process outside the conversation.

The communication primitives used in message

passing systems are shown to be vulnerable and require careful design when applied to systems requiring high integrity. Two message types were identified into which the communication types can be classified. These message types were modelled using the same state-transition model developed in chapter 3. Boundaries are identified for the placement of a breakout mechanism which will increase the integrity of the communications.

7.2 Future Work.

The implementation for the entry and exit statements of the conversation scheme has no mechanism to cope with the problem of a deserter process [25]. If a process is included in a conversation, but never enters it, then the other processes in the conversation will be blocked from exiting. A possible solution to this problem would be to place a timeout mechanism around the conversation block. This would ensure an exit from the conversation. However, this would complicate the recovery strategy considerably and is not easy to implement in the proposed system. A time-independent deserter detection mechanism would be more desirable.

This thesis has dealt with the problem of designing and implementing a backward error recovery mechan-

ism for a distributed system consisting of a set of communicating sequential processes. The other main mechanism for dealing with errors in software is the forward error recovery mechanism. The two mechanisms deal with different types of errors and should not be thought of as being mutually exclusive. A higher reliability system could be produced if it incorporated both forms of error recovery mechanisms. Thus, it would seem a very useful extension to this research to investigate the design and implementation of a forward error recovery mechanism which could be used with the design proposed here.

It is felt that the design technique described in chapter 4 could lend itself very well to an automated design procedure. Taking the Petri net graph of the program as a starting point, it should be possible to generate the reachability tree automatically, with possible operator input for certain timings of transitions. Once the reachability tree has been produced the state-change and communications state-change tables can be produced by using simple set operations and the rules given for identifying communication transitions. The designer could then specify where he requires the conversation boundary on all processes or any one process and the system will produce a minimum set of entry and exit states for the conversation. It should also be possible for the system to detect if

nested conversations are correctly nested or not.

The proposed starting point was the Petri net graph of the program. Given the transformation techniques in chapter 3, it is thought possible to automate this part of the design technique also. If possible, it would give a design tool which takes as its input a concurrent program or specification (in the form of an occam program) and produces minimum sets of entry and exit states for properly nested conversations. It would seem that this could be a very useful and powerful tool to have when designing fault tolerant distributed software.

Work is still required on the proposal for more reliable communications. The system proposed in this thesis has not been implemented. Since it requires an extension to the language occam. To take this proposal further a means would be required to actually add this extension to the language and then run tests on the proposed mechanism.

Appendix A.

Published Papers.

Aspects of Software Engineering for Systems with Safety Implications.

Eurocon '84, Brighton 1984.

The Design of Communication Software for Distributed Multivariable Control Systems.

Symposium on Application of Multivariable System Techniques, Plymouth 1984.

Guidelines for the Synthesis of Software for Distributed Processes.

PES3 Conference, Guernsey 1986.

Design of Reliable Software in Distributed Systems Using the Conversation Scheme.

IEEE Transactions on Software Engineering, Vol. SE - 12, No. 9, Sept. 1986.

ASPECTS OF SOFTWARE ENGINEERING FOR SYSTEMS WITH SAFETY IMPLICATIONS

D J Holding, G F Carpenter, A M Tyrrell

University of Aston in Birmingham, UK

INTRODUCTION

Microprocessors and other programmable electronic systems are being used increasingly in communication and control systems for the real-time control of process plant. Modern microprocessors now provide, on a single integrated circuit, the processing power of a more than competent minicomputer of only a few years ago. The processing capability of the microprocessor can be enhanced by complementary families of programmable interface and communications devices such that both open and embedded microelectronic systems can be built with full computing systems capabilities. These microprocessor-based systems have the computational power, systems capabilities, low power consumption, high reliability and low cost necessary for widescale use in centralised and distributed control and communication systems.

Software is used to define the information processing activities which turn the hardware into a functioning system. The required behaviour and performance will be achieved only if the software is specified correctly, and if the design satisfies the specification and is implemented properly. The systems and applications software necessary to apply a programmable electronic system to its task is basically independent of the size of the processor, but does depend on the power and complexity of the processor or processing systems. The problems of software generation have not been reduced as the hardware processing systems have been miniaturised.

The task of designing software for process control systems is not trivial. Many applications require the computing system to provide real-time sequential and continuous control over a number of process loops which may have safety implications. These systems are often required to produce computed responses within critical time constraints, even though the control program may be event-driven and subject to external interrupts. Multi-tasking systems of this type make severe demands on the systems software (which is responsible for the allocation and control of the computing resources) and on the applications software (which is responsible for the control of physical plant and processes). Much work has been done on the development of techniques for use in the design of such systems. More recent work has concentrated on methodologies for the application of these techniques.

The trends outlined above have added impetus to the development and establishment of microprocessor-based distributed control systems which consist of sets of controllers linked by a suitable communications network. Decentralised systems of this type require careful co-ordination and synchronisation. Although the software design of such a system can be complex, it may be helped by the formal notations and constructs developed in recent high-level languages for both conventional (sequential) and concurrent (communicating sequential) processes. These concepts are now being designed into the kernel of an emerging generation of microprocessors which should contribute much to systems design. Equally important, the software techniques on which these systems are based may be applied to the design of certain control systems and will provide a good foundation for research into new designs.

The rapid increase in the application of programmable electronic systems in the field of communication and control has been accompanied by a corresponding increase in the community of designers and users of such systems. Those involved must have an awareness of the specific demands imposed on the software, particularly in high performance systems which involve safety functions or have implications for safety. The quality of the software will be critically dependent on the use of proper procedures and techniques in all stages of the software life cycle (Daniels (1)).

This paper examines the various performance and environmental constraints which arise in communication and control applications and relates them to specific problems in software design. Well-established techniques for making such systems safe and robust are identified. The paper also draws on known techniques for specification and implementation, and describes current areas of research and concern which may affect design, particularly in distributed systems.

APPLICATIONS ENVIRONMENT

A typical microprocessor-based communication and control system will consist of a set of one or more processors which may be closely or loosely coupled to form an information processing system, and which will be interfaced to a physical process or plant via suitable data acquisition and control actuation units. The system will also be interfaced to man-machine interface units. The system will be classified as 'decentralised' if inter-processor communication is necessary for the overall control of the plant. The term 'distributed' will be used to describe the physical distribution of the processors.

The software for such a system can be partitioned functionally into the systems software, which governs the operation of the set of processors, and the applications software, which defines user tasks. The systems software provides the host environment for the applications software, and controls its execution. The applications software will manage the resources of the physical plant and will control its operation. Both the systems and applications software must be designed properly and implemented correctly

if the system is to carry out specified tasks in a predictable and safe manner.

The requirements specification for a specific application will constrain the type of host environment. It may also restrict the type of algorithm which can be used in the applications software. These two factors influence the selection of a suitable design methodology for systems analysis and software synthesis, and will determine the type of design rules and techniques which must be used to generate correct and safe software. The selection of the most appropriate method and technique should increase the probability of generating correct and properly-validated code. This approach can be illustrated best by considering a number of typical applications.

SEQUENTIAL DETERMINISTIC PROCESSES

The simpler host environments can be used when the software processes are independent of the sequence and time of occurrence of real-world events. These characteristics apply to much of scientific computing, such as data reduction and analysis; they are also typical of many programs which acquire their data by sampling under computer direction. The computations involve sequential and deterministic programs in which the algorithm specifies the sequence of operations which must be performed on the input data to obtain the desired output data. The techniques which can be applied in the specification, design, development and maintenance of sequential and deterministic processes have been summarised recently in the STARTS guide (2).

During the requirements specification phase the specification may be expressed in natural language or in a more formal notation. The use of a specification language is recommended (Ross (3), Sommerville (4)) to remove the inherent ambiguity and complexity of a natural language description. The specification languages which are available include general purpose notations for expressing requirements (for example, Bell et al (5)) and specialist notations developed for specific applications (thus, PSL/PSA for information processing, Teichrow and Hershey (6)).

The specification phase will include the identification of the methodologies to be adopted in design and assessment of the product. The STARTS guide identifies the need for appropriate management during the specification stage when the activity is chiefly intellectual and specifications rather than program code are being produced. However, this stage is necessary if costly re-iteration of the early design phase is to be avoided at a later stage in the lifecycle (Boehm (7)).

The program design phase will use the techniques of top-down analysis which have been widely described (Welsh and McKeag (8), Alagic and Arbib (9)). The principal technique involves successive functional partitioning in which the transformations affecting the data are analysed and refined. The resulting program modules will each perform a distinct function and will have a clearly-defined interface with other modules. The philosophy of structured programming is also used in the internal design of the modules.

A second and parallel activity carried out within the design phase is design for test, since in general it is not practical to prove that the software is free of faults. Tests can be specified during the top-down design, and executed during the test phase to detect faults. Stepwise refinement should constrain the scope of the modifications required to remove a fault and thus correct the error.

The specification will state the degree of robustness required under fault conditions. This will determine the extent to which fault-tolerant computing techniques should be used (Anderson and Lee (10)).

The third and largest phase in the software life cycle involves the maintenance of the software. This includes perfective maintenance which encompasses changes in the specification of the program, adaptive maintenance which arises from changes in the program environment, and corrective maintenance to remove residual faults. In safety applications criteria for controlling and accepting changes to the program must be applied rigorously.

Many users have experience and substantial ability in the field of designing prescriptive sequential deterministic programs to be run in a single-user environment or within a protected multi-user environment. Unfortunately, this computational environment does not exercise some of the most crucial techniques required in the control of complex systems, real-time systems, or time-critical real-time systems.

CONCURRENT PROCESSES

More complex host software is required when more than one software process is allowed to exist concurrently in a system. Typically this occurs when a task with implicit concurrency is partitioned into a set of communicating sequential processes, each of which can be executed concurrently. Techniques for the control of concurrent processes and the assignment of resources in concurrent systems were developed during the late 1960s and the early 1970s (Brinch Hansen (11), Dijkstra (12), Hoare (13)). These methods involve the identification of critical sections and the provision of mechanisms for enforcing mutual exclusion using semaphores and monitors.

It must be emphasised that the techniques of resource assignment, inter-process communication and synchronisation apply to concurrent systems in general. These techniques must be used in the control and allocation of the computational resources among the concurrent software processes, and in the control and allocation of the physical concurrent processes in the external plant. A proper understanding of the techniques is essential if they are to be used correctly in specification, design and implementation.

More recent developments in the theory of communicating sequential processes (Hoare (14)) have led to the development of formal methods for synchronising concurrent processes through communications. These extensions can be used to describe both centralised systems in which the processes exist within one processor, and distributed and decentralised systems in which the processes are distributed over a set of processors. In a general distributed system, there will not be a centralised monitoring facility to guarantee orderly processing among the processes. Process interaction, to collect or distribute data, or to promulgate control decisions, takes place through inter-process communication. Each process will advance asynchronously with its computations, unless

forced into synchronism by communication. Consequently, the integrity of the communication medium in this type of system must be a paramount concern.

Three forms of inter-process communication primitives can be recognised.

In a synchronous system the task initiating the communication (the source task) requires an acknowledgement from the object task before it can proceed; the process that reaches the communication point first must wait for the other before it can continue (see (14) and May (15)). The usual notation is:

```
SOURCE TASK                    OBJECT TASK

object ! <request>; -----> source ? <request>;
```

Process synchronisation is thus enforced; subsequently both processes proceed autonomously.

In an asynchronous system, the source task does not wait for an acknowledgement before it proceeds; the object task, however, will wait if it arrives at the communication point first (Liskov (16)). The notation is:

```
SOURCE TASK                    OBJECT TASK

SEND <request>
TO <object>;  ---------> RECEIVE <request>
                         FROM <source>;
```

A third form of inter-process communication is the remote procedure call, used most notably in Ada (17), but in wide use elsewhere (Brinch Hansen (18), Mao and Yeh (19), Kramer et al (20)). Here the source task invokes the object task to perform a specific function and waits for the object task to respond before continuing. The notation is:

```
SOURCE TASK

  objecttask.request <in-parameter,
                      out-parameter>;

OBJECT (PROCEDURE) TASK

  accept request <in-parameters> do
    <service request - send out-parameters>;
```

For distributed control systems, it is common practice to classify inter-process messages according to the function they require of the object task (Kramer et al (21)). Commands require a change of state or action in the object task; acknowledgement from the object task is a necessary requirement. Status messages are sent to give information about the source task; such messages may be issued upon command from another task, in which case acknowledgement from the object task may not be necessary. However, if a change of state in the source task indicates an alarm condition, then the source task may need to initiate an Alarm message; this would have high priority and would require an immediate response.

Analysis shows that in process control systems it is possible to identify two groups of messages: command messages, which require a reply to the initiator, and notify messages, which require no reply to the initiator. The communication primitives introduced above appear to support these message groups. In practical process control, however, it will be shown that the constructs are deficient and open to ambiguity.

STATE- OR EVENT-DRIVEN SYSTEMS

The number of processes within a concurrent processing system may vary as new independent processes are created or as existing processes are terminated. The determinism of such a system is destroyed if the creation, activation or termination of processes is not initiated by, and synchronised from, existing processes. Non-deterministic systems cannot be tested exhaustively, and the quality of the software will be dependent critically upon the use of proper techniques at all stages of specification, design and implementation, and upon the application of software quality control methods. The resulting system can only be accessed by qualitative measures of the software, and through statistical records of dynamic tests on the software.

Control applications often require the computational system to respond to an external event. Such an event may be detected as the computer analyses data from the external plant. Alternatively, it may interrupt the computing activity directly, and thus asynchronously create a new process. In the first case the external event will cause a computed response which may change the schedule of processing activities. While the system may eventually return to resume the schedule of processing which existed before the event occurred, the state of the external system and the assignment of external or computational resources may have changed, and the system become non-deterministic. The second case leads directly to a non-deterministic system.

The specification for an event-driven system will necessarily identify the causal events and define the corresponding response actions. It should also identify clearly the subset of events which are asynchronous, for the presence of such inputs will normally result in a non-deterministic system. In systems with safety applications it is preferable that an explicit statement be made of whether non-deterministic behaviour is to be allowed in the application.

REAL-TIME SYSTEMS

The majority of industrial control systems are real-time systems in which a periodic interrupt from an independent time source or 'real-time clock' is used to provide a time-reference event which does not depend on the state of the software processes. The computational system will synchronise its processing schedule on the occurrence of this event. Typically, this synchronisation is used in multi-tasking systems to apportion processing time to the processes according to their scheduling requirements. Such a technique could be used in a multi-tasking control system for applications which depend only on the relative sequence of events.

The design problem is more severe if the application is critically dependent on the absolute time attributes of the system states. For example, many monitoring and control systems are required to sample input data from the plant at specific and equally-spaced instances of time. Waits or delays are often used in control systems to sequence process activities on a relative or an absolute basis. Similarly, closed-loop direct digital control systems usually require inputs and outputs to be synchronised to a real-time schedule. Such systems are considered below.

TIME-CRITICAL REAL-TIME SYSTEMS

The generation of computed responses at specific instances of time is central to nearly all direct digital control (DDC) systems. This forms the basis of real-time sequence control. It is also fundamental to the design of control algorithms for continuous processes, for the sampling period is usually a critical parameter and can radically affect the performance and stability of the system. In extreme cases, plant which is intrinsically unstable will depend on the control system for stabilisation. These systems may be classified as time-critical real-time systems.

The dynamical performance of the above can be monitored at either scheduler or application program level. The timing performance is usually monitored by running a real-time time-lapse counter concurrently with the application task. Both processes will be attempted; the first task to complete succeeds and the other attempt is withdrawn. This 'timeout' technique is used as a 'watchdog' mechanism in real-time control systems.

The specification of a time-critical real-time system will include a definition of the actions to be taken when input data is out of range, or when outputs from processes fail to meet acceptance tests. Similar specifications are required for the actions to be taken in the event of timeouts. These could arise when processes exceed specified execution times, or from a general loss of synchronism at scheduler level. Failures of inter-process or inter-processor communications might also lead to timeout events; however, recovery in multi-process or distributed systems is much more complex and care is required if ambiguity is to be avoided.

Consider, for example, the failure of an asynchronous inter-process communication in which the object task would wait indefinitely to receive the inter-process message. A selective construct with a timeout would only partially resolve the problem due to the absence of any logical pairing between the source and object tasks:

```
SOURCE TASK

Send <request> to <object task>;
Select    Receive <reply> from <object>
   or     Timeout <period>;
```

With synchronous communications, such as in occam (15), a timeout can be placed on the receive primitive to break the wait-for-synchronisation. However, the source task would be suspended if the timeout were activated:

```
SOURCE TASK                OBJECT TASK

                         clock := NOW;
                         ALT
                           WAIT NOW AFTER
                             <clock + timeout> ;
object ! <request>; --> source ? <request> ;
```

In occam, a timeout cannot be placed on the send primitive since outputs are not allowed as guards in synchronous primitives.

With remote procedure calls, as in Ada (17), a timeout can be placed on the acceptance of the initiating message by the object task. However, the transaction is not completed until the reply message is sent by the object task. If this message is lost then the source task is suspended indefinitely:

```
SOURCE TASK

SELECT  TIMEOUT <period>
    OR  objecttask.request
          <in-parameters, out-parameters>;
```

The timeouts used above monitor specific communication primitives. The groups of messages used in control systems would be protected better if the whole transaction were included in the timeout (20). A new notation is required for such a mechanism. Perhaps the greatest advantage can be obtained from a model which uses the formal notation of communicating sequential processes, but which includes a select construct to provide a mechanism to break-out from the formal notation in the event of a timing failure. Thus:

```
SOURCE TASK.

SELECT
       SEQ
         objecttask ! <request>;
         objecttask ? <reply>;
    OR TIMEOUT (period);

OBJECT TASK.

SELECT
       SEQ
         sourcetask ? <request>;
         SEQ
           (service request);
           sourcetask ! <reply>;
    OR TIMEOUT (period);
```

In systems with safety applications, failure modes must be specified and measures taken to generate safe outputs. Specific problems arise in the case of distributed systems with distributed databases in which database migration may occur. The concept of 'conversations' addresses this problem (Randell (22), Kim (23)) and might provide a suitable basis for the design of a recovery mechanism for the 'break-out' described above.

In practical applications a time-critical real-time system may be required to undertake complex processing schedules defined by hierarchical sequential control schemes while maintaining continuous feedback control over a number of concurrent plant-processes. The bespoke design of such systems places the most stringent demands on designers. In many applications limited configurability systems are used in which system-specific safety-critical features are placed within a restricted-access kernel. However, this does not remove the need for the proper design of applications-specific safety functions.

CONCLUSION

The design of programmable electronic systems for safety applications or systems with safety functions places specific demands on the design of these systems. This paper has examined some of the performance and environmental constraints on these systems and has considered techniques for their solution.

REFERENCES

1. Daniels, B.K., 1983, Reliability Engineering, 4, 199-234.

2. Department of Trade and Industry, 1984, "Software Tools for Application to large Real Time Systems".

3. Ross, D.T., 1977, IEEE Trans on software engineering, SE-3, 16-34.

4. Sommerville, I., 1982, "Software Engineering", Addison-Wesley.

5. Bell, T.E., Bixler, D.C., and Dyer, M.E., 1977, IEEE Trans on software engineering, SE-3, 49-60.

6. Teichrow, D., and Hershey, E.A., 1977, IEEE Trans on software engineering, SE-3, 41-48.

7. Boehm, B.W., 1981, "Software Engineering Economics", Prentice-Hall.

8. Welsh, J., and McKeag, M., 1980, "Structured System Programming", Prentice-Hall.

9. Alagic, S., and Arbib, M.A., 1978, "The design of well-structured and correct programs", Springer-Verlag.

10. Anderson, T., and Lee, P.A., 1981, "Fault Tolerance: Principles and Practice", Prentice-Hall.

11. Brinch Hansen, P., 1973, "Operating system principles", Prentice-Hall.

12. Dijkstra, E.W., 1968, "Co-operating sequential processes" in "Programming Languages", 43-112, ed Genuys, F., Academic Press.

13. Hoare, C.A.R., 1974, Comm ACM, 17, 549-557.

14. Hoare, C.A.R., 1978, Comm ACM, 21, 666-677.

15. May, D., 1983, Sigplan Notices, 18, 69-79.

16. Liskov, B., 1979, "Primitives for Distributed Computing", Proc of 7th ACM SIGOPS Symp on Operating Systems Principles, 33-72.

17. United States Department of Defence, 1980, "Reference Manual for the Ada Programming Language".

18. Brinch Hansen, P., 1978, Comm ACM, 21, 934-941.

19. Mao, T., and Yeh, T., 1980, IEEE Trans on software engineering, SE-6, 194-204.

20. Kramer, J., Magee, J., Sloman. M., and Lister, A., 1983, IEE Proc Pt E, 130, 1-10.

21. Kramer, J., Magee, J., and Sloman, M., 1981, "Intertask communication primitives for distributed computer control systems", Proc of 2nd Int Conf on Distributed Computer Systems, 404-411.

22. Randell, B.R., 1975, IEEE Trans on software engineering, SE-1, 220-232.

23. Kim, K.J., 1982, IEEE Trans on software engineering, SE-8, 189-197.

THE DESIGN OF COMMUNICATIONS SOFTWARE FOR DISTRIBUTED MULTIVARIABLE CONTROL SYSTEMS

ANDREW M. TYRRELL, DAVID J. HOLDING

UNIVERSITY OF ASTON

Distributed computer control systems have advantages over a centralised system, including distributed functionality, increased fault tolerance, and piecewise growth capability. These advantages are gained at the expense of increased complexity in the systems software. This paper examines the communication mechanisms required to synchronise and co-ordinate distributed computer-based control systems under normal and abnormal operating conditions. Various formal notations and communication primitives are examined. The limitations of present solutions are determined. A new notation is introduced and used to describe a generalised distributed control function. Finally, a structure in this notation is identified in a wide class of control system.

1.0 INTRODUCTION

Distributed computer control systems offer a number of advantages over a centralised system, including distributed functionality, increased fault tolerance, higher speed of computation through concurrency, piecewise growth capabilities, and piecewise system upgrading [1]. However, these advantages are gained at the expense of increased system complexity.

In distributed systems, the distributed systems and applications software can be a major source of complexity. Much work has been done on the implementation of distributed systems [2-10]. This work concentrates mainly on programming languages for such systems. Work has also been done on the design of these systems [11-15] although, in general, these designs do not address explicitly the issues of communication and synchronisation. However, the kernel of any such design is the communication and synchronisation mechanisms used to coordinate the distributed system; these usually involve a message-based scheme. Mechanisms such as the monitor [17], which rely upon shared memory for such communications, limit the choice of hardware configuration for the system. A more flexible choice of system implementation is obtained if a message passing mechanism is used for inter-process or inter-processor communication.

This paper is concerned with a distributed computer control system in which the process control software consists of a set of distributed processes which communicate by message passing over a communication network. This type of system can be characterised by the following assumptions:

a. There is no centralised monitoring facility or resource management to guarantee orderly processing among the processors.

b. There is no overall clock control: processes therefore advance asynchronously with computations.

c. Processes can only interact by the transaction of passing messages.

d. The communication medium, while generally reliable, may be subject to transient or sustained failure when messages will be lost.

Any technique for representing distributed software must describe explicitly the interactions between tasks on different processors. In time-critical, real-time systems, tasks must be properly synchronised and must satisfy critical timing requirements. The system design must also be robust and incorporate fault-tolerant methods in the software [18,19].

This paper investigates the problems of designing robust software for distributed multi-input, multi-output control systems. Section 2 gives an overview of the different types of communication primitives used in languages intended for distributed systems and describes the applicability and limitations of these communication mechanisms.

Section 3 proposes a notation that is suitable for real-time system specification, design and implementation. This notation is formed by combining the transaction-timing communication controls described in CONIC [12], with some of the synchronous communication primitives developed in C.S.P. [2]. The suggested notation has the advantage of allowing formal methods to be used in the design of the system for normal operations. Break-out mechanisms are implemented to allow recovery procedures where timing constraints are involved.

The notation is used to develop a schematic design for a distributed control system. A number of design stages are identified. To implement the initial stages of the design, a design methodology derived from MASCOT [13] is used. This, combined with the proposed notation, forms a method which will help in the design of software for distributed multi-input, multi-output, time-critical, real-time systems.

Finally, a software structure which appears in each subsystem is identified as a template for use in the design of distributed systems. This software structure template could also be embedded within the recovery mechanisms required in fault-tolerant systems.

2.0 COMMUNICATION PRIMITIVES

In this paper a task which initiates communications will be referred to as a source task. The task to which a communication is sent is called an object task. A channel is the unidirectional medium used for such communications.

2.1 Communication Primitives for Distributed Processes

The different communication primitives used in message passing can be classified into three types: asynchronous, synchronous, and remote procedure call.

a. Asynchronous

An asynchronous communication does not require the object task to acknowledge the receipt of a message from the source. The source task issues an initiating message and then continues its operation. Since synchronisation is not enforced, some form of buffering is required to hold the initiating message should the object task not be ready to receive the message [4,5]. Similarly, the object task may be required to wait if the message has not been initiated by the source.

The asynchronous communication has two primitives; these are of the form:

```
SEND    <message>    TO    <object.task>;

RECEIVE    <message>    FROM    <source.task>;
```

b. Synchronous

In a synchronous communication the source task requires an acknowledgement from the object task before it can proceed. The process that reaches the communication point first must wait for the other process before it can continue. The source task may then issue an initiating message. On receipt of this message, the object task will issue an acknowledgement message. When the source has received the acknowledgement both processes may continue autonomously. Process synchronisation is thus enforced through communication [2-4].

Synchronous communication also has two primitives:

```
objecttask   !   <message>;     --send from source task
                                -- to object task <message>.

sourcetask   ?   <message>;     -- receive at object task
                                -- <message> from source task.
```

c. Remote Procedure Call

The third type of communication primitive is the remote procedure call. The source task waits for the object task to report completion of the procedure. The object task performs a specific function for the source task in a way similar to that of a subroutine. The object task is not initiated until it has received the source's message and completes its local computation before issuing a reply [7-12].

The remote procedure call primitives have two parameters in their communications as shown below:

```
SOURCE TASK

object.request <in,out parameters>;
```

```
OBJECT (PROCEDURE) TASK

accept.request <in parameters>

        do <service request

            send out parameters>;
```

2.2 Message Types in Distributed Computer Control Systems

Messages are used widely in distributed computer-based control systems for the collection or distribution of data and to promulgate control decisions or actions. In [16] three main functional classifications or transfer categories are identified:

COMMANDS are messages which cause a change of state or action in the object task. These messages generally require a response from the object task to signify completion of the action.

STATUS messages are sent by a source task to a number of object tasks and are used to convey source status. These messages may be initiated by the source either periodically or when the state of the source task changes. Status messages may also be generated by an object task in response to a request from a source task.

ALARMS are messages initiated by the source to inform the other tasks that the controlled process is malfunctioning or is in an unsafe state.

These three message types are divided into two groups in [16]:

Command Message Group: Those requiring a reply to their initial message: command and requested status messages.

Notify Message Group: Those requiring no reply: periodic status, event status and alarm messages.

2.3 Communication Primitives for Distributed Computer Control Systems

The message groups can be modelled using the communication primitives of section 2.1. Analysis shows that when these groups of messages are directly implemented they contain ambiguities and deficiencies.

2.3.1 Command Message Types

The command message can be modelled using the primitives described in Section 2.1:

Asynchronous Implementation:

```
SOURCE TASK                OBJECT TASK

SEND <request>

TO <object>;  --------->  RECEIVE <request> FROM

                           <source>;

                           (service request)

RECEIVE <reply><-------  SEND <reply> TO <source>;

FROM <object>;
```

The communication subsystem of a distributed computer control system may be subject to failure. It is common to incorporate a timeout mechanism which will initiate the appropriate recovery mechanism. The timing requirements of such a system are only partially satisfied if the object task RECEIVE primitive is put in a SELECTIVE construct with a timeout:

```
SOURCE TASK

SEND <request> TO <object>;


SELECT

      TIMEOUT <period>

    OR

      RECEIVE <request> FROM <object>
```

When the select statement is executed both the communication and timeout tasks are attempted. Whichever task is completed first is defined as executed and the other attempt is withdrawn. However, such a model is still ambiguous because of the absence of any logical or notational paring between the two halves of the transaction. This could lead to a situation in which the object task sends a message to a timed-out source task.

Synchronous Implementation:

```
SOURCE TASK                OBJECT TASK

object  !  <request>;  ------>source  ? <request>;

                                (service request)

object  ?  <reply>;  <-------- source  !  <reply>;
```

In this case the logical clarity is good. However, the timing requirements are not easily satisfied. For example, if a timeout was placed on the receive primitive to break the wait-for-synchronisation, then the source task could be suspended if the timeout was activated:

```
SOURCE TASK              OBJECT TASK

                              clock := NOW;

                              ALT

                                WAIT NOW AFTER <clock +

                                               timeout>

object  !  <request> ----->     source ? <request>
```

The send primitive cannot be modified to include a time-out by using an ALT mechanism, because outputs are not allowed as guards in synchronous primitives.

The remote procedure call primitive simulates the command message directly:

Remote Procedure Call Implementation:

```
SOURCE TASK

  objecttask.request <in-parameter,

                      out-parameter>;

OBJECT (PROCEDURE) TASK

  accept request <in parameters> do

          <service request - send out parameters>;
```

Remote procedure calls of this type are used for communication in the Ada language [7]. The Ada implementation uses a timeout on the acceptance of the message by the object task and has the form:

```
SOURCE TASK

SELECT

       TIMEOUT <period>

     OR

       object.request <in,out parameters>;
```

This system is again ambiguous, for if the reply message is lost, then the source task is suspended indefinitely. A solution to this problem was included in CONIC [12] in which the timeout is placed on the completion of the whole transaction.

The communication models outlined above must be assessed as formal notations for the expression of concurrency, the pairing of communication primitives, and the ability to incorporate timeout mechanisms successfully. None of the models outlined satisfy all of these objectives. The first two objectives are satisfied by the synchronous communication system which is developed in C.S.P. [2] and embedded as primitives in the derived

language occam [3]. The formal semantics of C.S.P. allow certain proofs of program correctness and include correctness preserving transformations. However, the scope of the language does not, of course, include timeout mechanisms. The third objective is met by CONIC.

3.0 PROPOSED SYSTEM DESIGN

The two separate lines of development have led to communication mechanisms such as CONIC which satisfy the transaction and timing requirements, and to synchronous communication systems such as those in occam which provide a formal notation, and mechanisms for the implementation of strictly synchronous systems. The following section explores the advantages to be realised by developing a formal construct which includes select and 'breakout' to provide the flexibility required to satisfy timing constraints.

3.1 Notation for Communications in Real-Time Control Systems

Consider the case of a command group message implemented as a synchronous communication by message passing, with the inclusion of a select mechanism which provides an ability to break out of the formal notation in the event of system behaviour conflicting with timing requirements.

```
SOURCE TASK

SELECT

      TIMEOUT (period);        --time-out if not completed by period

   OR

     SEQ

       objecttask  !  <request>;    --send request to object task

       objecttask  ?  <reply>;      --wait reply from object task

OBJECT TASK

SELECT

      TIMEOUT (period);        --time-out if above not completed by

                               -- period

   OR

     SEQ

       sourcetask  ?  <request>;    --receive from source request

       SEQ

         (service request);         --service request

         sourcetask  !  <reply>;    --send reply to source
```

In the normal sequence of events the synchronous communication will be successful. However, there are two cases when the timeout will be chosen:

i. When either the source or the object task is waiting for the other task to reach the communication point. In this event the communication will not have been initiated.

ii. After the initiation of the object task but before completion of the whole transaction.

In both cases the processes will have to be rolled-back to the start of the communication for recovery. The time-out on the whole transaction will prevent either task from being suspended indefinitely. This communication transaction built from occam-type primitives adds real-time communication to the notation.

3.2 Distributed Software Design

When designing software for distributed systems the design is split into a number of stages:

i. Partitioning of the problem into subsystems on a functional basis.

ii. Identification of concurrency.

iii. Design of process structure into sequential and parallel sub-components.

iv. Design of inter-process communication. Identification of those channels which require a real-time, time-critical communication such as those described in 3.1.

v. Design of initialisation and termination mechanism for the communication system.

vi. Design of processes.

3.3 Control Station Design

The above design procedure was used in the schematic design of a system consisting of two control stations each having part of the total system state information (Figure 1). To ensure total system controllability the observability, communications between systems were required [22].

For the initial stages of the design the real-time design methodology MASCOT [13] was used. MASCOT provides a unified discipline of modularisation based upon the decomposition of a program into a set of parallel processes. It also provides tight control over intra-processor communications. Tasks within a processor can only exchange data via data structures: the channel or the pool. Direct task-to-task communication is prohibited.

The control station was partitioned into eight subsystems in the initial stage of the design (Figure 2). Each of these subsystems was then decomposed into a number of simpler processes interacting via pools and channels (Figure 3).

A fragment of program was written specifying the communication between tasks and data structures. Each communication primitive was examined and the select mechanism introduced in those instances in which breakout facilities were required to satisfy timing constraints. Those tasks receiving data from pools required standard synchronous communications. However, tasks reading data from channels require extended communication primitives. In Figure 3 tasks that require the extended communication are shown in double rings.

Each task consists basically of an input channel using extended communications, a task taking inputs from a standard communication channel, and an output channel. This fundamental structure was found to be embedded within each subsystem (Figure 4).

3.4 Use of Template

Having identified a basic template which can be used in the design of all the subsystems, the design of robust systems can be simplified. Each template could be embedded within a fault-tolerant mechanism which would ensure correct operation of the system. For example, the recovery block mechanism proposed by Randell [17] could be used to provide fault tolerance with sequential processes. A number of different mechanisms have also been suggested for concurrent system fault tolerance [18-21]. The conversation first proposed by Randell [18] and extended by Kim [20] would be applicable in this case.

4.0 CONCLUSIONS

This paper has examined some aspects of the problems of designing software for distributed multi-input, multi-output control systems. It takes the view that in such systems communication and synchronisation should be addressed explicitly. It has shown that languages proposed before are not matched exactly to the design of distributed systems, or have deficiencies when applied to time-critical, real-time situations. However, the timing requirements of such systems are satisfied by CONIC while the formal notation of synchronous languages such as occam satisfy the concurrency and communication needs of strictly synchronous systems.

The paper has proposed a notation for the design of command group messages in distributed computer control systems. This notation provides strictly synchronous communication primitives in normal operation, but includes break-out facilities necessary to satisfy timing constraints.

The schematic software design for a control station in a distributed control system was formulated as a set of communicating processes. An analogous MASCOT design of the system was expressed in the proposed notation.

Examination of the design showed that each communicating process could be built about a common software structure. It is suggested that this structure is a good candidate for embedding within a fault-tolerant framework.

5.0 REFERENCES

[1] Halsall, F., Grimsdale, R.L., Shoja, G.C. and Lambert, J.E. DEVELOPMENT ENVIRONMENT FOR THE DESIGN AND TEST OF APPLICATIONS SOFTWARE FOR A DISTRIBUTED MICROPROCESSOR COMPUTER SYSTEM. I.E.E. Proc., Vol.130, Pt.E. (1983).

[2] Hoare, C.A.R. COMMUNICATING SEQUENTIAL PROCESSES. Comm.ACM, Vol.21, No.8, August 1978.

[3] May, D. OCCAM. Sigplan Notices, Vol.18, No.4, April 1983.

[4] Boussinot, F., Martin, R., Memmi, G., Ruggiu, G. and Vapne, J. A LANGUAGE FOR FORMAL DESCRIPTION OF REAL-TIME SYSTEMS. IFAC Safecomp, 1983.

[5] Liskov, B. PRIMITIVES FOR DISTRIBUTED COMPUTING. Proceedings of 7th ACMSIGOPS Symposium on Operating Systems Principles, December 1979.

[6] Feldmann, F. HIGH-LEVEL PROGRAMMING FOR DISTRIBUTED COMPUTING. Comm.ACM, Vol.22, No.6, June 1979.

[7] REFERENCE MANUAL FOR ADA PROGRAMMING LANGUAGE. United States Department of Defence, 1980.

[8] Brinch Hansen, P. DISTRIBUTED PROCESSES: A CONCURRENT PROGRAMMING CONCEPT. Comm.ACM, Vol.21, No.11, November 1978.

[9] Mao, T. and Yeh, T. COMMUNICATION PORT: A LANGUAGE CONCEPT FOR CONCURRENT PROGRAMMING. I.E.E.E. Trans. Software Engineering, SE-6, No.2, March 1980.

[10] Tsukamoto, M. LANGUAGE STRUCTURES AND MANAGEMENT METHOD IN A DISTRIBUTED REAL-TIME ENVIRONMENT. IFAC 3rd DCCS Workshop 1981.

[11] Grimsdale, R.L., Halsall, F., Martin-Polo, F. and Wong, S. STRUCTURE AND TASKING FEATURES OF THE PROGRAMMING LANGUAGE MARTLET. I.E.E. Proc. Vol.129, Pt.E, No.2, March 1982.

[12] Kramer, J., Magee, J., Sloman, M. and Lister, A. CONIC: AN INTEGRATED APPROACH TO DISTRIBUTED COMPUTER CONTROL SYSTEMS. I.E.E. Proc., Vol.130, Pt.E, No.1, January 1983.

[13] Jackson, K. and Simpson, H.R. MASCOT - A MODULAR APPROACH TO SOFTWARE CONSTRUCTION OPERATION AND TEST. RRE Tech. Note No.778, October 1975.

[14] Darondeau, Ph., Le Guernic, P. and Raynal, M. ABSTRACT SPECIFICATION OF COMMUNICATION SYSTEMS. Proceedings 1st International Conference on Distributed Computing Systems, October 1979.

[15] Yau, S., Yang, C.C. and Shatz, S.M. AN APPROACH TO DISTRIBUTED COMPUTER SYSTEM SOFTWARE DESIGN. I.E.E.E. Trans. Software Engineering, SE-7, No.4, July 1981.

[16] Kramer, J., Magee, J. and Sloman, M. INTERTASK COMMUNICATION PRIMITIVES FOR DISTRIBUTED COMPUTER CONTROL SYSTEMS. Proceedings of 2nd Internaticnal Conference on Distributed Computer Systems. April 1981.

[17] Hoare, C.A.R. MONITORS: AN OPERATING SYSTEM STRUCTURING CONCEPT. Comm.ACM, Vol.17, No.10, October 1974.

[18] Randell, B. SYSTEM STRUCTURE FOR SOFTWARE FAULT TOLERANCE. I.E.E.E. Trans. Software Engineering, SE-1, No.2, June 1975.

[19] Campbell, R.H., Anderson, T. and Randell, B. PRACTICAL FAULT-TOLERANT SOFTWARE FOR ASYNCHRONOUS SYSTEMS. IFAC Safecomp, 1983.

[20] Kim, K.H. APPROACHES TO MECHANISATION OF THE CONVERSATION SCHEME BASED ON MONITORS. I.E.E.E. Trans. Software Engineering, SE-8, No.3, May 1982.

[21] Russell, D.L. and Tiedeman, M.J. MULTIPROCESS RECOVERY USING CONVERSATIONS. Proceedings FTC-9, 1979.

[22] Momen, S.E.M. and Holding, D.J. CONTROL AND COMMUNICATION STRUCTURES IN DISTRIBUTED CONTROL SYSTEMS. Proc. I.E.E. International Conference on Control and its Applications, Publication 194, March 1981.

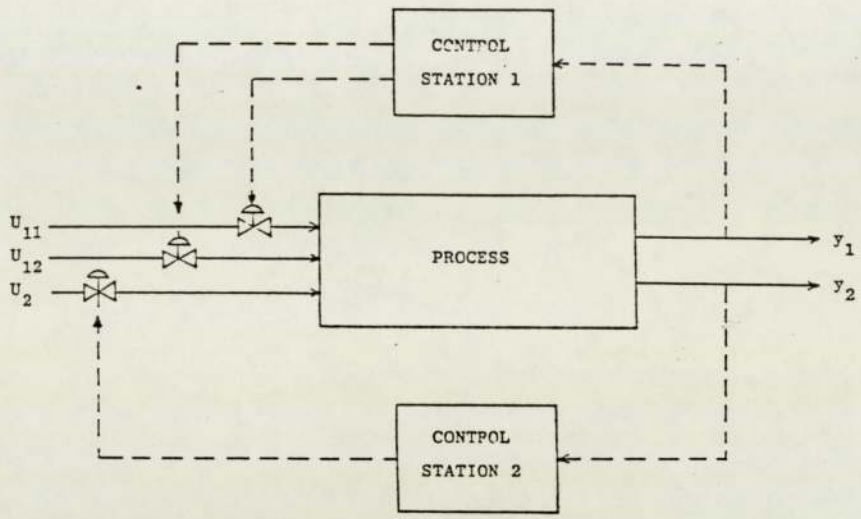


Figure 1: CONTROL SYSTEM

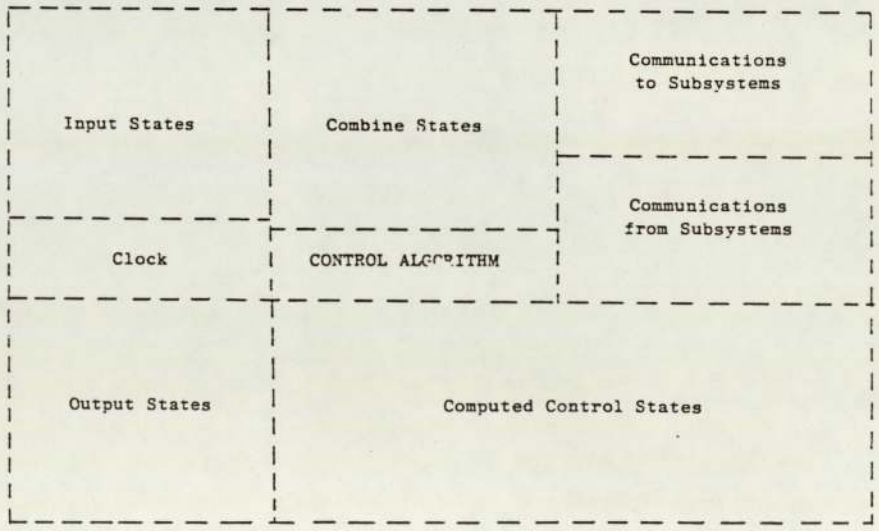


Figure 2: PARTITIONED STATION

202

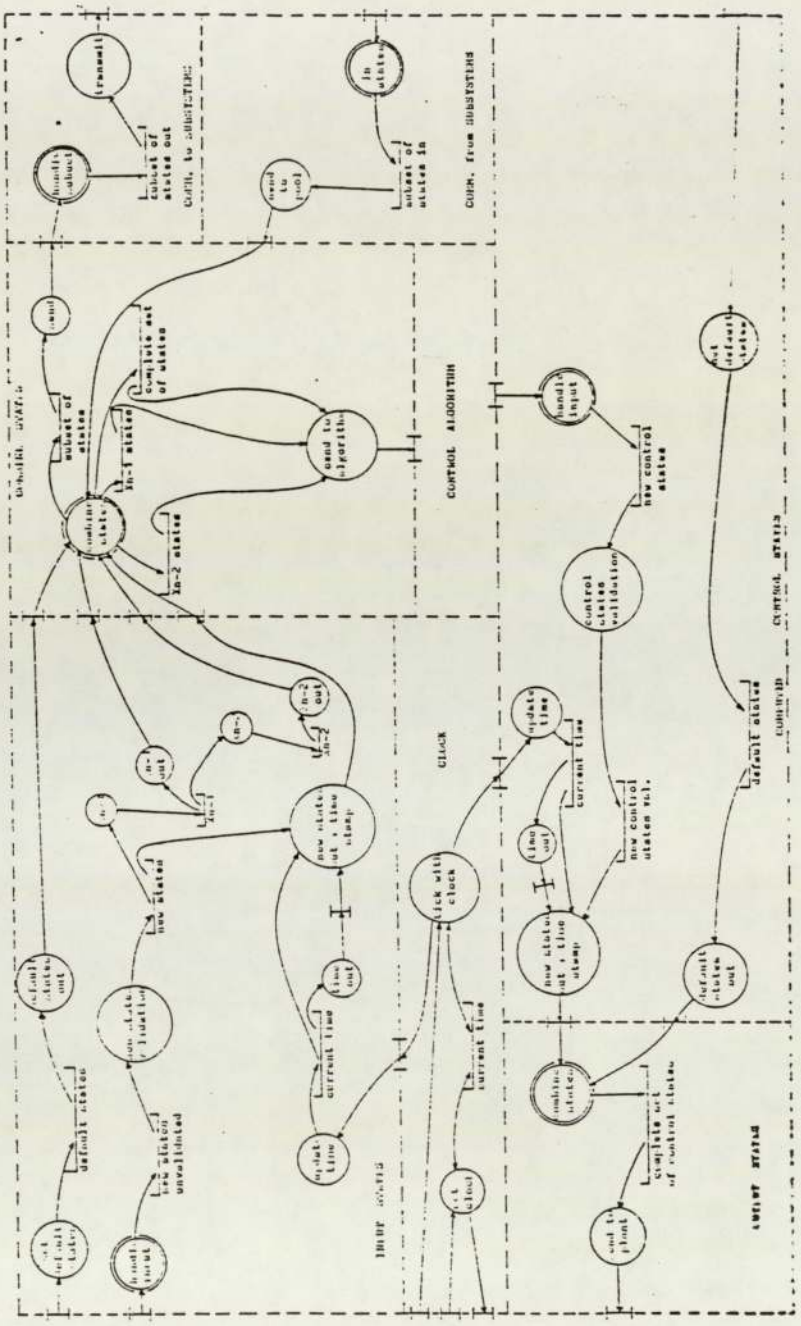


Figure 3: ACP DIAGRAM OF CONTROL STATION

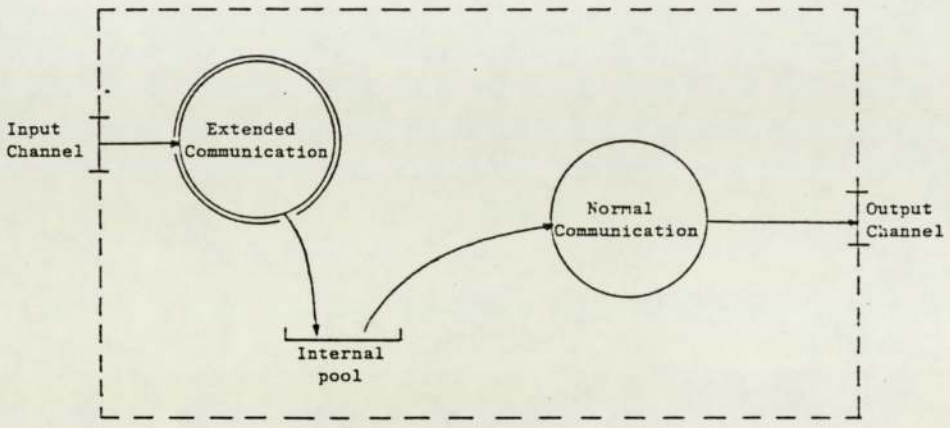


Figure 4: TEMPLATE OF SUBSYSTEM

GUIDELINES FOR THE SYNTHESIS OF SOFTWARE FOR DISTRIBUTED PROCESSORS

CARPENTER, G.F., TYRRELL, A.M. and HOLDING, D.J.

Department of Electrical and Electronic Engineering
and Applied Physics
Aston University
Birmingham

ABSTRACT

A system of distributed processors offers an attractive method for the control of many real-world systems, with the prospect of increased efficiency, throughput and reliability. Modern software engineering analysis methods, design techniques and programming languages should be used in the construction of such systems to control and exploit the parallel nature of the system. Where a robust system is required, particular attention must be paid to the role of interprocess communications, because they provide not only a mechanism for synchronising and co-ordinating the distributed system, but also a mechanism for the propagation of errors. A proper fault tolerant framework must be implemented to restrict such error propagation and to provide proper conversation error-recovery mechanisms.

INTRODUCTION

Microprocessors now offer high computational power, high reliability and low power consumption at a low cost. They are finding widespread use in instrumentation and control systems where the microprocessor provides a centralised computing resource. Increasingly, microprocessors are being used in the construction of decentralised and distributed systems, in which a number of processors are physically distributed about the application plant and interact, or exchange information, with each other by passing messages over interprocessor communication channels. The individual processors in these systems not only provide local functions, such as data acquisition, control, and operator interfaces, but also form part of an overall system which must be co-ordinated to give a global response.

The primary concern in the construction of a computational system is to produce a design which satisfies the requirements specification of the system. The question of whether a computing resource should be implemented as a centralised or distributed system may be only of secondary importance. When a satisfactory design has been generated, and a computational architecture selected, a software specification for the

chosen system has to be drawn up. Whether the computational parts of the system be centralised or decentralised, the function is determined by the software. Software must therefore be designed which meets the software specification and this design must be converted accurately into a program implemented on the target processing system. The resulting system will only operate correctly if the software is properly designed. For practical applications, the production of correct software is non-trivial.

The requirements specifications of monitoring and control systems often demand high levels of performance from a computational system. For example, the computational task may involve real-world data acquisition, combinational or sequential logic functions, complex arithmetic calculations, and the generation of control outputs to the application plant. The computational response may be required within very tight time constraints, perhaps as part of a real-time schedule. The schedule may have to be maintained in the presence of asynchronous external inputs, such as operator commands or alarms. In addition, the system may have to perform safety functions or functions with safety implications.

Requirements of this type make severe demands on the software, both at a systems level (involving the control and allocation of processor resources), and at the application level (responsible for the control of the plant). The design of such systems requires a proper understanding and application of the appropriate design techniques. These include, in the case of distributed systems, methods for the design of concurrent processing systems. The quality of the software, and of the resultant system, is critically dependent upon the adoption of proper methods and disciplines throughout the software life cycle (1).

This paper addresses some of the problems involved in the design of software for distributed processors, particularly where there are implications for safety. Modern software engineering techniques and languages are used to consider possible approaches to the design of such systems, and to discuss methods of providing fault tolerant structures for high reliability applications.

DESIGN CONSIDERATIONS FOR DISTRIBUTED PROCESSES.

The requirements specification for a computer system is chiefly concerned with identifying the functions which the system has to perform, the interfaces with the plant, and constraints within which it must operate. At this stage it is unlikely that a definite need to decentralise the computational system, or to distribute it, will have been identified. Indeed, only a detailed analysis of the requirements may lead to the decision that a distributed system is appropriate. The decision will normally be based on the following characteristics:

- Functional distribution

A real world system may be naturally distributed in a functional sense. Functionally distributed systems are often modelled and controlled as a set of distributed processes. The software for such a

system invariably reflects the distributed nature of the application. This should provide a good correspondence between the real-world system function and the computational function.

- Geographical distribution

Real world systems are often spatially, or geographically, distributed. It is then appropriate to distribute the computational resource across the plant, and to design software which can be implemented over the set of physically distributed processors. Such software will necessarily consist of a set of communicating processes. Since many geographically distributed applications also have functionally distributed attributes, then both characteristics naturally lead to software designs which consist of a set of communicating distributed processes.

Once the decision to distribute the system is taken, then the software design and synthesis must adopt design rules and techniques which will lead to a high probability of generating correct, properly validated code within the specific demands of a distributed system (2).

- Partitioning and the reduction of complexity

The technique of partitioning is used to divide a system into a set of processes. The criteria used to partition a system can alter the extent to which interprocess communications are necessary to maintain the overall system function. System partitions are often chosen to emphasise the physical topology of the plant, the functional characteristics of a system, or the physical location of the processors. If they are chosen so that they emphasise the dominant characteristics of a system, they may give, to a first approximation, a fully decoupled system.

In many cases the partitions lead to an apparent reduction in the complexity of the system, or allow aggregation to reduce design complexity. However, the granularity introduced by partitioning should be carefully considered because it will affect the type of system implementation. For example, as the number of parallel processes into which a computational task is partitioned is increased, so the volume of interprocess communications for control and data interchange is also increased, thus leading to a closely coupled system implementation.

- Concurrency

Physical processes in continuous plant inherently involve the flow of energy or materials which often flow simultaneously through parallel forward paths, or forward and feedback paths. When such systems are modelled, the parallel processes are represented by parallel or concurrent data flows and are readily amenable to parallel processing for model simulation or control. This removes the constraint of modelling these systems in sequential terms which is required for solution by sequential computer programs executing on computers with a Von Neumann architecture. Concurrent programming languages and computing systems can therefore be regarded as the digital equivalent of the analogue computers, simulators and control systems which find widespread acceptance

and continuous use in industry.

DISTRIBUTED PROCESSING

Each process (or processor) in a spatially or functionally distributed system may be equipped with local data acquisition or control interfaces. If each process is operated independently without communications with other processes then the system is said to be decoupled and each process can only operate asynchronously and autonomously and execute its local function only. Unfortunately, few practical applications have the characteristics necessary for decoupled control.

If a system can be controlled using a network of communicating processes, then the system is said to be coupled. The volume of interprocess communications determines the degree of coupling. In a loosely coupled system, relatively infrequent interprocess communications can be used to compute partitioned functions or to co-ordinate the distributed processes. A system is said to be closely or tightly coupled if there is a closer coupling between the component processes such that a high degree of interprocess communication is required to control and co-ordinate the system. Since the availability of communication links is often limited and the bandwidth of such links decreases with distance, closely coupled systems are often implemented as sets of processes communicating through shared memory on a centralised single or multiprocessor computing resource. Loosely coupled systems on the other hand can easily be implemented as geographically or spatially distributed systems.

A distributed system is said to be decentralised if the distributed processes have incomplete and non-identical information about the system state. Such a system requires the co-operative action of constituent processes in order to provide total system observability, controllability and overall function. The distribution of system function and the decentralisation of information can be used to enhance the robustness of the system.

For example, if the system is designed with the ability to recognise failure and can identify the processor or process concerned, then it may be possible to contain and isolate the fault. In distributed systems error migration through communications is a particular problem and it may be necessary to backtrack and trace or limit the effect of the erroneous communications. The reliability of the system may also be increased by the use of fault recovery techniques. If the fault leads to decreased functionality, then it may even be possible to regenerate a degraded function using other processes or processors provided the surviving communication systems will support the communications necessary for the recovery and operation of the reconfigured system (3).

COMMUNICATING SEQUENTIAL PROCESSES (CSP)

The software design of distributed systems necessarily involves the design of a set of communicating sequential processes, involving aspects of concurrency. The methods for the design of centralised multiprocessing systems have been developed over the last eighteen years (4-6).

However, the techniques for the identification of critical sections of code, and the provision of mechanisms for enforcing mutual exclusion and synchronism essentially provide bottom-up design primitives. They are used extensively in the kernal of design methodologies such as MASCOT (7) and are hidden from the applications designer. Such monitor based techniques are unsuited to distributed systems design since a centralised facility is unavailable.

The development of concurrent programming languages, such as CSP and its derivatives, such as occam (8-9), in which message-passing synchronising inter-process communications are a primitive of the language, allows the high level design of distributed systems. The use of such constructs simplifies systems analysis and facilitates the design of distributed systems. The formal background of CSP also provides an mathematical basis for the analysis of the system behaviour and the design of fault tolerant methods.

A CLASSIFICATION OF INTER-PROCESS COMMUNICATIONS.

Interprocess communications may be classified (10) into one of three groups:

i) synchronous communications, where neither the sending nor receiving process is allowed to proceed beyond the communication point until its complementary process has also reached that point. This is found most notably within CSP and occam.

ii) asynchronous communications, where the process sending the message does not wait for acknowledgement, but the receiving process is not permitted to proceed beyond the communication point until the message has arrived (11).

iii) remote procedure call, where the process sending the message requires the receiving process to perform some specific function and respond before they both can proceed further. In essence it is an asynchronous communication followed by a synchronous communication. This form is found in ADA and elsewhere (12-13).

It is common to associate inter-process messages with the function they perform (10). Alarm messages have high priority on the interprocess medium; they are issued by one process and require immediate response by the receiving process. Command messages require a change of state or action to occur in due course; acknowledgement is a necessary requirement before the issuing process continues. Status messages are sent to notify other processes of information about the source task. No acknowledgement is required.

These message groups can be constructed using any of the above communication primitives. However, detailed study is required to ensure that the logical structure of the inter-process action (the transaction level) is not disrupted should the lower level communication primitives fail in a particular application (14). Single failure detection systems, such as time-outs, can only be applied at process level on a per-process basis and this does not necessarily provide protection at the

transaction level.

For example, failure in an asynchronous communication system can leave a process suspended indefinitely awaiting a communication, or may leave one process aware of a failure but unable to co-ordinate recovery action through the absence of a logical pairing between participating processes. The remote procedure call requires the object task to acknowledge when its action is complete (12); if the reply is lost then the source process is suspended indefinitely. Synchronous primitives, such as those in occam, preclude the protection of individual transactions since the language is specifically intended for deterministic system design only and outputs can not be used as guards on synchronous primitives. Hence, timeouts cannot be used in parallel with other processes to form a race condition, and so they cannot be placed on the send primitives of inter-process transactions. However, experimental languages, such as Pascal m (15), have attempted to overcome these deficiencies, but no general consensus nor formal method is as yet suitable. Although transaction level protection cannot be supplied directly, software fault tolerance methods can be applied to such systems using state based recovery techniques which may enclose complete inter-process communication transactions within the distributed recovery block or conversation (16).

SOFTWARE DESIGN TOOLS FOR DISTRIBUTED SYSTEMS

The design and synthesis of software for distributed systems requires the use of a design methodology and programming language which builds on the inherent parallel nature of such systems. Formal methods applied to the design of software for distributed processors lead to the identification of processes, capable of asynchronous execution, interacting with other processes by communications. The provision of constructs for sequence, variable assignment, selection and iteration, augmented by constructs which enable parallel execution, the use of synchronous communications for input/output, the provision of guarded processes and the formal inclusion of time are sufficient for the design of software for distributed systems (17-18).

The programming language occam, which is derived from the theory of communicating sequential processes provides a good notation in which to pursue the design of distributed systems. Occam produces concise, elegant and easily understood software. The mathematical axiomatic base of CSP permits algebraic analysis of software. In particular, correctness preserving transforms can be applied to it. It therefore offers the prospect that in the future such software may be formally verified.

The language is intended for use with both sequential and inherently parallel systems. The starting point in design using occam is the identification of the natural parallelism and the partitioning of the software into naturally occurring processes. Interaction between processes is solely by means of interprocess communication. Again this is a good match with a distributed system. The mapping of processes onto target processors occurs at a late stage in the design. This allows the designer to concentrate on the application function rather than implementation details.

HIGH RELIABILITY DISTRIBUTED SYSTEMS

For high reliability applications it is essential that failure modes be identified and measures taken to ensure that the system recognises when a fault occurs, constrains the scope for error propagation, and recovers to generate a safe response (19). A taxonomy of faults, ranging from sensor failure to software faults, can be drawn up with methods for their detection, and appropriate remedial action. The framework for recovery is relatively straightforward for centralised sequential systems, involving the use of process roll-back within recovery blocks and offering alternative processes (20), possibly within the confines of time-out watchdog timers (21). A fundamental assumption is that the framework itself is immune from faults.

However, the situation becomes more complex for decentralised systems since there is scope for error propagation by inter-process communication, which once initiated cannot be retracted. Methods are required which restrict the scope of error propagation between communicating processes, and which co-ordinate recovery amongst all processes participating in erroneous communications. The conversation scheme (20) is appropriate, and requires all participating processes to perform an acceptance test at pre-determined points in their processing. If any process is found to be in error at that test then all participating processes must perform co-ordinated recovery. No process is allowed to proceed beyond the acceptance test until all the other processes also pass the test.

The chief design problem is the proper placement of conversations. It is evident that the conversation scheme requires synchronism at, or following, the acceptance test to exchange the results of the test and, if necessary, coordinate process action. Current approaches to this problem have used the centralised concept of a monitor to implement acceptance tests. Unfortunately, this centralising feature of the monitor makes it unsuitable for distributed systems. An alternative approach developed by the authors (16) makes use of the synchronising properties of CSP/occam communications to design and implement distributed acceptance tests.

The crucial problem of conversation placement has received somewhat less attention. In effect the designer must identify the extent of process corruption and error migration through inter-process communication for all faults in the system. The objective of this fault effect analysis is the identification of a boundary or set of properly nested boundaries, which define known entry (recovery line) and exit (acceptance) states for the system. This allows the entry and exit state for each component process to be determined. Software must be designed to save recovery line entry states, and to implement and synchronise the acceptance tests on all processes in the conversation. Attempts to identify recovery lines and acceptance points dynamically are prone to progressive collapse (20).

An alternative approach (16) is to utilise the deterministic state properties of CSP/occam in the static design of conversations within the known state reachability space of the distributed system. This approach

offers considerable advantages and allows the use of design aids which automàtically generate sets of proper conversation boundaries within the system. It is then for the designer to choose the features of a design which he wishes to protect and the degree of software fault tolerance appropriate to a particular class of application.

The conversation scheme offers the most appropriate structure for recovering from unanticipated faults. The nature of the conversation scheme is that the acceptance test results (go/no-go) must be compared amongst the participating processes. The detection of an error during an acceptance test does not necessarily identify uniquely the fault; indeed the fault might lie in the interprocessor communication medium. Circumstances may arise where it is impossible to promulgate the result of the acceptance test to promote error recovery, perhaps due to a failure of the interprocess medium. In this cases the framework for recovery fails because the fault affects the recovery structure itself. Similar problems are inherent in any recovery structure and have been recognised for recovery block structures applied to sequential software (22). However, thay have not detracted significantly from the effectiveness of the technique.

A CONTROL EXAMPLE

The program fragment presented in Figures 1-4 is taken from a example program used to explore the problems involved in the design of software for distributed processes. It concerns the motion of a robot in each of three axes. It illustrates a number of points:

i) inherently local functions are modelled as processes, each capable of execution in parallel. Thus motion in each axis is programmed, and occurs, independently of motion in the other axes. Each process could be targetted onto separate processors at a late stage in the design.

ii) interprocess communications is the only form of interaction between the processes. This would take place over an interprocess communication medium. Thus each process is commanded to perform its activity, and signals when it has completed its activity.

iii) The software also contains a command process which initiates parallel commands and receives, as they occur and in whatever order they occur, the responses corresponding to the execution of those commands.

iv) The software contains a proper conversation boundary for the protection of the critical interprocess communication which governs the co-ordinated axial movement of the robot.

An equivalent fault-tolerant program written in a conventional language would be much more difficult to design, program and verify.

Figure 1

```
{{{ PROGRAM robot
... system parameters
CHAN request, return, motion[3], finished[3], stop[4]:
... process operator
... process motor
... process control
-- initiate processes
PAR
  PAR i = [0 FOR 3]
    motor(motion[i], finished[i], stop[i])
  control(request, return, stop[3])
  operator(request, return)
}}}
```

Figure 2

```
{{{ process motor
PROC motor (CHAN motion, finished, stopi)=
  VAR step, direction, going:
  SEQ
    going := TRUE
    WHILE going
      ALT
        stopi ? ANY
          going := FALSE
        motion ? step
          SEQ
            motion ? direction
--  move motor
            finished ! ANY:
}}}
```

Figure 3

```
{{{ process operator
PROC operator (CHAN send, receive)=
  VAR  x, y, z, run:
  SEQ
    run := TRUE
    WHILE run
      SEQ
        Screen ! 'i'
        Screen ! EndBuffer
        Keyboard ? x
        Screen ! x
        Screen ! EndBuffer
        Keyboard ? y
        Screen ! y
        Screen ! EndBuffer
        Keyboard ? z
        Screen ! z
        Screen ! EndBuffer
        x := x - '0'
        y := y - '0'
        z := z - '0'
        IF
          (x=0) AND (y=0) AND (z=0)
            SEQ
              Screen ! 'f'
              Screen ! EndBuffer
              PAR i = [0 FOR 4]
                stop[i] ! ANY
              run := FALSE
          TRUE
            SEQ
              send ! x
              send ! y
              send ! z
              receive ? ANY
              Screen ! 'm'
              Screen ! EndBuffer:
}}}
```

Figure 4

```
{{{ process control
PROC control (CHAN receive, send, stopi)=
  VAR xold, yold, zold, xnew, ynew, znew, count, step[3], direction[3],
      going:
  SEQ
    xold := 0
    yold := 0
    zold := 0
    going := TRUE
    WHILE going
      ALT
        stopi ? ANY
          going := FALSE
        receive ? xnew
          SEQ
            receive ? ynew
            receive ? znew
-- calculate distance and direction of each
-- motor. These can be calculated in parallel.
            PAR i = [0 FOR 3]
              SEQ
                motion[i] ! step[i]
                motion[i] ! direction[i]
            xold := xnew
            yold := ynew
            zold := znew
            count := 0
            WHILE count <> 3
              ALT i = [0 FOR 3]
                finished[i] ? ANY
                  count := count + 1
            send ! ANY:
}}}
```

CONCLUSION

The design of distributed computer systems requires specific methodologies and techniques if high reliability is to be achieved. Systematic analysis of the specification is required to identify and to exploit the parallelism inherent in the application. This must be complemented by design methods and programming languages suited to a highly parallel computing environment. Careful analysis of the communications is required to co-ordinate the processing. and to ensure that proper conversations are produced for recovery activity.

REFERENCES

1. Daniels, B.K., Reliability Engineering, 4, 1983, 199.

2. Boehm, B.W., Software Engineering Economics, 1981, Prentice-Hall.

3. Momen, S.E.M., Holding, D.J., Proc Int Conf on Control and its applications, I.E.E., 1981, 291.

4. Brinch Hansen, P., Operating System Principles, 1973, Prentice-Hall.

5. Dijsktra, E.W., Co-operating sequential processes, 1968, in Programming Languages, ed. Genuys, F., Academic Press.

6. Hoare, C.A.R., Comm ACM, 17, 1974, 549.

7. Simpson, H.R., MASCOT 3, I.E.E. Coll on MASCOT, 1984.

8. Hoare, C.A.R., Comm ACM, 21, 1978, 666.

9. May, D., Sigplan Notices 18, 1983, 67.

10. Kramer, J., Magee, J., Sloman, M., Proc 2nd Int Conf on Distributed Computer Systems, 1981, 404.

11. Liskov, B., Proc 7th ACM SIGOPS Symp on Operating System Principles, 1979, 33.

12. US Department of Defense. ADA Reference Manual. 1980.

13. Kramer, J., Magee, J., Sloman, M., Lister, A., IEE Proc Pt E, 130, 1983, 1.

14. Holding, D.J., Carpenter,G.F., Tyrrell,A.M., Proc 6th IEEE/Eurel Conf on Computers in communications and control, 1984, 235.

15. Bornat, R., A protocol for generalised occam, Research report 348. Queen Mary College, 1984.

16. Tyrrell, A.M., Holding, D.J., submitted to I.E.E.E. Trans on Software Engineering, 1986.

17. Linger, R.C., Mills, H.D., Witt, R.C., Structured programming, theory and practice, Addison Wesley, 1979.

18. Hoare, C.A.R., Communicating Sequential Processes. Prentice-Hall; 1985.

19. Anderson, T., Lee, P.A., Fault Tolerance: Principles and Practice, 1981, Prentice Hall.

20. Randell, B.R., I.E.E.E. Trans on Software Engineering, SE-1, 1975, 220.

21. Kim, K.J., I.E.E.E. Trans on Software Engineering, SE-8, 1982, 189.

22. Jackson, P.R., White, B.A., The application of fault tolerant techniques to a real-time system., Safety of Computer Control Systems, Pergamon, 1983.

Design of Reliable Software in Distributed Systems Using the Conversation Scheme

ANDREW M. TYRRELL, MEMBER, IEEE, AND DAVID J. HOLDING

Abstract—A fundamental problem in the design of error detection and recovery mechanisms for networks of cooperating asynchronous processes is the prevention of error propagation through process interaction. The recovery procedure must be a cooperative effort involving all the interactive processes and may be limited to bounded parts of the system by the conversation mechanism proposed by Randell.

This paper examines the problems of error detection and recovery in a number of concurrent processes expressed as a set of communicating sequential processes (C.S.P). A method is proposed which uses a Petri net model to identify formally both the state and the state reachability tree of a distributed system. These are used to define systematically the boundaries of a conversation including the recovery and test lines which are essential parts of the fault-tolerant mechanism.

The method can be used as a design tool to determine a single conversation or a set of properly nested conversations. The technique can be used to identify the full set of processes enclosed within a particular conversation, or to design a conversation which will protect a specific functional aspect of a distributed system.

The techniques described in this paper are implemented using the occam programming language, which is derived from C.S.P. The application of this method is shown by a control example.

Index Terms—Communicating sequential processes, concurrent processes, conversation, distributed systems, fault-tolerant software, occam, Petri nets, recovery block.

I. INTRODUCTION

A FUNDAMENTAL problem in the development of fault-tolerant distributed systems is the design of error detection and recovery procedures for the distributed system [1].

This paper addresses the problem of error detection and recovery in distributed systems which consist of a cooperating set of asynchronous processes. These systems can be modeled as a set of communicating sequential processes using the C.S.P. notation [2]. In such a system, error detection and recovery must be a cooperative effort involving all interacting processes [3]. If the recovery operation is to be limited in extent, rather than global, then it is necessary to identify boundaries within the state space of the network of processes which can be used for error detection and recovery [4].

The conversation mechanism proposed by Randell [5] uses such a boundary as a recovery block for a general set of distributed processes. A number of mechanisms have been proposed for implementing this type of system [6]–[8]. However, these methods do not address the problem of determining the boundary of the conversation and their implementation requires language extensions or involves the use of centralized techniques such as monitors [9].

Fault-tolerant mechanisms such as the recovery block and the conversation implement recovery by backtracking operations which restore the system to a previous state. If the state involves temporal attributes, then the backtracking operation will also return the system to the virtual time at which the previous state was instantiated, and the problems of time warp must be accommodated in the alternative path of the recovery block.

The definition of the state of the system and the assignment of state identifiers are fundamental parts of the design procedure [10]. This is not a major problem in sequential systems in which the state of the active process can be ascertained and saved at appropriate points in the program in order to implement the recovery block technique [5]. The design problem is more complex in distributed systems because the set of concurrent processes may operate asynchronously until brought into synchronism by interprocess communications. Since the state of each process can be independent of the state of other processes, it is not possible to determine *a priori* the particular sequence of states which will be instantiated during operation or execution of the system. Thus, the conversation boundary must be identified dynamically or must be independent of the sequence of occurrence of the independent states.

A number of papers have addressed the problem of the dynamic identification of conversation boundaries in distributed systems [10]–[13]. However, dynamic recovery techniques may exhibit the domino effect [5] and this limits their usefulness.

This paper proposes a method for the *a priori* design of conversations for the class of distributed system which can be expressed as a set of communicating sequential processes. It is shown that the problem of defining the system state can be resolved using Petri nets [14] to identify the state and state reachability tree of the system. The dynamic behavior of the system can be characterized by a state-change table derived from the state reachability tree. It is shown that a conversation can be generated by defining a closed boundary on any branch of the state-

Manuscript received August 30, 1985; revised February 28, 1986.

A. M. Tyrrell is with the Department of Electrical, Electronic, and Systems Engineering, Coventry (Lanchester) Polytechnic, Coventry CV1 5FB, England.

D. J. Holding is with the Department of Electrical and Electronic Engineering and Applied Physics, Aston University, Birmingham B4 7ET, England.

IEEE Log Number 8609737.

nge table. The boundary encloses all processes which party to the conversation. The associated acceptance and recovery states can be identified at the intersecs of the boundary and the state-change table.

onversations are generally designed to provide fault rance for specific functional parts of a system. In this er the functional boundaries of a system are mapped o the Petri net model of the system. The functional ibutes of the system states are then used to reduce the em state-change table to a table of those states which changed by interfunction actions. It is shown that this e can be used either to design conversations which tect a particular functional aspect of a system or to ermine those functions which would be effected by a icular conversation.

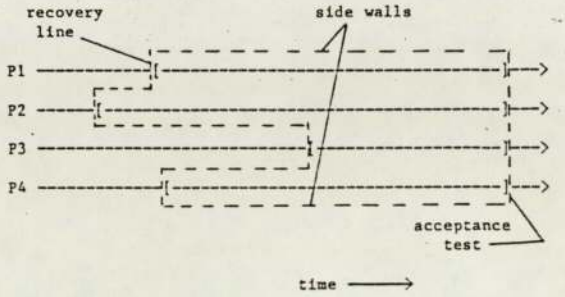
n this paper the distributed systems are expressed in concurrent programming language occam [15] which lerived from the C.S.P. notation. Occam is used for design and implementation of the conversation mechsm for error detection and recovery. These methods illustrated by an example which involves the control a three-axis robot using five concurrent processes.

II. Recovery in Concurrent Systems

The recovery block technique [5] for error detection and overy in single process sequential systems cannot be ended directly to networks of communicating sequenprocesses. Error detection mechanisms for distributed tems must take into account the possibility that errors l promulgate through process interaction and any reery scheme must involve all processes which interact hin the space of the recovery mechanism. The converon [5] uses a coordinated set of recovery blocks to plement the distributed error detection and recovery chanisms.

The boundary of a conversation consists of a recovery e, a test line, and two side walls. The boundary enses the set of communicating (interacting) processes ich are party to the conversation. The recovery line is part of the boundary which defines the start of the nversation. It consists of a coordinated set of states (reery points) for the interacting processes. At the start a conversation, the state of each entry process is stored use during recovery. The entry to a conversation need be a synchronous event.

The test line is a coordinated set of acceptance tests for set of interacting processes. Each test line process is uired to pass an acceptance test. A conversation is sucsful only if all test line processes pass their acceptance ts. Processes must exit from a conversation synchrously. If any acceptance test is failed, recovery is ieved by rolling back the conversation to the recovery e, restoring the process state to that on entry to the consation, and executing the alternate blocks. Thus, proses in the conversation cooperate in error detection. e side walls of the conversation prohibit the passing of ormation to processes not involved in the conversation (prevent information smuggling). A conversation consisting of four processes is shown below.



III. Distributed System Model

In a distributed system modeled using the concurrent programming language occam, each process will proceed asynchronously until forced into synchronism by interprocess action. In such a model, process synchronization and information exchange are unified in the occam interprocess communication primitives.

A. Functional Boundaries

The functional boundaries of a distributed system can be mapped onto the C.S.P. or occam model of the system. Thus each process in the model can be associated with a particular functional partition or boundary and can be given a function-identifier attribute. These attributes will be used to discriminate between intrafunctional and interfunctional communications.

B. Robot Description

Consider the problem of controlling the position of a three-axis robot [16]. Let the proposed control system consist of five functional processes: the operator interface, the controller, and three axial motor controllers.

The "operator" process inputs the coordinates of the desired position of the robot, and checks the input data for reserved values or control overrides (such as the final (stop) position 0, 0, 0) and outputs control values to the "control" process. Process "control" accepts inputs from process "operator," calculates the relative direction and distance of the new coordinates, and outputs the computed values of derived axial movement to each of the "motor" processes. Each of the three axial "motor" processes inputs axial values of direction and distance from process "control" and moves the robot to the desired axial position.

The proposed solution to the control problem is described in the occam program listed in Fig. 1.

C. Petri Net Models

Considerable research has been done on Petri nets [14], [17]–[19] and a formal definition for the basic structure of a Petri net has been published [14]. Petri nets are commonly used to model asynchronous and synchronous logic systems. They have also been used to model the primitives and constructs of sequential software [20]. The GMB [21] technique can also be used for modeling such systems [22].

```
Robot Example.OCC
-- Occam program for 3-Axis Robot Arm Controller.
-- Declaration of inter-process channels.
CHAN request,return,motion[3],finished[3],stop[4],go[4] :
-- Declaration of process 'operator'.
PROC operator (CHAN send,receive) =
  VAR x,y,z,run :
  SEQ
    run := TRUE
    WHILE run
    SEQ
...   input x,y,z from keyboard.                --(t1)
--
      send ! x          --send to control process. (t2)
      send ! y
      send ! z
--
      receive ? ANY              --motors moved. (t3)
--
      IF
        (x=0)AND(y=0)AND(z=0)   --check for finish. (t4)
          SEQ
            PAR i = [0 FOR 4]
              stop[i] ! ANY      --finish. (t5,t16,t22,t28)
            run := FALSE         -- (t6)
        TRUE                     -- (t7)
          PAR i = [0 FOR 4]
            go[i] ! ANY :        --continue. (t8,t18,t24,t30)
--
-- Declaration of process 'motor'.
PROC motor (CHAN motion,finished,stopi,goi) =
  VAR step,direction,going :
  SEQ
    going := TRUE
    WHILE going
      SEQ
        motion ? step          --get from control. (t10,t19,t25)
        motion ? direction
--
...     move motor             --(t15,t20,t26)
--
        finished ! ANY         --(t13,t21,t27)
--
        ALT
          stopi ? ANY          --finish.  (t16,t22,t28)
            going := FALSE     --(t17,t23,t29)
          goi ? ANY            --continue.  (t18,t24,t30)
            SKIP :
--
```

(a)

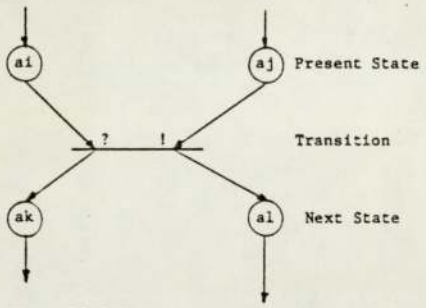
```
-- Declaration of process 'control'.
PROC control (CHAN receive,send,stopi,goi) =
  VAR xold,yold,zold,xnew,ynew,znew,
  count,step[3],direction[3],going :
  SEQ
... initialise xold,yold,zold
    going := TRUE
    WHILE going
      SEQ
        receive ? xnew  --input from operator. (t2)
        receive ? ynew
        receive ? znew
--
...     caiculate distance and direction  --(t9)
--
        PAR i = [0 FOR 3]
          SEQ
            motion[i] ! step[i]      --send to each motor.(t10,t19,t25)
            motion[i] ! direction[i]
--
...     update xold,yold,zold        --(t11,t12)
--
        count := 0
        WHILE count <> 3
          ALT i = [0 FOR 3]
            finished[i] ? ANY        --check motors moved.(t13,t21,t27)
              count := count + 1
        send ! ANY                   --(t3)
--
        ALT
          stopi ? ANY                --finish.(t5)
            going := FALSE           --(t14)
          goi ? ANY                  --continue.  (t8)
            SKIP :
--
-- main program.
PAR
  PAR i = [0 FOR 3]
    motor(motion[i],finished[i],stop[i],go[i])
  control(request,return,stop[3],go[3])
  operator(request,return)
```

(b)

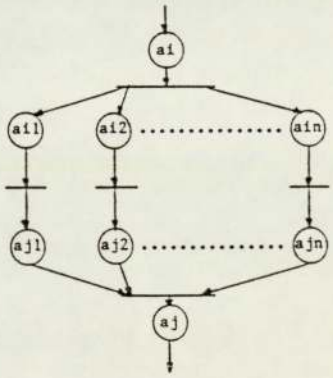
Fig. 1. Occam program for three-axis robot arm controller.

Concurrent programming languages can be modeled using Petri nets (or GMB) if models are developed for the primitives and constructs in the concurrent languages. In the following it is shown that the concurrent programming language occam can be modeled using Petri nets. This enables Petri net models to be derived for distributed systems described by occam programs.

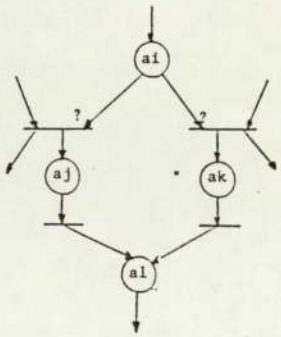
1) Communications and Synchronization: Occam processes communicate by message passing. This also provides interprocess synchronism because communications only take place when both the input (?) and output (!) processes are ready. This primitive process can be modeled by the Petri net transition shown below.



2) PAR (Parallel Construct): In the parallel construct, PAR, all actions are initiated simultaneously. The construct does not terminate until all actions have terminated. This can be modeled by the Petri net shown below.



3) ALT (Alternative Construct): The alternative construct chooses one of its components for execution. Each component process has a guard which is an input (?). The process whose guard is satisfied earliest is executed. If more than one guard is satisfied the choice as to which alternative is taken is defined as being arbitrary. This construct can be modeled by the Petri net shown below.



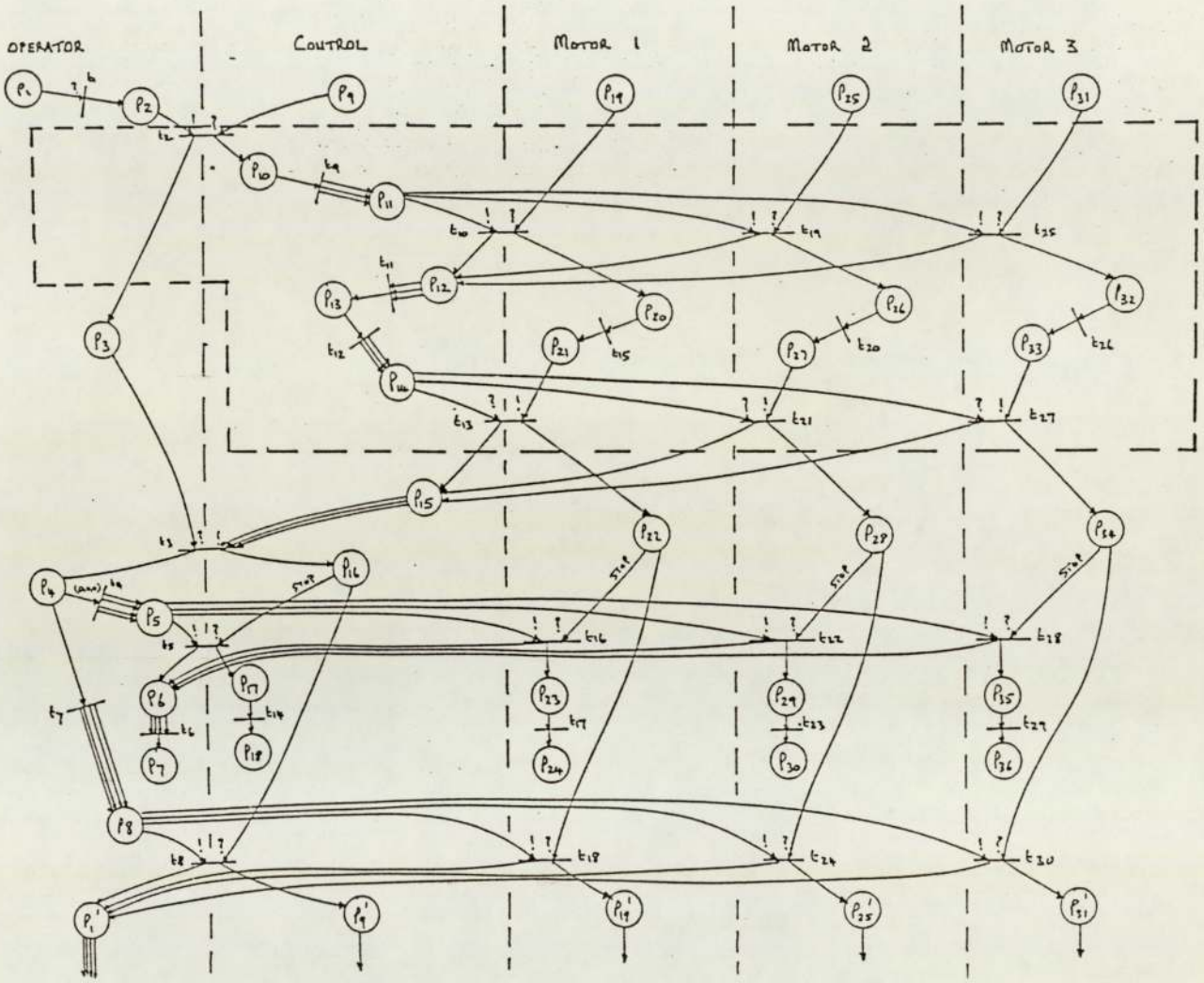


Fig. 2. Petri net graph of robot arm controller.

4) Functional Attributes: The functional boundaries of distributed systems can also be mapped onto Petri net models. The transitions (ti) and the states (pi) of the Petri net can then be associated with specific functions and assigned the attributes of the function identifier or process (PROCi).

$$\text{PROC}i = \{ti, pi\} \quad \text{where} \quad ti = \{ta \,..\, tn\}$$
$$\text{and} \quad pi = \{pa \,..\, pk\}$$

D. Petri Net Model of Robot Example

The robot arm control program of Fig. 1(a), (b), which consists of five concurrent processes, can be translated into a Petri net graph using the transformations described above. The complete Petri net graph for the robot program of Fig. 1 is shown in Fig. 2 and is partitioned into five functional processes which correspond to the actual processes in the program. The repetitive construct in each functional process gives rise to cyclic structures in the Petri net graph which serve to bound the graph. The closure of the cyclic loops is signified in Fig. 2 by the primes on the states identifiers ($p1', p9', p19', p25', p31'$). The functional attributes of the system can be mapped onto the Petri net and the attributes of each state and transition are listed below.

$$\text{PROCoperator} = \{t1, t2, t3, t4, t5, t6, t7, t8, t16, t22, t28, t18, t24, t30, p1, p2, p3, p4, p5, p6, p7, p8\}$$

$$\text{PROCcontrol} = \{t2, t9, t10, t19, t25, t11, t12, t13, t21, t27, t3, t5, t14, t8, p9, p10, p11, p12, p13, p14, p15, p16, p17, p18\}$$

$$\text{PROCmotor1} = \{t10, t15, t13, t16, t17, t18, p19, p20, p21, p22, p23, p24\}$$

$$\text{PROCmotor2} = \{t19, t20, t21, t22, t23, t24, p25, p26, p27, p28, p29, p30\}$$

$$\text{PROCmotor3} = \{t25, t26, t27, t28, t29, t30, p31, p32, p33, p34, p35, p36\}$$

E. System State and Reachability Tree

If M is the state (or marking) of the Petri net, with state variables $pa \,..\, pz$, such that $pk \in M$, then for a given transition tj the next state function $\delta(\mu, tj) = \mu'$ defines the

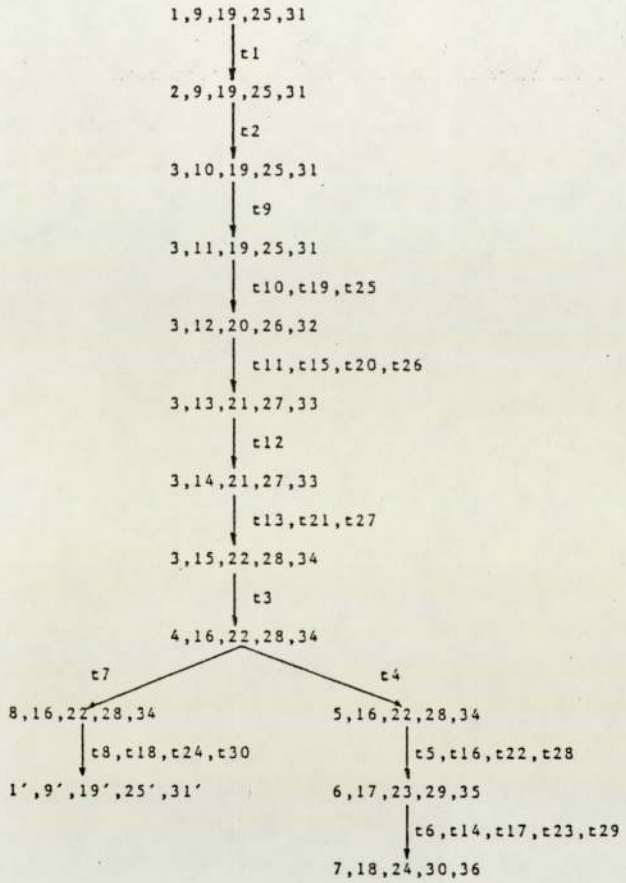


Fig. 3. Reachability tree of Fig. 2.

transition from present state $\mu = \{pa\,..\,pm\}$ to the next state $\mu' = \{pc\,..\,pn\}$. The next state function can be determined for each transition in the Petri net. For example, transition $t2$ of Fig. 2 corresponds to an occam communcation primitive and defines the transition from the present state $\{p2, p9, p19, p25, p31\}$ to the next state $\{p3, p10, p19, p25, p31\}$.

The reachability tree of a Petri net model can be formed from the set of all next state functions. The set of all states forms the state space of the system and is known as the reachability set $R(C, \mu)$. The reachability tree defines the system behaviour within the state space of the system. It therefore forms a good basis for the placement of conversations and the identification of the associated test and recovery lines.

The reachability tree for the Petri net graph of the robot controller is shown in Fig. 3. The tree has two branches, the bifurcation being caused by the conditional clause in the process "operator." This detects whether the robot should be operated normally or moved to the reserved portion (0, 0, 0) and shut down.

IV. Conversation Placement

A. State Transitions and Interprocess Communication

The system dynamics are characterized by the evolution of the system states through a sequence of state transitions. This can be defined by a state-change table which lists the state changes for each transition in the reachability tree. The elements of the state-change table can be identified by taking the relative complements of the present state μ and the next state μ' for each transition tj:

$$\mu - \mu' = \{pe\,..\,ps\} = Ij$$

$$\mu' - \mu = \{pg\,..\,pt\} = Ej$$

The sets I and E represent, respectively, the subset of the initial states which are terminated by the transition and the subset of the final states which are created by the transition. The state-change table for the robot controller can be derived from the reachability tree (Fig. 3) and is shown in Table I.

TABLE I
STATE CHANGE TABLE OF FIG. 3

Transitions	I	E
t1	1	2
t2	2,9	3,10
t9	10	11
t10	11,19	12,20
t19	11,25	12,26
t25	11,31	12,32
t11	12	13
t15	20	21
t20	26	27
t26	32	33
t12	13	14
t13	14,21	15,22
t21	14,27	15,28
t27	14,33	15,35
t3	3,15	4,16

t7	4	8
t8	8,16	1',9'
t18	8,22	1',19'
t24	8,28	1',25'
t30	8,34	1',31'

t4	4	5
t5	5,16	6,17
t16	5,22	6,23
t22	5,28	6,29
t28	5,34	6,35
t6	6	7
t14	17	18
t17	23	24
t23	29	30
t29	35	36

B. Identification of Conversations

A conversation limits the extent of error propagation in a distributed system. Conversations can be constructed by generating systematically the entry and exit lines of the conversation such that no process interaction takes place through the side walls of the conversation. Such a boundary will contain all processes which participate in the conversation.

The reachability tree defines all process interactions: no teractions take place between different branches of the ee. The proposed method uses these properties of the achability tree to form boundaries within the state space the system. Any two transitions on the same branch of e reachability tree can be considered to form a boundary partition enclosing part of the branch. The partition can considered to be the boundary of a conversation and n be mapped onto (or defined within) the state-change ble. For any such partition of the state-change table, o sets S and F, can be formed from the union of all esent and next states within the partition boundary.

$$S = \{I1\ U\ I2\ U\ldots In\}$$

$$F = \{E1\ U\ E2\ U\ldots En\}$$

he relative complements of these sets can be formed into o sets, J and K, which can be considered to be the entry d exit states of the conversation.

$$S - F = \{p1\ldots pn\} = K$$

$$F - S = \{pr\ldots py\} = J$$

If conversation boundaries overlap it is essential that ese conversations should be properly nested for the re-very mechanism to work correctly [5]. The properties the reachability tree ensure that a number of conver-tions will be properly nested.

. *Interfunctional Communications*

Interfunctional communications can be identified by ex-nining the functional attributes of the elements of the ate-change table. Let the element corresponding to tran-tion tj contain $p1$ and $p2$:

here $\quad p1, p2 \in Ij$ or $p1, p2 \in Ej$

nd $\quad p1 \in \text{PROC}q;\quad p2 \in \text{PROC}r$

en the transition is an interfunctional communication if $\neq r$. When $q = r$ the transition can be classified as an trafunction action.

The state-change table may be reduced to a communi-tion state-change table consisting of only interfunction ansitions by removing all intrafunctional transitions and rming equivalent relationships between states created by trafunctional actions (since these form local states be-veen interfunctional transitions). This table will be nown as the communication state-change table.

The state-change table, Table I, can be reduced to a mmunication state-change table. This table can be fur-er reduced as shown in Table II by grouping together l transitions corresponding to replicated ALT state-ents. For example, transitions $t10$, $t19$, $t25$ can be mbined to form a single element in the communication ate-change table, because all three ALT statements must e before the replicated ALT process can terminate.

The set of functional processes which are party to a nversation can be identified by the functional attributes the entry and exit states, K and J. These functional processes will be involved in the error detection and recovery procedures.

TABLE II
COMMS. STATE TABLE OF TABLE I

t2	2,9	3,11
t10,t19,t25	11,19,25,31	14,21,27,33
t13,t21,t27	14,21,27,33	15,22,28,34
t3	3,15	4,16
EITHER		
t8,t18,t24,t20	4,16,22,28,34	1',9',19',25',31'
OR		
t5,t16,t22,t28	4,16,22,28,34	7,18,24,30,36

TABLE III
PARTITION OF TABLE II

t2	2,9	3,11
t10,t19,t25	11,19,25,31	14,21,27,33
t13,t21,t27	14,21,27,33	15,22,28,34

D. *Design of Conversations*

The design problem usually involves protecting a particular part or function of the system. The states and transitions associated with this function can be identified through their functional attributes. Similarly, it is possible to identify the corresponding elements in the communication state-change table. The functions may be protected by specifying a boundary which encloses the complete set of identified states (and a minimal set of other states), provided all such states lie along the same branch of the reachability tree. The boundary can then be used to identify the test and recovery lines as described above. In addition to protecting a particular function, this technique identifies all processes within a particular conversation and all functions which are party to the conversation.

Consider the robot control example whose communication state-change is shown in Table II. Let the main conversation protect the "control" process from the point at which new coordinates are input (state 9) to the point at which all axial control "motor" processes have reported correct execution of the axial movement commands output by the "control" process (state 15). Therefore, the communication state-change table can be partitioned on the main branch to enclose state 9 as the recovery line and state 15 as the test line as shown in Table III. The corresponding initial and final states of the conversation S and F, and the recovery line states and test line states K and J can be determined as follows:

$$S = \{2, 9, 11, 19, 25, 31, 14, 21, 27, 33\}$$

$$F = \{3, 11, 14, 21, 27, 33, 15, 22, 28, 34\}$$

$$K = S - F = \{2, 9, 19, 25, 31\}$$

$$J = F - S = \{3, 15, 22, 28, 34\}$$

Examination of K and J shows that the functional attributes of the states involved in the conversation are:

$$9, 15 \in \text{PROC control}$$

$$2, 3 \in \text{PROC operator}$$

$$19, 22 \in \text{PROC motor1}$$

$$25, 28 \in \text{PROC motor2}$$

$$31, 34 \in \text{PROC motor3}$$

The boundary of this conversation is as shown by the dotted line on the Petri net graph (Fig. 2).

This design technique may also be used as a structuring tool by a designer who wishes to protect a particular part of a concurrent program. The designer would simply specify the parts of the program which are to be protected and identify the associated states and transitions. A proper conversation boundary could then be generated using the communications state-change table to enclose these states (and a minimum set of other states). The design procedure would then continue as above.

E. Implementation

The conversation scheme can be implemented using the concurrent language occam. The constructs available within this language facilitate the design process. For example, test lines can be implemented using ALT constructs. Similarly, occam communication channels (each of which link two named processes only) can be used to remove the problem of information smuggling by ensuring that no channel belongs to a process outside the conversation.

The conversation consists of its constituent processes and a conversation control process which acts as a test line coordinator for the conversation. When a conversation is started, a nominated member of the set of entry processes initializes the conversation coordinator. The coordinator exists for the duration of the conversation.

Each constituent process, when complete, executes a local acceptance test and enters an exit process. The result of these acceptance tests are reported to the test line process. The test line process collects the results of all local acceptance tests and determines whether the conversation has succeeded. If all local acceptance tests are successful the test line process notifies all exit processes in the conversation and the conversation is terminated. If one or more of the acceptance tests has failed the test line process notifies all exit processes that recovery roll back is to be executed.

The test line process is implemented using an ALT construct which receives notification of the results of local acceptance tests. The acceptance process does not therefore assume any particular order for the termination of the constituent process; nor does it impose any timing constraints on the systems performance.

V. Conclusions

This paper has considered some of the fundamental problems of designing robust software for distributed control systems. It has specifically addressed the problem of specifying and designing error detection and recovery mechanisms for a class of distributed systems. A method was described for the systematic identification of conversation boundaries.

The paper formalized the definition of system state and reachability by using Petri net techniques. The properties of the state reachability tree were then exploited in the development of a method for the design of proper conversations. The functional attributes of the system were used to identify conversations which would protect a particular part of a system (the conversation placement problem). The conversations designed using this method automatically enclose all processes which are party to the conversation.

The design method reduced the complexity of the problem by systematically reducing design considerations to only those system states which are changed through interfunctional actions. These states provided the minimum set required for the design procedure and the identification of the recovery and test lines. The use of the technique has been demonstrated by example.

References

[1] B. Randell, P. A. Lee, and P. C. Treleaven, "Reliability issues in computing," *Comput. Surveys*, vol. 10, no. 2, pp. 123–165, June 1978.

[2] C. A. R. Hoare, "Communicating sequential processes," *Commun. ACM*, vol. 21, no. 8, pp. 666–677, 1978.

[3] T. Anderson and P. A. Lee, *Fault Tolerance Principles and Practice*. Englewood Cliffs, NJ: Prentice-Hall, 1981.

[4] R. H. Campbell, T. Anderson, and B. Randell, "Practical fault tolerant software for asynchronous systems," in *IFAC Safecomp '83*, Cambridge, England, Aug. 1983, pp. 59–65.

[5] B. Randell, "System structure for software fault tolerance," *IEEE Trans. Software Eng.*, vol. SE-1, pp. 220–232, June 1975.

[6] K. H. Kim, "Approaches to mechanisation of the conversation scheme based on monitors," *IEEE Trans. Software Eng.*, vol. SE-8, pp. 189–197, May 1982.

[7] D. L. Russell and M. J. Tiedeman, "Multiprocess recovery using conversations," in *Proc. FTC-9*, 1979, pp. 106–109.

[8] T. Anderson and J. C. Knight, "A framework for software fault tolerance in real time systems," *IEEE Trans. Software Eng.*, vol. SE-9, pp. 355–364, May 1983.

[9] C. A. R. Hoare, "Monitors: An operating system structuring concept," *Commun. ACM*, vol. 17, no. 10, pp. 549–557, Oct. 1974.

[10] P. M. Merlin and B. Randell, "State restoration in distributed systems," in *Dig. Papers FTCS—8: Eighth Annu. Int. Symp. Fault Tolerant Comput.*, Toulouse, France, 1978, pp. 129–134.

[11] K. M. Kim, "An approach to programmer-transparant coordination of recovering parallel processes and its efficient implementation rules," in *Proc. Int. Conf. Parallel Processing*, 1978, pp. 58–68.

[12] G. Barigazzi and L. Stringini, "Application transparent setting of recovery points," in *Proc. 13th Int. Symp. Fault Tolerant Comput.*, 1983, pp. 48–55.

[13] D. L. Russell, "State restoration in systems of communicating processes," *IEEE Trans. Software Eng.*, vol. SE-6, pp. 183–194, Mar. 1980.

[14] J. L. Peterson, *Petri Net Theory and the Modeling of Systems*. Englewood Cliffs, NJ: Prentice-Hall, 1981.

[15] D. May, "Occam," *Sigplan Notices*, vol. 18, no. 4, pp. 69–79, 1983.

[16] J. M. Kerridge and D. Simpson, "Three solutions for a robot arm controller using Pascal-Plus, occam and Edison," *Software—Practice and Experience*, vol. 14, pp. 3–15, 1984.

[17] J. L. Peterson, "An introduction to Petri nets," in *Proc. Nat. Electron. Conf.*, vol. 32, 1978, pp. 144–148.

[18] ——, "Petri nets," *Comput. Surveys*, vol. 9, no. 3, pp. 223–252, Sept. 1977.

[19] J. Dennis, "Concurrency in software systems," in *Advanced Course in Software Engineering*, X. Eauer, Ed. 1973, pp. 111–127.

[20] L. J. Mekly and S. S. Yau, "Software design representation using abstract process networks," *IEEE Trans. Software Eng.*, vol. SE-6, pp. 420–434, Sept. 1980.

J. W. Winchester and G. Estrin, "Requirements definition and its interface to the SARA design methodology for computer-based systems," in *AFIPS Conf. Proc. Nat. Comput. Conf.*, vol. 51, 1982, pp. 369-379.

W. Ruggiero, G. Estrin, R. Fenchel, R. Razouk, D. Schwabe, and M. Vernon, "Analysis of data flow models using the SARA graph model of behavior," in *AFIPS Conf. Proc. Nat. Comput. Conf.*, vol. 48, 1979, pp. 975-988.

Andrew M. Tyrrell (S'82-M'85) received the Honours degree in electronic engineering from Bolton Institute of Technology, Bolton, England, in 1982. From 1982 to 1985 he was a research student at Aston University, Birmingham, England, working toward the Ph.D. degree in the design of fault-tolerant distributed systems.

Since September 1985 he has been employed as a Lecturer at Coventry (Lanchester) Polytechnic, Coventry, England. His research interests are in fault-tolerant software, distributed processor systems, and distributed topologies for image processing applications.

David J. Holding was born in Nottingham, England, on August 12, 1946. He received the B.Sc. (Eng.) and Ph.D. degrees from the University of London, London, England, in 1968 and 1974, respectively.

From 1964 to 1969 he was employed and sponsored by the East Midlands Electricity Board. From 1969 to 1980 he was a Lecturer in Electrical and Electronic Engineering at Queen Mary College, University of London. The period 1977-1978 was spent on leave at the Warren Spring Laboratory, Department of Industry, England. Since 1980 he has been at Aston University, Birmingham, England, first in the Aston Microprocessor Unit as Head of Unit and Senior Lecturer, and then in the Department of Electrical and Electronic Engineering as a Senior Lecturer. His main research interests are in the design of decentralized and distributed processing systems, fault-tolerant systems, and control applications.

Dr. Holding is a member of the Institute of Electrical Engineers and the Institute of Measurements and Control of the U.K.

References.

[1] P.M. Melliar-Smith and B. Randell, Software Reliability: The role of programmed exception handling, Proc. Conf. Language Design for Reliable Software, SIGPLAN Notices, Vol. 12, May 1977, pp 95 - 100.

[2] T. Anderson, P.A. Lee and S.K.Shrivastava, A Model of Recoverability in Multilevel Systems, IEEE Trans. on Software Engineering, Vol. SE - 4, No. 6, Nov 1978, pp 486 - 494.

[3] W.H. Pierce, Fault - Tolerant Computer Design, New York, Academic Press, 1965.

[4] B. Randell, System Structure for Software Fault Tolerance, IEEE Trans. on Software Engineering, Vol. SE -1, No. 2, June 1975, pp 220 - 232.

[5] Software Tools for Application to Large Real Time Systems, Department of Trade and Industry, 1984.

[6] Z.C. Chen and C.A.R. Hoare, Partial Correctness of Communicating Processes and Protocols, Research Monograph PRG - 20, Oxford University Computing Laboratory, May 1981.

[7] A. Avizienis, Fault Tolerant Systems, IEEE Trans. on Computing, Vol. C - 25, No. 12, Dec. 1976, pp 1304 - 1312.

[8] L. Chen and A. Avizienis, N-version Programming: A Fault Tolerance Approach to Reliability of Software Operation, Digest of 8th Annual International Conference on Fault Tolerant Computing, Toulouse, June 1978, pp 3 - 9.

[9] H. Hecht, Fault Tolerant Software, IEEE Trans. on Reliability, Vol. R - 28, No. 3, Aug. 1979, pp 227 - 232.

[10] B. Randell, P.A. Lee and P.C. Treleaven, Reliability Issues in Computing System Design, Computing Surveys, Vol. 10, No. 2, June 1978, pp 123 - 165.

[11] J.J Horning, H.C. Lauer, P.M. Melliar-Smith and B. Randell, A Program Structure for Error Detection and Recovery, Proc. Conf. Operating Systems: Theoretical and Practical Aspects, April 1974, pp 177 - 193.

[12] R.H. Campbell, T. Anderson and B. Randell, Practical Fault Tolerant Software for Asynchronous Systems, Proc. Conf. IFAC Safecomp '83, Cambridge 1983, pp 59 - 65.

[13] A. Avizienis and L. Chen, On the Implementation of N-Version Programming for Software Fault Tolerance During Program Execution, Proc. COMPSAC 77, Nov. 1977, pp 149 - 155.

[14] E. Yourdon and L. Constantine, Structured Design,

Englewood Cliffs, NJ: Prentice Hall, 1979.

[15] J. Welsh and M. McKeag, Structured System Programming, Prentice Hall, 1980.

[16] D.B. Lomet, Process Structuring, Synchronization and Recovery using Atomic Actions, Sigplan Notices, Vol. 12, No. 3, March 1977, pp 128 - 137.

[17] T. Anderson and P.A. Lee, Fault Tolerance, Principles and Practice, Englewood Cliffs, NJ: Prentice Hall, 1981.

[18] I. Gertner and R.L. Gordon, Experiences with Exception Handling in Distributed Systems, 2nd Symposium on Reliability in Distributed Software and Database Systems, Pittsburgh 1982, pp 144 - 149.

[19] E. Dijkstra, Solution of a Problem in Concurrent Program Control, Comm. ACM, Vol. 8, No. 9, Sept. 1965, pp 569.

[20] D.J. Holding, G.F. Carpenter and A.M. Tyrrell, Aspects of Software Engineering for Systems with Safety Implications, Eurocon ´84, Brighton 1984, pp 235 - 239.

[21] J.L. Peterson, Petri Nets, Computing Surveys, Vol. 9, No. 3, Sept. 1977, pp 223 - 252.

[22] Occam Programming Manual, Prentice Hall, 1984.

[23] A.M. Tyrrell and D.J. Holding, Design of Reliable Software in Distributed Systems using the Conversation Scheme, IEEE Trans. on Software Engineering, Vol. SE - 12, No. 9, Sept. 1986, pp 921 - 928.

[24] A.M. Tyrrell and D.J. Holding, The Design of Communication Software for Distributed Multivariable Control Systems, Symposium on Application of Multivariable System Techniques, Plymouth 1984, pp 191 - 204.

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

[26] P.M. Merlin and B. Randell, Consistent State Restoration in Distributed Systems, Digest of Papers FTCS-8: 8th Annual Int. Symposium on Fault Tolerant Computing, Toulouse, June 1978, pp 129 - 134.

[27] G. Barigazzi and L. Strigini, Application - Transparent Setting of Recovery Points, 13th Annual Int. Symposium on Fault Tolerant Computing, 1983, pp 48 - 55.

[28] D.L. Russell, State Restoration in Systems of Communicating Processes, IEEE Trans. on Software Engineering, Vol. SE - 6, No. 2, March 1980, pp 183 - 194.

[29] K.H. Kim, An Approach to Programmer Transparent Coordination of Recovering Parallel Processes and its Efficient Implementation Rules, Proc. Int. Conf. on Parallel Processing, Aug. 1978, pp 58 - 68.

[30] C.A.R. Hoare, Monitors: An Operating System Structuring Concept, Comm. ACM, Oct. 1974, pp 549 - 557.

[31] J.L. Peterson, Petri Net Theory and the Modeling of Systems, Prentice Hall, 1980.

[32] W.L. Heimerdinger, A Petri Net Approach to System Level Fault Tolerance Analysis, Proc. of the National Electronics Conference, Vol. 32, 1978, pp 161 - 165.

[33] Y.W. Han, Performance Evaluation of a Digital System using a Petri Net - Like Approach, Proc. of the National Electronics Conference, Vol. 32, 1978, pp 166 - 172.

[34] C.B. Jones, Software Development - A Rigorous Approach, Prentice Hall, 1980.

[35] C. Morgan, Schemas in Z : A Preliminary Reference Manual, Programming Research Group, Oxford, March 1984.

[36] D. May, Occam, Sigplan Notices, Vol. 18, No. 4, 1983, pp 69 - 79.

[37] D.L. Russell and M.J. Tiedeman, Multiprocess Recovery using Conversations, Proc. FTC - 9, 1979, pp 106 - 109.

[38] P. Brinch Hansen, The Architecture of Concurrent Programs, Englewood Cliffs, NJ: Prentice Hall, 1977.

[39] D. May and R. Taylor, Occam - An Overview, Microprocessors and Microsystems, Vol. 8, No. 2, March 1984, pp 73 - 79.

[40] R. Williamson and E. Horowitz, Concurrent Communication and Synchronization Mechanisms, Software - Practice and Experience, Vol. 14, No. 2, Feb. 1984, pp 135 - 151.

[41] A.S. Tanenbaum, Computer Networks, Prentice Hall, 1981.

[42] R.H. Campbell, K.H. Horton and G.G. Belford, Simulations of a Fault Tolerant Deadline Mechanism, Digest of papers FTCS - 9, Madison, June 1979, pp 95 - 101.

[43] J. Dennis, Concurrency in Software Systems, Advanced Course in Software Engineering, Ed. X. Eauer, 1973, pp 111 - 127.

[44] J.L. Peterson, An Introduction to Petri Nets, Proc. of the National Electronics Conference, Vol. 32, 1978, pp 144 - 148.

[45] S.S. Yau and S.M. Shatz, On Communication in the Design of Software Components of Distributed Computer Systems, 3rd Int. Conf. on Distributed Computing Systems, 1982, pp 280 - 287.

[46] C.A.R. Hoare, Communicating Sequential Processes, Comm. ACM, Vol. 21, No. 8, 1978, pp 666 - 677.

[47] L.J. Mekly and S.S. Yau, Software Design Representation using Abstract Process Networks, IEEE Trans. on Software Engineering, Vol. SE - 6, Sept 1980, pp 420 - 434.

[48] J.L. Peterson, Computation Sequence Sets, Journal of Computer and System Sciences, Vol. 13, No. 1, Aug. 1976, pp 1 - 24.

[49] E.W. Dijkstra, Notes on Structured Programming, in Structured Programming, O.J. Dahl, E.W. Dijkstra and C.A.R. Hoare, Academic Press, 1972.

[50] C.A.R. Hoare, Communicating Sequential Processes, Prentice Hall, 1985.

[51] E.W. Dijkstra, Co-operating Sequential Processes, in Programming Languages, Ed. F. Genuys, Academic Press, 1968.

[52] E.W. Dijkstra, Guarded Commands, Non-determinancy and Formal Derivation of Programs, Comm. ACM, Vol. 18, No. 8, 1975, pp 453 - 457.

[53] R. Shapiro and H. Saint, A New Approach to Optimization of Sequencing Decisions, Annual Review in Automatic Programming, Vol. 6, Part 5, 1970, pp 257 - 288.

[54] T. Anderson and J.C. Knight, A Framework for Software Fault Tolerance in Real Time Systems, IEEE Trans. on Software Engineering, Vol. SE - 9, May 1983, pp 355 - 364.

[55] S.K. Shrivastava and J-P. Banatro, Reliable Resource Allocation between Unreliable Processes, IEEE Trans. on Software Engineering, Vol. SE - 4, May 1978, pp 230 - 241.

[56] G.F. Carpenter, A.M. Tyrrell and D.J. Holding, Guidelines for the Synthesis of Software for Distributed Processes, PES3 Conference, Guernsey, March 1986, pp 164 - 175.

[57] J. Welsh and J Elder, Introduction to Pascal, Prentice Hall, 1982.

[58] J.M. Kerridge and D. Simpson, Three Solutions for a Robot Arm Controller using Pascal-Plus, occam and Edison, Software-Practice and Experience, Vol. 14, 1984, pp 3 - 15.

[59] P. Wilson, Programming System Builds Multiprocessor Software, Electronic Design, July 1983.

[60] D. May, Communicating Processes and Occam, Esprit Summer School on Future Parallel Computers, 1986.

[61] D. May and R. Shepherd, The Transputer Implementation of Occam, Esprit Summer School on Future Parallel Computers, 1986.

[62] F. Boussinot, R. Martin, G. Memmi, G. Ruggiu and J. Vapne, A Language for Formal Description of Real Time Systems, IFAC Safecomp, 1983, pp 119 - 126.

[63] B. Liskov, Primitives for Distributed Computing, Proc. 7th ACM SISOPS Symposium on Operating Systems Principles, Dec. 1979, pp 33 - 42.

[64] J.A. Feldman, High-Level Programming for Distributed Computing, Comm. ACM, Vol. 22, No. 6, June 1979, pp 353 - 368.

[65] Reference Manual for ADA Programming Language, United States Department of Defence, 1980.

[66] P. Brinch Hansen, Distributed Processes: A Concurrent Programming Concept, Comm. ACM, Vol. 21, No. 11, Nov. 1978, pp 934 - 941.

[67] T. Mao and T. Yeh, Communication Port: A Language Concept for Concurrent Programming, IEEE Trans. on Software Engineering, Vol. SE - 6, No. 2, March 1980, pp 194 - 204.

[68] M. Tsukamoto, Language Structures and Management Method in a Distributed Real Time Environment, 3rd IFAC Workshop on Distributed Computer Systems, 1981, pp 103 - 113.

[69] R.L. Grimsdale, F. Halsall, F. Martin-Polo and S. Wong, Structure and Tasking Features of the Programming Language Martlet, IEE Proc. Vol 129, Part E, No. 2, March 1982, pp 63 - 69.

[70] J. Kramer, J. Magee, M. Sloman and A. Lister, CONIC: An Integrated Approach to Distributed Computer Control Systems, IEE Proc. Vol. 130, Part E, No. 1, Jan. 1983, pp 1 - 10.

[71] K. Jackson and H.R. Simpson, MASCOT - A Modular Approach to Software Construction Operation and Test, RRE Technical Note No. 778, Oct. 1975.

[72] Ph. Darondeau, P. Le Guernic and M. Raynal, Abstract Specification of Communication Systems, Proc. 1st International Conference on Distributed Computing Systems, Oct. 1979, pp 339 - 346.

[73] S. Yau, C.C. Yang and S.M. Shatz, An Approach to Distributed Computer System Software Design, IEEE Trans. on Software Engineering, Vol. SE - 7, No. 4, July 1981, pp 427 - 436.

[74] J. Kramer, J. Magee and M. Sloman, Intertask Com-

munication Primitives for Distributed Computer Control Systems, Proc. of 2nd International Conference on Distributed Computer Systems, April 1981, pp 404 - 411.

[75] J.D. Noe, Hierarchical Modelling with Pro-Nets, Proc. of the National Electronics Conference, Vol. 32, 1978, pp 155 - 160.